Checking Race Freedom via Linear Programming

Tachio Terauchi
Tohoku University
terauchi@ecei.tohoku.ac.jp

Abstract

We present a new static analysis for race freedom and race detection. The analysis checks race freedom by reducing the problem to (rational) linear programming. Unlike conventional static analyses for race freedom or race detection, our analysis avoids explicit computation of locksets and lock linearity/must-aliasness. Our analysis can handle a variety of synchronization idioms that more conventional approaches often have difficulties with, such as thread joining, semaphores, and signals. We achieve efficiency by utilizing modern linear programming solvers that can quickly solve large linear programming instances. This paper reports on the formal properties of the analysis and the experience with applying an implementation to real world C programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Languages, Theory, Verification

Keywords Fractional Capabilities,Linear Programming

1. Introduction

Race condition occurs when one thread writes to a memory location that another thread is concurrently writing or reading. Race freedom, the absence of race conditions, is a basic building block for developing and verifying shared-memory parallel programs, and static analysis for race freedom and race detection has been an active focus of research.

In many static (or dynamic) analyses for race freedom or race detection, the central idea is to compute *locksets*. A lockset is the set of locks that are always held when accessing some memory location (abstract memory location in static analyses) such that a potential race is detected when the lockset is empty. It is important that the locks in the locksets are *linear* [20] (or *must-alias* [17]), in the sense that each lock corresponds to a unique concrete lock. Inferring locksets and lock linearity/must-aliasness statically can be non-trivial, especially in the presence of non-lexically scoped locks, that sometimes analyses make optimistic approximations. Also, this approach is usually limited to locks and other lock like synchronization idioms.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

e	::=	x	(*variable*)
	|	n	(*integer constant*)
	|	`let` x `=` e_1 `in` e_2	(*variable binding*)
	|	`if * then` e_1 `else` e_2	(*branch*)
	|	`while * do` e	(*loop*)
	|	`newtid`	(*new thread identifier*)
	|	`spawn`$(e_1)\{e_2\}$	(*new thread*)
	|	`join` e	(*thread join*)
	|	`ref` e	(*new reference cell*)
	|	$!e$	(*reference read*)
	|	e_1 `:=` e_2	(*reference write*)
	|	`newlock`	(*new lock*)
	|	`freelock` e	(*free lock*)
	|	`lock` e	(*lock acquire*)
	|	`unlock` e	(*lock release*)

Figure 1. The syntax of the simple concurrent language.

This paper presents a different approach for statically checking race freedom. The key idea is to reduce the race checking problem to a linear programming problem such that if there exists a solution to the constructed linear programming instance, then the program is guaranteed to be race free. In contrast to previous approaches, our approach only needs standard may-aliasing information, and does not require locksets and lock linearity/must-aliasness. By utilizing efficient linear programming algorithms, we achieve both good theoretical computational complexity (polynomial in the size of the program), and good practical running times, without sacrificing soundness. The approach can be extended to synchronization methods beyond simple lock-based idioms. The prototype implementation described in Section 3 handles programming idioms that other analyses often have difficulties with, such as thread joining, semaphores, signals, and read-write locks, as well as read only accesses and local accesses.

The rest of the paper is organized as follows. Section 2 introduces the key concepts with a toy language that contains only locks and thread spawning/joining, and formally proves soundness. Section 3 discusses the implementation, LP-Race, a tool for detecting races in multithreaded C programs. Section 4 discusses related work, Section 5 discusses open issues, and Section 6 concludes.

2. Formal Aspects

Figure 1 shows a simple first order expression language (with side effects) we use to present the key concepts of the analysis. The language is minimized in ordered to focus on the novel aspects of the analysis. We briefly describe the syntax. Variables are ranged over by meta variables x, x_1, etc. The construct `let` x `=` e_1 `in` e_2 binds the result of evaluating e_1 to x and evaluates e_2. We write $e_1; e_2$ for `let` x `=` e_1 `in` e_2 such that x is not free in e_2. The language contains non-deterministic branches and loops. The

```
let x = ref 0 in
  spawn(newtid){!x};
  x := 1
```
(a)

```
let x = ref 0 in
let t = newtid in
  spawn(t){!x};
  join t;
  x := 1
```
(b)

```
let x = ref 0 in
let l = newlock in
  spawn(newtid){
    lock l;
    !x;
    unlock l };
  lock l;
  x := 1;
  unlock l
```
(c)

Figure 2. Simple race (a), and race avoidance via thread joining (b) and locking (c).

$$
\begin{array}{rcl}
F & ::= & t.E \mid F||p \mid p||F \\
E & ::= & \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{ref}\ E \mid E := e \mid v := E \mid !E \\
 & \mid & \texttt{lock}\ E \mid \texttt{unlock}\ E \mid \texttt{freelock}\ E \\
 & \mid & \texttt{spawn}(E)\{e\} \mid \texttt{join}\ E
\end{array}
$$

Figure 3. The evaluation contexts

construct $\texttt{spawn}(e_1)\{e_2\}$ creates a new thread to evaluate e_2. Here, e_1 is a *thread identifier* that can be used at `join` e to join threads. Multiple threads are allowed to have the same thread identifier. The language contains reference cells, that could be written and read concurrently. Finally, the language contains syntax for creating, deleting, acquiring, and releasing locks.

Figure 2 (a) shows a simple example program that contains a read write race. The spawned thread may read from the reference cell bound to the variable `x` while the spawner thread writes to it. Such a race can be avoided by using locks, as shown in (c), or by using join to wait until the other thread finishes, as shown in (b).

2.1 The Semantics

We formally define the semantics of the language. The semantics is defined as small-step reductions from states to states. A *state* is a quadruple $(\iota, \kappa, \theta, p)$ where ι is the set of currently allocated thread identifiers, κ is the current *lock state*, a mapping from the currently available locks to $\{\texttt{U}, \texttt{L}\}$ where `U` denotes that the lock is unlocked and `L` denotes that the lock is locked, θ is a *store* mapping *locations* to *values*, and p is a *program state*. Values, v, are defined as

$$v ::= t \mid l \mid \ell \mid n$$

where the symbol t ranges over thread identifiers, l ranges over locks, and ℓ ranges over locations. A program state is defined as follows.

$$
\begin{array}{rcl}
e & ::= & \cdots \mid v \\
p & ::= & t.e \mid p_1 || \, p_2
\end{array}
$$

Here, e is extended with values. Intuitively, an (extended) expression prefixed by a thread identifier, $t.e$, denotes a thread with the thread identified by t whose current program counter is e, whereas $p_1 \ || \ p_2$ denotes a parallel composition of threads. Therefore, a program state is a parallel composition of finitely many expressions prefixed by thread identifiers. We let the parallel composition operator $||$ be commutative and associative.

We define the following standard notational convention. Given a mapping (i.e., a set-theoretic function) f, $f[a \mapsto b]$ denotes the mapping $\{c \mapsto f(c) \mid c \in \mathit{dom}(f) \setminus \{a\}\} \cup \{a \mapsto b\}$.

Figure 3 defines evaluation contexts. Figure 4 shows the reduction rules. **If1**, **If2**, **Loop**, and **Let** are straightforward. Note that because the language is an expression language, every expression needs to evaluate to a value if they terminate, and so we use 0 as the "unit" value for expressions that are evaluated purely for their effects.

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{if}\ *\ \texttt{then}\ e_1\ \texttt{else}\ e_2]) \to (\iota,\kappa,\theta,F[e_1])}\ \textbf{If1}$$

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{if}\ *\ \texttt{then}\ e_1\ \texttt{else}\ e_2]) \to (\iota,\kappa,\theta,F[e_2])}\ \textbf{If2}$$

$$\frac{}{\begin{array}{c}(\iota,\kappa,\theta,F[\texttt{while}\ *\ \texttt{do}\ e)) \to \\ (\iota,\kappa,\theta,F[\texttt{if}\ *\ \texttt{then}\ (e;\texttt{while}\ *\ \texttt{do}\ e)\ \texttt{else}\ 0])\end{array}}\ \textbf{Loop}$$

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{let}\ x = v\ \texttt{in}\ e]) \to (\iota,\kappa,\theta,F[e[v/x]])}\ \textbf{Let}$$

$$\frac{t \notin \mathit{dom}(\iota)}{(\iota,\kappa,\theta,F[\texttt{newtid}]) \to (\iota \cup \{t\},\kappa,\theta,F[t])}\ \textbf{NewT}$$

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{spawn}(t)\{e\}]) \to (\iota,\kappa,\theta,F[0] \,||\, t.e)}\ \textbf{Spawn}$$

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{join}\ t] \,||\, t.v) \to (\iota,\kappa,\theta,F[0])}\ \textbf{Join}$$

$$\frac{\ell \notin \mathit{dom}(\theta)}{(\iota,\kappa,\theta,F[\texttt{ref}\ v]) \to (\iota,\kappa,\theta[\ell \mapsto v],F[\ell])}\ \textbf{Ref}$$

$$\frac{}{(\iota,\kappa,\theta,F[!\ell]) \to (\iota,\kappa,\theta,F[\theta(\ell)])}\ \textbf{Read}$$

$$\frac{}{(\iota,\kappa,\theta,F[\ell := v]) \to (\iota,\kappa,\theta[\ell \mapsto v],F[0])}\ \textbf{Write}$$

$$\frac{l \notin \mathit{dom}(\kappa)}{(\iota,\kappa,\theta,F[\texttt{newlock}]) \to (\iota,\kappa[l \mapsto \texttt{U}],\theta,F[l])}\ \textbf{NewL}$$

$$\frac{\kappa = \kappa' \cup \{l \mapsto U\} \quad l \notin \mathit{dom}(\kappa')}{(\iota,\kappa,\theta,F[\texttt{freelock}\ l]) \to (\iota,\kappa',\theta,F[0])}\ \textbf{FreeL}$$

$$\frac{\kappa(l) = \texttt{U}}{(\iota,\kappa,\theta,F[\texttt{lock}\ l]) \to (\iota,\kappa[l \mapsto \texttt{L}],\theta,F[0])}\ \textbf{Lck}$$

$$\frac{}{(\iota,\kappa,\theta,F[\texttt{unlock}\ l]) \to (\iota,\kappa[l \mapsto \texttt{U}],\theta,F[0])}\ \textbf{Ulck}$$

Figure 4. The operational semantics.

NewT creates a fresh thread identifier. **Spawn** spawns a new thread using the given thread identifier, and **Join** waits for a thread with the thread identifier to finish. **Ref** allocates a new reference cell that can be read by **Read** and written by **Write**. **NewL** creates a new lock, initialized to unlocked status. If the lock is unlocked, **Lck** may acquire the lock, setting the lock status to locked. **Ulck** releases the lock, setting the lock status to unlocked. **FreeL** deletes a lock if the lock is unlocked[1].

We write $(\iota_1,\kappa_1,\theta_1,p_1) \ \to^* \ (\iota_2,\kappa_2,\theta_2,p_2)$ for zero or more reduction steps from the state $(\iota_1,\kappa_1,\theta_1,p_1)$ to the state $(\iota_2,\kappa_2,\theta_2,p_2)$. We now formally define race freedom.

DEFINITION 2.1 (Race Freedom). *A state $(\iota_1,\kappa_1,\theta_1,p_1)$ is said to be race free if for any state $(\iota_2,\kappa_2,\theta_2,p_2)$ such that $(\iota_1,\kappa_1,\theta_1,p_1) \to^* (\iota_2,\kappa_2,\theta_2,p_2)$, p_2 is not of the following form.*

- $F_1[\ell := v_1] \ || \ F_2[\ell := v_2]$

[1] This models `pthread_mutex_destroy` in the POSIX threads library.

- $F_1[\ell := v] \mid\mid F_2[!\ell]$

2.2 The Type System

We formulate the analysis as a type inference problem for a type system. The type system guarantees that a typable program is race free. The types are defined as follows.

$$\begin{array}{rcll} \tau & ::= & \mathit{ref}(\rho, \tau) & (\textit{reference cells}) \\ & \mid & \mathit{int} & \textit{integers} \\ & \mid & \mathit{lock}(\Psi) & (\textit{locks}) \\ & \mid & \mathit{tid}(\Psi) & (\textit{thread identifiers}) \end{array}$$

The type $\mathit{ref}(\rho, \tau)$ denotes a type of a reference cell pointing to the *abstract location* ρ, storing a value of the type τ. Symbols Ψ, Ψ_1, etc. range over *capability mappings*. A capability mapping is a function from abstract locations to non-negative rational numbers $[0, \infty)$.

Capability mappings denote access capabilities to abstract locations. Each thread holds some amount of capabilities, representing the access capabilities of the thread. Intuitively, a thread holding capabilities Ψ such that $\Psi(\rho) \geq 1$ is allowed to write to the abstract location ρ, and a thread holding capabilities Ψ such that $\Psi(\rho) > 0$ is allowed read from the abstract location ρ (thus the write capability implies the read capability). The type system ensures that the total amount of capabilities summed across all live threads is at most 1 for any abstraction location. This property ensures race freedom as there cannot be two threads, say holding capabilities Ψ_1 and Ψ_2 respectively, such that one thread can write to an abstract location ρ, (i.e., $\Psi_1(\rho) \geq 1$) while the other thread can read or write to it (i.e., $\Psi_2(\rho) > 0$), because then the total amount of capabilities for ρ would exceed 1.

Threads may *transfer* capabilities at synchronization points, which in this simple language, is when accessing locks and spawning and joining threads. The capabilities Ψ appearing in a lock type $\mathit{lock}(\Psi)$ represents the amount of capabilities transferred to the thread when acquiring the lock, and transferred from when releasing the lock. The capabilities Ψ appearing in a thread identifier type $\mathit{tid}(\Psi)$ represents the amount of capabilities transferred to the joiner thread by joining a thread with the thread identifier.

Figure 5 shows the type checking rules. The judgement are of the form $\Gamma, \Psi_1, \vdash e : \tau, \Psi_2$ where Γ is a type environment mapping variables to their types, the *pre-capability* Ψ_1 is the capabilities before the evaluation of e, the *post-capability* Ψ_2 is the capabilities after the evaluation of e, and τ is the type of e. **VAR** and **INT** are self-explanatory. **LET**, **IF**, **WHILE** make sure that capabilities are "conserved" (i.e., not created out of thin air) through the sequential flow of computation. The inequality $\Psi_1 \geq \Psi_2$ is defined as $\forall \rho. \Psi_1(\rho) \geq \Psi_2(\rho)$.

Also, the subtraction of capabilities is defined point-wise as $\Psi_1 - \Psi_2 = \lambda \rho. \Psi_1(\rho) - \Psi_2(\rho)$. Note that because the range of any capability mapping is restricted to non-negative rationals, the subtraction is undefined if the result is negative. Similarly, the addition of capabilities is defined as $\Psi_1 + \Psi_2 = \lambda \rho. \Psi_1(\rho) + \Psi_2(\rho)$.

NEWT, **SPAWN**, and **JOIN** type thread creation and thread joining. At **SPAWN**, the parent thread gives part of its capabilities, Ψ_3, to the newly created thread, and so $\Psi_2 - \Psi_3$ amount of capabilities are left for the continuation of the parent thread. The capabilities that are left after the spawned thread finishes, Ψ_4, or at least a part of it (Ψ_1), may be recovered by using the thread identifier. At **JOIN**, the joiner thread gains capabilities from the joined thread through the thread identifier.

REF, **WRITE**, and **READ** type reference cell accesses. As remarked above, **READ** requires a positive amount of capability for the abstract location, and **WRITE** requires capabilities at least 1. Because there is no notion of lockset, these rules do not assert anything about which locks protect the reference cell. Note that

$$\frac{}{\Gamma, \Psi \vdash x : \Gamma(x), \Psi}\ \textbf{VAR} \qquad \frac{}{\Gamma, \Psi \vdash n : \mathit{int}, \Psi}\ \textbf{INT}$$

$$\frac{\Gamma, \Psi \vdash e_1 : \tau_1, \Psi_1 \quad \Gamma[x \mapsto \tau_1], \Psi_1 \vdash e_2 : \tau_2, \Psi_2}{\Gamma, \Psi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, \Psi_2}\ \textbf{LET}$$

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e_1 : \tau, \Psi_1 \quad \Psi_1 \geq \Psi_3 \\ \Gamma, \Psi \vdash e_2 : \tau, \Psi_2 \quad \Psi_2 \geq \Psi_3\end{array}}{\Gamma, \Psi \vdash \texttt{if} * \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau, \Psi_3}\ \textbf{IF}$$

$$\frac{\Gamma, \Psi_1 \vdash e : \tau, \Psi_2 \quad \Psi_2 \geq \Psi_1 \quad \Psi \geq \Psi_1}{\Gamma, \Psi \vdash \texttt{while} * \texttt{do}\ e : \mathit{int}, \Psi_1}\ \textbf{WHILE}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{newtid} : \mathit{tid}(\Psi_1), \Psi}\ \textbf{NEWT}$$

$$\frac{\Gamma, \Psi \vdash e_1 : \mathit{tid}(\Psi_1), \Psi_2 \quad \Gamma, \Psi_3 \vdash e_2 : \tau, \Psi_4 \quad \Psi_4 \geq \Psi_1}{\Gamma, \Psi \vdash \texttt{spawn}(e_1)\{e_2\} : \mathit{int}, \Psi_2 - \Psi_3}\ \textbf{SPAWN}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{tid}(\Psi_1), \Psi_2}{\Gamma, \Psi \vdash \texttt{join}\ e : \mathit{int}, \Psi_1 + \Psi_2}\ \textbf{JOIN}$$

$$\frac{\Gamma, \Psi \vdash e : \tau, \Psi_1}{\Gamma, \Psi \vdash \texttt{ref}\ e : \mathit{ref}(\rho, \tau), \Psi_1}\ \textbf{REF}$$

$$\frac{\Gamma, \Psi \vdash e_1 : \mathit{ref}(\rho, \tau), \Psi_1 \quad \Gamma, \Psi_1 \vdash e_2 : \tau, \Psi_2 \quad \Psi_2(\rho) \geq 1}{\Gamma, \Psi \vdash e_1 := e_2 : \mathit{int}, \Psi_2}\ \textbf{WRITE}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{ref}(\rho, \tau), \Psi_1 \quad \Psi_1(\rho) > 0}{\Gamma, \Psi \vdash !e : \tau, \Psi_1}\ \textbf{READ}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{newlock} : \mathit{lock}(\Psi_1), \Psi - \Psi_1}\ \textbf{NEWL}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{lock}(\Psi_1), \Psi_2}{\Gamma, \Psi \vdash \texttt{freelock}\ e : \mathit{int}, \Psi_2 + \Psi_1}\ \textbf{FREEL}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{lock}(\Psi_1), \Psi_2}{\Gamma, \Psi \vdash \texttt{lock}\ e : \mathit{int}, \Psi_2 + \Psi_1}\ \textbf{LCK}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{lock}(\Psi_1), \Psi_2}{\Gamma, \Psi \vdash \texttt{unlock}\ e : \mathit{int}, \Psi_2 - \Psi_1}\ \textbf{ULCK}$$

Figure 5. The type checking rules.

READ naturally allows parallel reads (i.e., read-only accesses), because n threads may each hold $1/n$ amount of capabilities for the same abstract location and still satisfy the "at most 1" property.

NEWL types lock creations. As remarked above, releasing a lock loses the amount of capabilities associated with the lock. Because new locks are initialized to the unlocked state, we subtract the capabilities as if the lock is unlocked. Dually, **FREEL** gets back the capability associated with the lock for destroying the lock. At **LCK**, the thread gains the capabilities, and at **ULCK**, the capabilities are lost.

The simplicity of these rules may appear deceptive. Researchers familiar with lockset-based race analysis might have expected to see a rule requiring the locks to be *linear* [20] (or *must-alias* [17]). A formal proof of correctness appears at the end of the section. To see how the type system avoids linearity/must-aliasness requirement, suppose the rule **NEWL** is replaced by the following un-

sound rule.

$$\frac{}{\Gamma, \Psi \vdash \texttt{newlock} : \mathit{lock}(\Psi_1), \Psi} \textbf{ NEWL-unsound}$$

Consider the following program.

```
let x = ref 0 in
let l1 = newlock in
let l2 = newlock in
  spawn(newtid){
    lock l1; !x; unlock l1 };
  lock l2; x := 1; unlock l2
```

The program has a race on x because two threads concurrently access x (by holding different locks). However, with **NEWL-unsound**, the program would type check (cf. Definition 2.3) by assigning the type $\mathit{lock}(\Psi)$ to both l1 and l2 such that $\Psi(\rho) = 1$ where x has the type $\mathit{ref}(\rho, \mathit{int})$. **NEWL** prevents such a situation by making sure that capabilities (like Ψ above) are not created "out of thin air". In practice, this implies that if the type system cannot distinguish two locks that are alive at the same time such that at least one of them is used to guard some location, then the type system may report a false positive because the total capability would exceed 1 at one of the lock allocation point. The locality extension discussed in Section 3.1.5 can soundly allow may-aliased live locks to be used to guard locations in some situations.

We now define the notion of a well-typed state. We extend type environments so that they map values to types as well as variables. Non-integer values are typed by the following rule.

$$\frac{}{\Gamma, \Psi \vdash v : \Gamma(v), \Psi}$$

We define the notion of a well-typed store.

DEFINITION 2.2 (Well-typed Store). *We write $\Gamma \vdash \theta$ if for each $\ell \in \mathit{dom}(\theta)$, $\Gamma, \mathit{0} \vdash \theta(\ell) : \tau, \mathit{0}$ where $\Gamma(\ell) = \mathit{ref}(\rho, \tau)$ for some ρ.*

Here, $\mathit{0}$ is the null capability, defined as $\mathit{0} = \lambda\rho.0$. Threads are typed by the following rule, which says that the capabilities left at the end of the thread is at least the capabilities obtainable by joining the thread.

$$\frac{\Gamma, \Psi \vdash e : \Psi_1 \quad \Gamma(t) = \mathit{tid}(\Psi_2) \quad \Psi_1 \geq \Psi_2}{\Gamma, \Psi \vdash t.e : \mathit{int}, \mathit{0}}$$

Let $\mathit{cap}(\mathit{lock}(\Psi)) = \Psi$. Recall that any program state is a parallel composition of finitely many threads, that is, it is of the form $t_1.e_1 \mid\mid t_2.e_2 \mid\mid \ldots \mid\mid t_n.e_n$ where n is the number of threads.

DEFINITION 2.3 (Well-typed State). *Let $p = t_1.e_1 \mid\mid \ldots \mid\mid t_n.e_n$ such that there are no free variables in p. We write $\Gamma \vdash (\iota, \kappa, \theta, p)$ if there exist $\Psi_1, \ldots, \Psi_n$ such that*

(1) For all $t \in \iota$, $\Gamma(t)$ is a thread identifier type.
(2) For all $l \in \mathit{dom}(\kappa)$, $\Gamma(l)$ is a lock type.
(3) $\Gamma \vdash \theta$.
(4) For all $i \in \{1, \ldots, n\}$, $\Gamma, \Psi_i \vdash t_i.e_i : \mathit{int}, \mathit{0}$.
(5) Let $U = \{l \mid \kappa(l) = \mathtt{U}\}$. Let

$$\Psi = \sum_{l \in U} \mathit{cap}(\Gamma(l)) + \sum_{i=1}^{n} \Psi_i$$

Then $\forall \rho.\Psi(\rho) \leq 1$

The first three conditions ensure that ι, κ, θ are well-typed. The condition (4) states that all threads are well-typed. The condition (5) asserts that the total amount of capabilities summed across all threads (i.e.., $\sum_{i=1}^{n} \Psi_i$) and the amount of capabilities obtainable by acquiring locks (i.e., $\sum_{l \in U} \mathit{cap}(\Gamma(l))$) is at most 1 for any abstract location.

Well-typedness is preserved across reductions.

LEMMA 2.4 (Preservation). *Suppose $\Gamma \vdash (\iota_1, \kappa_1, \theta_1, p_1)$ and $(\iota_1, \kappa_1, \theta_1, p_1) \rightarrow (\iota_2, \kappa_2, \theta_2, p_2)$. Then, there exists $\Gamma' \supseteq \Gamma$ such that $\Gamma' \vdash (\iota_2, \kappa_2, \theta_2, p_2)$.*

Proof: By case analysis on the reduction. □

Note that the condition (5) of Definition 2.3 implies that a well-typed state cannot have two threads such that one thread is trying to write to a location and the other thread is accessing the same location. More formally,

LEMMA 2.5. *Suppose p is of the form $F_1[\ell := v_1] \mid\mid F_2[\ell := v_2]$ or $F_1[\ell := v] \mid\mid F_2[!\ell]$. Then for no Γ, $\Gamma \vdash (\iota, \kappa, \theta, p)$.*

The soundness of the type system follows from the above lemmas.

THEOREM 2.6. *Suppose $\Gamma \vdash (\iota, \kappa, \theta, p)$. Then $(\iota, \kappa, \theta, p)$ is race free.*

Proof: Straightforward from Lemma 2.4, Lemma 2.5, and Definition 2.1. □

The type system is inspired by research on *fractional permissions/capabilities*. Fractional permissions were originally invented to guarantee determinism of multithreaded programs in the presence of parallel reads [5]. The idea has been adopted in concurrent separation logic [3], and has also been used to statically check determinism of channel communicating processes [21].

2.2.1 Example

Consider the following example which spawns threads in a loop, and uses locks and joins to avoid races to the shared reference cells bound to x and y.

```
let x = ref 0 in let y = ref 0 in
let t = newtid in let l = newlock in
  while * do
    spawn(t){let z = !y in lock l; x := z; unlock l};
    spawn(t){let z = !y in lock l; x := z; unlock l};
    join t; join t; y := 1
```

Let e be the code above, and let $p = t_1.e$. Consider the state $(\{t_1\}, \emptyset, \emptyset, p)$. Let $\Psi_1 = \mathit{0}[\rho_x \mapsto 1][\rho_y \mapsto 1]$. Then we have $\emptyset, \Psi_1 \vdash (\{t_1\}, \emptyset, \emptyset, p)$, guaranteeing that Figure 2 (b) is race free. The type derivation uses a type environment of the form

$$\{\ t_1 \mapsto \mathit{tid}(\mathit{0}), \mathtt{x} \mapsto \mathit{ref}(\rho_x, \mathit{int}), \mathtt{y} \mapsto \mathit{ref}(\rho_y, \mathit{int})$$
$$\mathtt{t} \mapsto \mathit{tid}(\mathit{0}[\rho_y \mapsto 0.5]), \quad \mathtt{l} \mapsto \mathit{lock}(\mathit{0}[\rho_x \mapsto 1])\}$$

to type check the main body of e. Note that the type of t indicates that the spawned thread gets a fraction of the capability to access y in read-only mode, which is combined at joins so that the main thread can write to y after the threads finish. The type of l indicates that the spawned threads get the full (i.e., write) capability for x by acquiring l.

2.3 The Analysis Algorithm

Intuitively, the analysis algorithm is a type inference algorithm for the type system presented in Section 2.2. The analysis is separated in two phases. Informally, the first phase infers everything about the type derivation except for the amount of capabilities. The second phase uses linear programming to check if there exist an assignment of capabilities that satisfies the capability constraints.

2.3.1 Phase 1

The first phase is mostly a standard unification-based inference, generating capability constraints on the side. Figure 6 shows the constraint generation rules. Here, α's are type variables, ϱ's are abstract location variables, and φ's are capability mapping variables. These rules are straightforward syntax-directed inference rules for the type rules from Figure 5.

$$\frac{\varphi \text{ fresh}}{\Delta, \varphi \vdash x : \Delta(x), \varphi; \emptyset} \quad \frac{\alpha, \varphi \text{ fresh}}{\Delta, \varphi \vdash n : \alpha, \varphi; \{\alpha = \mathit{int}\}}$$

$$\frac{\Delta, \varphi \vdash e_1 : \alpha_1, \varphi_1; C_1 \quad \Delta, \varphi_2 \vdash e_2 : \alpha_2, \varphi_3; C_2}{\Delta, \varphi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \alpha_2, \varphi_3; C_1 \cup C_2 \cup \{\alpha_1 = \Delta(x), \varphi_1 = \varphi_2\}}$$

$$\frac{\Delta, \varphi \vdash e_1 : \alpha, \varphi_1; C_1 \quad \Delta, \varphi' \vdash e_2 : \alpha', \varphi_1'; C_2}{\Delta, \varphi \vdash \texttt{if * then}\ e_1\ \texttt{else}\ e_2 : \alpha, \varphi_2; C_1 \cup C_2 \cup \{\alpha = \alpha', \varphi = \varphi', \varphi_1 \geq \varphi_2, \varphi_1' \geq \varphi_2\}}$$

$$\frac{\Delta, \varphi_1 \vdash e : \alpha, \varphi_2; C \quad \alpha_2, \varphi \text{ fresh}}{\Delta, \varphi \vdash \texttt{while * do}\ e : \alpha_2, \varphi_1; C \cup \{\alpha_2 = \mathit{int}, \varphi_2 \geq \varphi_1, \varphi \geq \varphi_1\}} \quad \frac{\alpha, \varphi, \varphi_1 \text{ fresh}}{\Delta, \varphi \vdash \texttt{newtid} : \alpha, \varphi; \{\alpha = \mathit{tid}(\varphi_1)\}}$$

$$\frac{\Delta, \varphi \vdash e_1 : \alpha_1, \varphi_2; C_1 \quad \Delta, \varphi_3 \vdash e_2 : \alpha_2, \varphi_4; C_2 \quad \alpha, \varphi_1, \varphi_5 \text{ fresh}}{\Delta, \varphi \vdash \texttt{spawn}(e_1)\{e_2\} : \alpha, \varphi_5; C_1 \cup C_2 \cup \{\alpha = \mathit{int}, \alpha_1 = \mathit{tid}(\varphi_1), \varphi_4 \geq \varphi_1, \varphi_5 = \varphi_2 - \varphi_3\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha, \varphi_2; C \quad \alpha_1, \varphi_1, \varphi_1 \text{ fresh}}{\Delta, \varphi \vdash \texttt{join}\ e : \alpha_1, \varphi_3; C \cup \{\alpha = \mathit{tid}(\varphi_1), \alpha_1 = \mathit{int}, \varphi_3 = \varphi_1 + \varphi_2\}} \quad \frac{\Delta, \varphi \vdash e : \alpha, \varphi_1; C \quad \alpha_1, \varrho \text{ fresh}}{\Delta, \varphi \vdash \texttt{ref}\ e : \alpha_1, \varphi_1; C \cup \{\alpha_1 = \mathit{ref}(\varrho, \alpha)\}}$$

$$\frac{\Delta, \varphi \vdash e_1 : \alpha_1, \varphi_1; C_1 \quad \Delta, \varphi_1' \vdash e_2 : \alpha_2, \varphi_2; C_2 \quad \alpha, \varrho \text{ fresh}}{\Delta, \varphi \vdash e_1\ \texttt{:=}\ e_2 : \alpha, \varphi_2; C_1 \cup C_2 \cup \{\alpha = \mathit{int}, \alpha_1 = \mathit{ref}(\varrho, \alpha_2), \varphi_1 = \varphi_1', \varphi_2(\varrho) \geq 1\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha, \varphi_1; C \quad \alpha_1, \varrho \text{ fresh}}{\Delta, \varphi \vdash !e : \alpha_1, \varphi_1; C \cup \{\alpha = \mathit{ref}(\varrho, \alpha_1), \varphi_1(\varrho) > 0\}} \quad \frac{\alpha, \varphi_1, \varphi' \text{ fresh}}{\Delta, \varphi \vdash \texttt{newlock} : \alpha, \varphi'; \{\alpha = \mathit{lock}(\varphi_1), \varphi' = \varphi - \varphi_1\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha, \varphi_1; C}{\Delta, \varphi \vdash \texttt{freelock}\ e : \alpha', \varphi_3; C \cup \{\alpha' = \mathit{int}, \alpha = \mathit{lock}(\varphi_2), \varphi_3 = \varphi_1 + \varphi_2\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha, \varphi_2; C \quad \alpha_1, \varphi_1, \varphi_3 \text{ fresh}}{\Delta, \varphi \vdash \texttt{lock}\ e : \alpha_1, \varphi_3; C \cup \{\alpha = \mathit{lock}(\varphi_1), \alpha_1 = \mathit{int}, \varphi_3 = \varphi_2 + \varphi_1\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha, \varphi_2; C \quad \alpha_1, \varphi_1, \varphi_3 \text{ fresh}}{\Delta, \varphi \vdash \texttt{unlock}\ e : \alpha_1, \varphi_3; C \cup \{\alpha = \mathit{lock}(\varphi_1), \alpha_1 = \mathit{int}, \varphi_3 = \varphi_2 - \varphi_1\}}$$

Figure 6. The type inference rules.

The inference judgement, $\Delta, \varphi_1 \vdash e : \alpha, \varphi_2; C$, reads "under the environment Δ, e is inferred to have the type α, the pre-capability φ_1, the post-capability φ_2, with the set of constraints C." For simplicity, we assume that `let`-bound variables are distinct. We initialize Δ to map each variable to a fresh type variable. We visit each AST node (i.e., expression) in a bottom up manner to build the set of constraints.

The resulting set of constraints contains three kinds of constraints:

(a) Type unification constraints: $\sigma = \sigma'$.

(b) Capability (in)equality constraints: $\phi = \phi'$ and $\phi \geq \phi'$.

(c) Access constraints: $\varphi(\varrho) \geq 1$ and $\varphi(\varrho) > 0$.

where

$$\sigma ::= \alpha \mid \mathit{ref}(\varrho, \alpha) \mid \mathit{int} \mid \mathit{lock}(\varphi) \mid \mathit{tid}(\varphi)$$
$$\phi ::= \varphi \mid \phi + \phi \mid \phi - \phi$$

The constraints of the kind (a) can be resolved by the standard unification algorithm, which may create more constraints the kind (b). In addition, it creates constraints of the form $\varrho_1 = \varrho_2$, which can also be resolved by the standard unification algorithm. This leaves us with a set of constraints of the kind (b) and (c).

Because we used simple types for the sake of exposition, the analysis algorithm may fail at this point, for example, when the program uses integers as locks. We may reject the program at this point. But it is easy to extend the system with sum types and recursive types so that this phase never fails. We take such an approach in the implementation described in Section 3.

2.3.2 Phase 2

The second phase of the algorithm finds a satisfying solution to the remaining constraints generated in the first phase so that the program is well-typed. We reduce the problem to linear programming as follows. Let e be the program being analyzed. Phase 1 returns pre-capability φ_e such that $\Delta, \varphi_e \vdash e : \tau, \varphi'; C$ for some τ, φ' and C.

For each ϱ (that is, its equivalence class obtained by the unification in phase 1), we instantiate a linear programming problem using the remaining constraints together with the constraint $\varphi_e(\varrho) \leq 1$. More precisely, each constraint mapping variables φ is instantiated as a linear programming variable $\varphi(\varrho)$, and access constraints $\varphi(\varrho') \geq 1$ and $\varphi(\varrho') > 0$ are removed from the constraints if $\varrho' \neq \varrho$. We add constraints $\varphi(\varrho) \geq 0$ for each φ to ensure that capabilities are non-negative.

To apply linear programming algorithms that can only take non-strict inequalities such as the one implemented in GLPK [1], we add a fresh linear programming variable ϵ and replace each $\varphi(\varrho) > 0$ with $\varphi(\varrho) \geq \epsilon$, and set the objective function to be ϵ. We ask the linear programming solver to find a solution that maximizes ϵ. If the solver returns a solution such that $\epsilon > 0$, then we accept the program as race free on the location ϱ. Otherwise, we report a possible race on ϱ.

A write-write race is reported if the linear programming solver cannot find any solution. A read-write race is reported if the linear programming solver finds a solution but $\epsilon = 0$.

2.3.3 Analysis of the Algorithm

We prove the correctness of the analysis algorithm. We use the symbol η to denote a *constraint solution*, which is a sorted substitution mapping type variables to types, abstract location variables to abstract locations, and capability mapping variables to capability mappings. A constraint solution becomes a mapping from σ, Δ, and ϕ in the obvious way.

DEFINITION 2.7. *We write $\eta \models C$ ("η solves C") if*

- *for each $\sigma = \sigma' \in C$, $\eta(\sigma) = \eta(\sigma')$.*
- *for each $\phi \geq \phi' \in C$, $\eta(\phi) \geq \eta(\phi')$.*
- *for each $\phi = \phi' \in C$, $\eta(\phi) = \eta(\phi')$.*
- *for each $\phi(\varrho) \geq 1 \in C$, $\eta(\phi)(\eta(\varrho)) \geq 1$.*
- *for each $\phi(\varrho) > 0 \in C$, $\eta(\phi)(\eta(\varrho)) > 0$.*
- *for each $\phi(\varrho) \leq 1 \in C$, $\eta(\phi)(\eta(\varrho)) \leq 1$.*

LEMMA 2.8. *Suppose $\Delta, \varphi_1 \vdash e : \alpha, \varphi_2; C$ and $\eta \models C$. Then, $\eta(\Delta), \eta(\varphi_1) \vdash e : \eta(\alpha), \eta(\varphi_2)$.*

Proof: By induction on the type derivation. □

THEOREM 2.9 (Soundness). *Suppose $\Delta, \varphi_e \vdash e : \tau, \varphi; C$ and*

$$\eta \models \bigcup_{\varrho} \{\varphi_e(\varrho) \leq 1\} \cup C$$

Let $\Gamma = \eta(\Delta)$. Let $x_1, \ldots, x_n$ be the free variables in e. Let $v_1, \ldots, v_n$ be such that $\Gamma \vdash v_i : \Gamma(x_i)$ for each v_i. Let ι, κ, θ be such that $\iota \cup dom(\kappa) \cup dom(\theta) \cup \mathbb{Z} \supseteq \{v_1, \ldots, v_n\}$, $\Gamma \vdash \theta$, and $\kappa(l) = \mathtt{L}$ for each $l \in dom(\kappa)$. Let $t \notin \iota$. Then, $(\{t\} \cup \iota, \kappa, \theta, t.e[v_1/x_1] \ldots [v_n/x_n])$ is race free.

Proof: Straightforward from Lemma 2.8 and Theorem 2.6. □

We argue the theoretical computational complexity of the analysis algorithm. The instance of linear programming problem in phase 2 can be solved in time polynomial in the size of the constraints by algorithms such as interior points methods. Therefore, the complexity of the algorithm is polynomial in the time phase 1 takes to generate the capability constraints, which is polynomial in the size of the program for our simple language. Therefore, the complexity of the analysis algorithm is polynomial in the size of the program.

In general, the complexity will increase if we include more complex programming constructs such as data structures and higher order functions if we stick with the simple types. But this can be avoided by incorporating recursive types, as is done in the LP-Race implementation.

3. LP-Race

We have implemented a prototype of the analysis algorithm, LP-Race, a tool for detecting races in C programs. LP-Race uses CIL [19] as a front-end to parse C files and handles the full-set of C.

3.1 Handling C Features

LP-Race extends the analysis framework to handle C features not covered in the formalism detailed in Section 2. This includes structs and unions, functions, and synchronization methods such as signaling and semaphores. This section highlights some of the notable extensions.

3.1.1 Alias Analysis

C programs use pointers and arrays extensively. To generate a sensible set of abstract locations, LP-Race performs points-to analysis and use the computed may-alias sets as abstract locations. For the prototype implementation, we choose one-level-flow analysis [6, 7] (with optimistic field sensitivity[2]), which is fast and known to produce good alias sets in practice. In principle, any may alias analysis can be used to obtain abstract locations.

As remarked earlier, LP-Race uses sum types and recursive types so that the first phase of the analysis never fails. This allows, among other things, LP-Race to report all single-threaded C programs to be race free.

3.1.2 Generic Control Flow

C contains unstructured control flow such as gotos and breaks. To handle generic control flow, LP-Race uses CIL to generate a control flow graph for each function, and associate a fresh capability for each node in the control flow graph. Then, for each successor node b of a node a, LP-Race adds the constraint $\varphi_a \geq \varphi_b$ where φ_a is the capability associated with a and φ_b is the capability associated with b.

3.1.3 Functions

Because threads are typically created using function pointers, handling first class functions is crucial for analyzing multithreaded C programs. We extend the type system with function types of the form

$$\tau ::= \cdots \mid (\Psi_{pre}, \vec{\tau}) \rightarrow (\Psi_{post}, \tau_{ret})$$

where $\vec{\tau}$ are the arguments types (the notation $\vec{a}$ denotes a sequence), and τ_{ret} is the return type. Intuitively Ψ_{pre} is the capability that the caller of the function must be holding, and Ψ_{post} is the capability that can be returned to the caller when the function returns. Ψ_{pre} is taken from the entry node of the function body, and Ψ_{ret} is the solution for the capability variable φ_{ret} such that for each return node a in the function, LP-Race adds the constraint $\varphi_a \geq \varphi_{ret}$ where φ_a is the capability associated with a.

LP-Race type check function calls by the following rule.

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e : (\Psi_{pre}, \tau_1, \ldots, \tau_n) \rightarrow (\Psi_{post}, \tau_{ret}), \Psi_1 \\ \forall i \in \{1, \ldots, n\}.(\Gamma, \Psi_i \vdash e_i : \tau_i, \Psi_{i+1}) \\ \Psi_{n+1} = \Psi_{pre} + \Psi_{keep}\end{array}}{\Gamma, \Psi \vdash e(e_1, \ldots, e_n) : \tau_{ret}, \Psi_{post} + \Psi_{keep}}$$

Here, Ψ_{n+1} is the capability held by the caller just before entering the function. Note that only a part of Ψ_{n+1}, that is Ψ_{pre}, needs to be given to the function. The remaining capabilities, Ψ_{keep} is kept by the caller and combined with the return capability of the function. This capability "flow around" technique provides context sensitivity as each call site can use a different Ψ_{keep} to avoid conflating capabilities.

The flow around technique is inspired by similar ideas used in Cqual [12] and Locksmith [20]. However, unlike Cqual or Locksmith, LP-Race does not require an effect analysis to determine what to give to the function and what to keep, because the flow around relation becomes just an additional set of linear equations so that linear programming automatically discovers what to flow around. Also, it is more general because it allows fractional amount of capabilities to be flown around.

Polymorphic Function Signatures A polymorphic (i.e., a context sensitive) alias analysis [8, 7, 23, 14] can be used to generate polymorphic types for functions. That is, we can quantify function

[2] The fields of a struct/union are allowed to have different types and abstract locations.

types by abstract locations so that functions are given types of the form

$$\forall\vec{\rho}.(\Psi_{pre},\vec{\tau}) \rightarrow (\Psi_{post},\tau_{ret})$$

This allows us to type check situations requiring parametric polymorphism, as in the code below.

```
int c, d;
pthread_t tid1, tid2;

void *f(void *p) {
  *p = 1;
}
void main(void) {
  pthread_create(&tid1, NULL, &f, &c);
  pthread_create(&tid2, NULL, &f, &d);
}
```

Currently, LP-Race uses monomorphic one-level-flow analysis and so polymorphic function signatures cannot be obtained. We leave extending LP-Race with polymorphic function signatures for future work.

3.1.4 Synchronization Primitives

As remarked earlier, our analysis approach is not limited to locks. Here, we discuss other kinds of synchronization primitives LP-Race handles.

Signaling Perhaps the simplest form of synchronization is to send a signal from one thread to another thread waiting for a signal. For example, POSIX threads programs use conditional variables for signaling. LP-Race gives signal primitives like a conditional variable the type of the form $\mathit{sig}(\Psi)$ so that a send of a signal is typed as follows.

$$\frac{\Gamma,\Psi \vdash e : \mathit{sig}(\Psi_1),\Psi_2}{\Gamma,\Psi \vdash \texttt{send}\ e : \mathit{int},\Psi_2-\Psi_1}$$

And a wait on a signal is typed as follows.

$$\frac{\Gamma,\Psi \vdash e : \mathit{sig}(\Psi_1),\Psi_2}{\Gamma,\Psi \vdash \texttt{wait}\ e : \mathit{int},\Psi_2+\Psi_1}$$

Semaphores Semaphores are straightforward to handle in our framework. A semaphore is given the type of the form $\mathit{sem}(\Psi)$. A post of a semaphore is typed as follows.

$$\frac{\Gamma,\Psi \vdash e : \mathit{sem}(\Psi_1),\Psi_2}{\Gamma,\Psi \vdash \texttt{V}\ e : \mathit{int},\Psi_2-\Psi_1}$$

A wait on a semaphore is typed as follows.

$$\frac{\Gamma,\Psi \vdash e : \mathit{sem}(\Psi_1),\Psi_2}{\Gamma,\Psi \vdash \texttt{P}\ e : \mathit{int},\Psi_2+\Psi_1}$$

Unlike a lock release, a post of a semaphore is not idempotent. For example, a race must be reported for the program in Figure 7. It is not difficult to prove that type system is sound for semaphores with their usual semantics of a post incrementing a counter and a wait waiting for a counter to be positive and then decrementing the counter.[3] We omit the details for space.

Read-Write Locks LP-Race models read-write locks when the upper bound on the number of live threads are known. Read-write locks are given types of the form $\mathit{rwlock}(\Psi_r,\Psi_w)$. Figure 8 show the type rules for read-write lock acquires and releases, which are much like those of regular locks, except that Ψ_w is used in write mode and Ψ_r is used in read-only mode.

Here, `newrwlock` creates a new read write lock, `rdlock` e (`rdunlock` e) acquires (releases) the read-write lock e in read-only mode, and `wrlock` e (`wrunlock` e) acquires (releases) the read-write lock e in write mode.

In the rule for `newrwlock`, N is the upper bound on the number of threads. For instance, if a read-write lock l is used to guard an abstract location ρ, then the type of l would be of the form $\mathit{rwlock}(\Psi_w,\Psi_r)$ such that $\Psi_w(\rho) \geq 1$. Obtaining a read lock grants $\Psi_r(\rho) \leq \Psi_w(\rho)/N$ amount of capability, which could be less than 1, but is enough to do a read (i.e., greater than 0). It is easy to prove that this scheme is sound when the program spawns at most N threads.

3.1.5 Local Accesses

Consider the program shown in Figure 9. The memory region allocated in the function `f` is used only inside the function (which include threads spawned by the function). To check that such local uses of memory regions are race free, LP-Race performs an escape analysis to determine if an abstract location escapes through globals or the function arguments or returns. Suppose ϱ does not escape. Then, LP-Race adds the constraint $\varphi_{pre}(\varrho) \leq 1$ where φ_{pre} is the capability at the entry of the function, and removes the constraints

[3] This actually implies that the type rule for `unlock` is somewhat conservative for double unlocking. However, unlocking an already unlocked lock is often considered a bug and has undefined behavior in many thread library specifications. Issues on recursive locking and unlocking is discussed further in Section 5.

```
int c;
pthread_t tid1, tid2;
sem_t sem;

void *f(void *) {
  sem_wait(&sem);
  c = 1;
}
void main(void) {
  sem_init(&sem, 0, 0);
  pthread_create(&tid1, NULL, &f, &c);
  pthread_create(&tid2, NULL, &f, &c);
  sem_post(&sem);
  sem_post(&sem);
}
```

Figure 7. Double semaphore post.

$$\frac{\Psi_r \leq \Psi_w/N}{\Gamma,\Psi \vdash \texttt{newrwlock} : \mathit{rwlock}(\Psi_r,\Psi_w),\Psi-\Psi_w}$$

$$\frac{\Gamma,\Psi \vdash e : \mathit{rwlock}(\Psi_r,\Psi_w),\Psi_1}{\Gamma,\Psi \vdash \texttt{rdlock}\ e : \mathit{int},\Psi_1+\Psi_r}$$

$$\frac{\Gamma,\Psi \vdash e : \mathit{rwlock}(\Psi_r,\Psi_w),\Psi_1}{\Gamma,\Psi \vdash \texttt{rdunlock}\ e : \mathit{int},\Psi_1-\Psi_r}$$

$$\frac{\Gamma,\Psi \vdash e : \mathit{rwlock}(\Psi_r,\Psi_w),\Psi_1}{\Gamma,\Psi \vdash \texttt{wrlock}\ e : \mathit{int},\Psi_1+\Psi_w}$$

$$\frac{\Gamma,\Psi \vdash e : \mathit{rwlock}(\Psi_r,\Psi_w),\Psi_1}{\Gamma,\Psi \vdash \texttt{wrunlock}\ e : \mathit{int},\Psi_1-\Psi_w}$$

Figure 8. Read-write lock type rules.

```
pthread_t tid1, tid2;
pthread_mutex_t lock;

void *g(void *q) {
  pthread_mutex_lock(&lock);
  *q = *q + 1;
  pthread_mutex_unlock(&lock);
}
void *f(void *) {
  int * p = malloc(sizeof(int));
  *p = 1;
  pthread_mutex_init(&lock, NULL);
  pthread_create(&tid1, NULL, &g, p);
  pthread_create(&tid2, NULL, &g, p);
}
void main(void) {
  pthread_create(&tid1, NULL, &f, NULL);
  pthread_create(&tid2, NULL, &f, NULL);
}
```

Figure 9. Local access example.

App	Size	LP Instances	Warnings	Time
aget	2.2	40	15	4
ctrace	2.2	24	12	5
smtprc	9.0	85	42	145
retawq	52.3	605	14	6855

Table 1. Experiment results.

of the form $\varphi(\varrho) \geq \varphi_{pre}(\varrho)$. This allows functions to use the locally allocated locations in a race free manner.

3.1.6 Subtyping

LP-Race takes advantage of the one-level-flow points-to analysis framework to do one level of subtyping (types are unified under pointers).

We show the subtyping rules for lock types. LP-Race extends the lock type to $\mathit{lock}(\Psi_{in}, \Psi_{out})$ such that Ψ_{in} is used at **LCK** and Ψ_{out} is used at **NEWL** and **ULCK**. We assert $\Psi_{out} \geq \Psi_{in}$, and use the following subtyping rule.

$$\frac{\Psi_{in} \geq \Psi'_{in} \quad \Psi_{out} \geq \Psi'_{out}}{\mathit{lock}(\Psi_{in}, \Psi_{out}) \leq \mathit{lock}(\Psi'_{in}, \Psi'_{out})}$$

The idea is inspired by the read-type write-type separation for subtyping reference cells. We type other synchronization primitives similarly. This has the effect of reducing false-aliasing of locks.

3.2 Experiments

LP-Race is implemented in OCaml. LP-Race uses CIL 1.3.6 [19] as the frontend parser, and GLPK 4.2.1 [1] as the backend linear programming solver. In general, any tool capable of solving a system of rational linear inequalities can be used as the backend. The code was compiled using OCaml 3.08 and gcc 3.4.4. The experiments were run on a PC with a Intel T7200 2GHZ processor with 2GB of RAM, running Cygwin inside Windows XP.

We ran LP-Race on several POSIX threads applications. We chose three benchmarks from the Locksmith paper [20], aget, ctrace and smtprc, mainly to check that the results from LP-Race agree with their findings, and also tried a larger application, retawq, a multithreaded webserver, to see how LP-Race scales to larger code base. Table 1 summarizes the results. The size column is the number of kilo lines of code after CIL merges the preprocessed application files and filters out duplicate or unused definitions. The time column is in seconds. The warnings column shows the number of possible races reported.

It is worth noting that the warning counts are sensitive to the underlying alias analysis. LP-Race currently distinguishes possible races by alias sets only (after removing duplicates), therefore, for instance, if we had used a very coarse alias analysis that returns a single alias set containing all locations, then the analysis would always report at most one warning. Also, with this method, even if there are multiple races on the same alias set, only one warning is reported. Section 5 discusses issues regarding error reporting.

We reviewed the error reports. For aget, LP-Race was able to detect the races reported in [20]. For ctrace, LP-Race detects the two races reported in [20]. In addition, it reports ten false positives. Examining these false positives, as pointed out in [20], some appeared to be due to semaphores. While LP-Race can handle standard semaphore uses, ctrace contains a manual read-write semaphore implemented with a counter, which LP-Race is not able to handle. Replacing this manual read-write semaphore with LP-Race's read-write lock eliminated four false positives. The other false positives are related to the unused lock problem discussed in Section 5, and accesses to a global array indexed by thread identifiers. We also observed that the handling of join was necessary for eliminating one false positive.

For smtprc, many of the warnings are false read-write races reported due to loops spawning unbounded number of threads (cf. Section 5). Manually unrolling the loops twice eliminates 19 false positives. The other false positives are due to accesses to a global data structure indexed by thread identifiers. We noticed that smtprc dangerously releases a lock in a loop, which, according to the POSIX threads specification, has undefined behavior. Such lock releases could lead to a race if a thread can release a lock that other threads are holding (though this is implementation dependent). LP-Race correctly reports warnings for such situations thanks to the conservative handling of recursive unlocks. For retawq, many of the warnings appear to be false positives caused by false aliasing of locations returned by memory allocation wrapper functions. One warning appears to be a race, though seemingly benign.

3.2.1 Discussion

The backend of LP-Race, which dominates the running time, is embarrassingly parallel. The backend solves one linear programming instance per an alias set containing a possibly thread shared location. The third column of Table 1 shows the number of instances created, after some filtering to remove redundant instances. Each linear programming instance is nearly the same size for the same code base and can be solved independently of the others. Therefore, by solving each instance in parallel, in theory, we should get near N times speedup with N parallel processors for applications with more than N alias sets.

Also, because the running times are dominated by the backend, the choice of the linear programming solver may affect the performance. We chose GLPK mostly out of convenience [4], but GLPK is by no means the fastest linear programming solver. Running times often differ by several orders of magnitude across different solvers [16]. We leave experimenting with other linear programming solvers for future work.

4. Related Work

Many static race analyses focus on lexically-scoped locking patterns such as the `synchronized` blocks in Java. The essence of lexically-scoped locking patterns can be cleanly captured as a

[4] We based the implementation on a partial OCaml interface for GLPK [15].

lockset-based analysis based on the classical type and effect system. Early systems [9, 10, 13, 4] required the users to supply annotations, whereas more recent work [2, 11, 18, 17] infer locksets automatically by utilizing powerful reasoning techniques such as binary decision diagrams and SAT solvers.

A lockset-based analysis for non-lexically scoped locks is said to require "flow-sensitivity" to infer locks held at each program point [20, 22], and considered more difficult than that for lexically scoped locks. Like lockset-based analyses for scoped locks, these analyses are usually limited to locks and lock-like synchronization patterns. An issue with handling non-lock synchronization patterns like semaphores and signals is that it is ok for another thread to "release" a semaphore that another thread has "acquired", while such a behavior is uncommon for locks.[5]

Also, unlike that for scoped locks that has a straightforward formalization as a type and effect system, a lockset-based analysis for non-scoped locks are rarely formalized and proven sound.[6] This paper gives a simple formalization of non-scoped locks (as well as other synchronization patterns) in terms of capabilities.

Locksmith [20] is a lockset-based analysis that introduces the notion of *correlation analysis* to reason about non-scoped locks used in a context sensitive manner. Relay [22] is a lockset-based analysis for non-scoped locks that has been applied to a much larger code base (millions of lines) than the applications we analyzed with LP-Race. To scale to such a large code base, they parallelize the analysis to utilize a cluster of high performance machines. As remarked in Section 3.2, LP-Race should also benefit from parallel computation. We leave implementing parallelized LP-Race for future work.

Both Relay and Locksmith employ a number of techniques to trade unlikely sources of unsoundness for precision or speed, such as optimistic thread-sharedness assumptions and using C types to refine aliasing. Such techniques are important for analyzing large-scale real-world programs. Many of such techniques affect only the "may alias" part of the analysis, and therefore, they should also be adaptable to our framework.

The technique to reduce a static analysis problem to linear programming may be of independent interest. The approach was inspired by the idea of fractional permissions/capabilities, originally proposed by Boyland [5] as a way to allow parallel reads while guaranteeing determinism. The idea has been used to reason about concurrent reads in separation logic [3], and also to check determinism of channel communicating processes [21]. This paper is the first application of the fractional permissions/capabilities idea to real world programs.

5. Open Issues

We identify the limitations of the current analysis system. We address each issue and describe possible remedies.

The analysis does not handle recursive locks because it assumes that a thread blocks when trying to acquire a lock that is already held, regardless of who holds the lock. Therefore. If this condition is violated, it is easy to construct a program that gains an invalid amount of capabilities (i.e., greater than 1) by acquiring the same lock multiple times. Effect-based approaches [9, 10, 13, 4] can naturally express Java-style lexically-scoped recursive locks. Effectively handling non-scoped recursive locks is an open issue [22].

One limitation that seems unique to our analysis (and possibly to others based on the permissions/capabilities idea) occurs when a program spawns an unbounded number of live threads in a loop. This implies that either the created threads start with no capability, or the loop head has an infinite amount of capabilities (which is invalid for any program locations that are reachable). In practice, this makes the analysis report some read-only access as a possible read-write race. For example, a false race is detected in the program below.

```
let x = ref 0 in
  while * do
    spawn(newtid){ !x }
```

Currently, we unroll thread allocating loops manually when LP-Race reports a false read-write race.

A related issue is locks created early. The key argument used to prove soundness is to interpret locks and other lock-like primitives to be "storing" the capabilities in their unlocked state (cf. Definition 2.3). Therefore, once a lock is created, the capabilities stored in the lock can prevent a thread from accessing the locations guarded by the lock even when there is no contention on the locations. In particular, this makes some thread-local accesses untypable, as shown below.

```
let x = ref 0 in
let l = newlock in
x := 1;
spawn(newtid){ lock l; x := 0; unlock l}
spawn(newtid){ lock l; x := 0; unlock l}
```

Here, `l` must hold the full (i.e., 1) capability for `x` so that the spawned threads can write to `x`. But this implies that `x := 1` is not typable because `l` is at an unlocked state at that point. A similar issue appears when accesses are made when the lock is no longer used (but before the lock is explicitly destroyed). Currently, we fix such situations by manually moving lock allocation/deletion points or by inserting a lock acquire/release pair around the unguarded access. A more principled remedy is to infer program points that can acquire a lock without contention, and allow such program points to use the capabilities stored in the lock without actually acquiring the lock.

Another issue with our approach is error reporting. Because the analysis does not compute locksets, LP-Race does not give a feedback containing held locks at each program location when it detects a possible race. This can make analyzing false positives somewhat inconvenient. Currently, LP-Race reports the program location and the kind (i.e., a read or a write) of accesses made to the abstract location that failed the check, and whether the error is a possible write-write race or is a read-write race. How to concisely represent the *reason* for the race (or a false positive) is an important issue in race analysis. One approach that may work well with LP-Race is an interactive debugging interface in which the user specifies a subset of reported accesses as race free so that LP-Race re-solves the linear programing instance for that location with the reduced accesses. With such a strategy, we can avoid re-running the entire analysis from scratch.

6. Conclusions

We have presented a new static analysis for race freedom that reduces the problem to linear programming. The analysis is quite different from more traditional analyses and does not require computation of locksets nor lock linearity/must-aliasness. The analysis has a straightforward formalization as a permissions/capabilities system, and enjoys benefits such as being able to handle a variety of synchronization primitives. The preliminary experiment reports encouraging results analyzing small to medium size multithreaded C programs.

[5] In fact, many threads libraries, including POSIX threads, condemns such idioms as erroneous.

[6] The Locksmith paper [20] formalizes by assuming that each access is annotated with the held locks.

References

[1] GNU Linear Programming Kit. http://www.gnu.org/software/glpk/glpk.html.

[2] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Proceedings*, pages 149–160.

[3] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–270, Long Beach, California, Jan. 2005.

[4] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[5] J. Boyland. Checking interference with fractional permissions. In *Static Analysis, Tenth International Symposium*, pages 55–72, San Diego, CA, June 2003.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver B.C., Canada, June 2000.

[7] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis, Eighth International Symposium*, pages 260–278, Paris, France, July 2001.

[8] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, Vancouver B.C., Canada, June 2000.

[9] C. Flanagan and M. Abadi. Types for safe locking. In D. Swierstra, editor, *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

[10] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver B.C., Canada, June 2000.

[11] C. Flanagan and S. N. Freund. Type inference against races. In *Static Analysis, Eleventh International Symposium*, pages 116–132, Verona, Italy, August 2004.

[12] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[13] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the 2003 ACM Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, Louisiana, Jan. 2003.

[14] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–218, Washington DC, USA, June 2004.

[15] S. Mimram. ocaml-glpk. http://ocaml-glpk.sourceforge.net/.

[16] H. Mittelmann. Benchmarks for optimization software. http://plato.asu.edu/bench.html.

[17] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, Nice, France, Jan. 2007.

[18] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, Ottawa, Ontario, Canada, June 2006.

[19] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction, 11th International Conference*, pages 213–228, Grenoble, France, Apr. 2002.

[20] P. Pratikakis, J. S. Foster, and M. W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, Ottawa, Ontario, Canada, June 2006.

[21] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. In *Concurrency Theory, 17th International Conference*, volume 4137, pages 218–232, Bonn, Germany, Aug. 2006.

[22] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214, Dubrovnik, Croatia, Sept. 2007.

[23] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, Washington DC, USA, June 2004.