LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection

Polyvios Pratikakis
University of Maryland, College Park
polyvios@cs.umd.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Abstract

One common technique for preventing data races in multi-threaded programs is to ensure that all accesses to shared locations are consistently protected by a lock. We present a tool called LOCKSMITH for detecting data races in C programs by looking for violations of this pattern. We call the relationship between locks and the locations they protect *consistent correlation*, and the core of our technique is a novel constraint-based analysis that infers consistent correlation context-sensitively, using the results to check that locations are properly guarded by locks. We present the core of our algorithm for a simple formal language $\lambda_\triangleright$ which we have proven sound, and discuss how we scale it up to an algorithm that aims to be sound for all of C. We develop several techniques to improve the precision and performance of the analysis, including a sharing analysis for inferring thread locality; existential quantification for modeling locks in data structures; and heuristics for modeling unsafe features of C such as type casts. When applied to several benchmarks, including multi-threaded servers and Linux device drivers, LOCKSMITH found several races while producing a modest number of false alarms.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Verification

Keywords context-sensitivity, correlation, race detection, type inference, multi-threaded programming, locksmith

1. Introduction

Data races occur in multi-threaded programs when one thread accesses a memory location at the same time another thread writes to it. While some races are benign, those that are erroneous can have disastrous consequences [31, 37]. Moreover, race-freedom is an important program property in its own right, because race-free programs are easier to understand, analyze, and transform [4, 42]. For example, race freedom is necessary for reasoning about code that uses locks to achieve atomicity [16, 20].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006…$5.00.

In this paper, we present a static analysis tool called LOCKSMITH for automatically finding all data races in a C program. Our analysis aims to be sound so that any potential races are reported, modulo the unsafe features of C such as arbitrary pointer arithmetic and type casting. We check for data races by enforcing one of the most common techniques for race prevention: We ensure that for every shared memory location ρ there is some lock ℓ that is held whenever ρ is accessed. While this technique is not the only way to prevent races, it is common in multi-threaded software.

Our goal is to produce a practical race-detection tool for C. A number of type systems have been developed for preventing races given specifications [5, 6, 13, 14, 22], but these can require considerable programmer annotations, limiting their practical application. Most completely-automatic static analyses have considered Java [2, 43, 17, 34, 28], thus avoiding many of the problematic features of C, such as type casts, low-level pointer operations, and non-lexically scoped locks. Those that consider C are either unsound, do not check certain idioms, or may have trouble scaling. A lengthy discussion of related work may be found in Section 5.

The core algorithm used by LOCKSMITH is an analysis that can automatically infer the relationship between locks and the locations they protect. We call this relationship *correlation*, and a key contribution of our approach is a new technique for inferring correlation context-sensitively. We present our correlation analysis algorithm for a formal language $\lambda_\triangleright$ that abstracts away some of the complications of operating directly on C code. Our analysis is constraint-based, using context-free language reachability [40, 41] and semi-unification [24] for context-sensitivity. Because each location must be consistently correlated with at least one lock, we use ideas from linear types to maintain a tight correspondence between abstract locks used by the static analysis and locks created at run time. We allow locks created in polymorphic functions to be treated distinctly at different call sites, and we use a novel type and effect system to ensure that this is safe.

In order to move from $\lambda_\triangleright$ to C, we use a number of additional techniques. One novel contribution is that we support *existential quantification*, which allows us to model correlations among fields of a data structure element, even after that element is merged into the "blob" typical of constraint-based alias analysis [9]. We use flow-sensitive analysis to model lock acquires and releases, which need not be lexically scoped. Our implementation includes a sharing analysis to model thread-local data, and uses heuristics to model type casts, including the special case of casts to and from `void *`. Finally, we use a lazy technique to efficiently model the large `struct` types typical of C programs.

We ran LOCKSMITH on a set of benchmarks, including programs that use POSIX threads and several Linux kernel device drivers. Our tool runs in seconds or minutes on our example programs, although for some other programs we have tried it does not complete due to resource exhaustion. LOCKSMITH found a num-

```
pthread_mutex_t L1 = ..., L2 = ...;
int x, y, z;

void munge(pthread_mutex_t *l, int *p) {
  pthread_mutex_lock(l);
  *p = 3;
  pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```

Figure 1. Locking Example in C

ber of data races, and overall produces few total warnings, making it easy to inspect the output manually. We also measured the effectiveness of the various analysis features mentioned above, and we found that all are useful, reducing the number of warnings in total by as much as a factor of three overall.

In summary, this paper makes the following contributions:

- We describe a context-sensitive correlation analysis for the language $\lambda_\triangleright$. Given a source program in $\lambda_\triangleright$, our analysis determines whether every memory location in the program is consistently correlated with a lock. Our analysis models locks linearly and uses a novel effect system to treat locks created in different calls to a function distinctly. (Section 2)
- We scale up our analysis to the C programming language with a series of additional techniques, including flow-sensitivity, existential quantification, and a sharing analysis to infer thread-local data. (Section 3)
- We evaluate our implementation on a small set of benchmarks. LOCKSMITH was able to find several races with few overall warning messages. (Section 4)

Although we focus on locking in this paper, we believe that the concept of correlation may be of independent interest. For example, a program may correlate a variable containing an integer length with a array having that length [49]; it may correlate an environment structure with the closure that takes it as an argument [32]; or it may correlate a memory location with the *region* in which that location is stored [23, 25].

2. Race Freedom as Consistent Correlation

Consider the C program in Figure 1. This program has two locks, `L1` and `L2`, and three integer variables, `x`, `y`, and `z` (we omit initialization code for simplicity). The function `munge` takes a lock and a pointer and writes through the pointer with the lock held. Suppose that the program makes the three calls to `munge` as shown, and that this sequence of calls is invoked by two separate threads.

This program is race-free because for each location, there is a lock that is always held when that location is accessed. In particular, `L1` is held for all accesses to `x`, and `L2` is held for all accesses to both `y` and `z`. More formally, we say that a location ρ is *correlated* with a lock ℓ if at some point a thread accesses ρ while holding ℓ. We say that a location ρ and a lock ℓ are *consistently correlated* if ℓ is *always* held by the thread accessing ρ. Thus if all locations in a program are consistently correlated, then that program is race free.

Establishing consistent correlation is a two-step process. First, we determine what locks ℓ are held when the thread accesses some location ρ. Having gathered this information, we can then ask whether ρ is consistently correlated with some lock.

To simplify our presentation, we present the core of our algorithm for a small language $\lambda_\triangleright$ in which locations can be guarded

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e_1\ e_2 \mid \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \mid & (e_1, e_2) \mid e.j \mid \texttt{let}\ f = v\ \texttt{in}\ e_2 \mid \texttt{fix}\, f.v \mid f^i \\
 & \mid & \texttt{newlock} \mid \texttt{ref}\ e \mid !^{e_2} e_1 \mid e_1 :=^{e_3} e_2 \\
v & ::= & n \mid \lambda x.e \mid (v_1, v_2)
\end{array}
$$

Figure 2. $\lambda_\triangleright$ Syntax

```
let L1 = newlock in
let L2 = newlock in
let x = ref 0 in
let y = ref 1 in
let z = ref 2 in

let munge l p =
  p :=^l 3 in
munge^1 L1 x;
munge^2 L2 y;
munge^3 L2 z
```

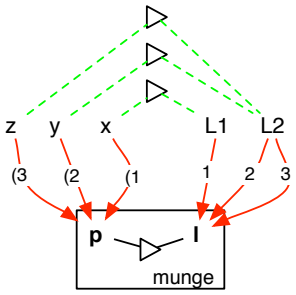


Figure 3. Locking example in $\lambda_\triangleright$ and its constraint graph

by at most one lock (rather than a set of locks), and in which the lock correlated with a memory read or write is made explicit in the program text. This allows us to defer the problem of determining what locks are held at each dereference and focus on checking for consistent correlation. In Section 3, we describe how to extend our ideas to find data races in the full C programming language, including how to infer held lock sets at each program point.

2.1 The Language $\lambda_\triangleright$

Figure 2 presents the syntax of $\lambda_\triangleright$, a polymorphic lambda calculus extended with integers, comparisons, pairs, a primitive for generating mutual exclusion locks, and updatable references. We annotate function occurrences f^i with an *instantiation site* i, as in some other context-sensitive analyses [40]. Dereferences $!^e\ e_1$ and assignments $e_1 :=^e e_2$ take as an additional argument an expression e that evaluates to a lock, which is acquired for the duration of the memory access and released afterward. To keep the presentation simpler $\lambda_\triangleright$ does not include other language features such as recursive data structures, although those are handled by LOCKSMITH. The left side of Figure 3 gives the program in Figure 1 modeled in $\lambda_\triangleright$. The body of `munge` has been reduced to the expression $\texttt{p} :=^{\texttt{l}} 3$, indicating that `l` will be held during the assignment to `p`.

To check whether this program is consistently correlated, a natural approach would be to perform a points-to analysis for all of the pointers and locks in the program. At the assignment $\texttt{p} :=^{\texttt{l}} 3$ in the program, we could correlate all of the locations ρ to which `p` may point with the *singleton* lock ℓ to which `l` points. The lock `l` must point to a single ℓ or else some location ρ might be accessed sometimes with one lock and sometimes with another. Unfortunately, this condition is not satisfied in our example: the points-to set of `l` is $\{\texttt{L1}, \texttt{L2}\}$, since it will be `L1` at the first call to `munge` and `L2` at the second call. Thus our hypothetical algorithm would erroneously conclude that no single lock is held for all accesses, leading to false reports of possible races.

The problem is that correlation between `l` and `p` is not being treated *context-sensitively*. Even if we were to use a context-sensitive alias analysis [9], the points-to sets mentioned above would be the same, assuming that within the body of the function we summarized all calls, which is a standard technique.

We address this problem in two steps. First, we introduce *correlation constraints* of the form $\rho \triangleright \ell$, which indicate that the location

$$
\begin{array}{lrcll}
\text{types} & \tau & ::= & \mathit{int} \mid \tau \times \tau \mid \tau \rightarrow^{\varepsilon} \tau' \mid \mathit{lock}\ \ell \mid \mathit{ref}^{\rho}\ \tau & \\
\text{labels} & l & ::= & \ell \mid \rho & \\
\text{effects} & \varepsilon & ::= & \emptyset \mid \{\ell\} \mid \chi \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon' & \\
\text{polytypes} & \sigma & ::= & (\forall.\tau, \vec{l}) & \\
\text{constr. sets} & C & ::= & \emptyset \mid \{c\} \mid C \cup C & \\
\text{constraints} & c & ::= & \tau \leq \tau' & \text{(subtyping)} \\
 & & \mid & \ell = \ell' & \text{(lock unification)} \\
 & & \mid & \rho \leq \rho' & \text{(location flow)} \\
 & & \mid & \rho \rhd \ell & \text{(correlation)} \\
 & & \mid & \varepsilon \leq \chi & \text{(effect flow)} \\
 & & \mid & \varepsilon \leq_{\vec{l}} \chi & \text{(effect filtering)} \\
 & & \mid & \mathit{effect}(\tau) = \emptyset & \text{(effect emptiness)} \\
 & & \mid & \tau \preceq^{i}_{p} \tau' & \text{(type instantiation)} \\
 & & \mid & \ell \preceq^{i} \ell' & \text{(lock instantiation)} \\
 & & \mid & \rho \preceq^{i}_{p} \rho' & \text{(location inst.)} \\
 & & \mid & \varepsilon \preceq^{i} \chi & \text{(effect inst.)}
\end{array}
$$

Figure 4. Types and Constraints

ρ is correlated with the lock ℓ. Here, ρ and ℓ are location and lock *labels*, used to represent locations and locks that arise at run time. Our analysis generates correlation constraints based on occurrences of $!^{e}\ e_1$ and $e_1 :=^{e}\ e_2$ in the program. Second, we formalize an analysis to propagate correlation constraints in a context-sensitive way throughout the program, by creating a variety of other (flow) constraints and solving them to determine whether correlations are consistent. We define consistent correlation precisely as follows.

DEFINITION 1 (Correlation Set). *Given a location ρ and a set of constraints C, we define the* correlation set *of ρ in C as*

$$S(C, \rho) = \{\ell \mid C \vdash \rho \rhd \ell\}$$

Here we write $C \vdash \rho \rhd \ell$ to say that $\rho \rhd \ell$ can be proven from the constraints in C.

DEFINITION 2 (Consistent Correlation). *A set of constraints C is* consistently correlated *if* $\forall \rho.\ |S(C, \rho)| \leq 1$.

Thus, a constraint set C is consistently correlated if all abstract locations ρ are either correlated with one lock, or are never accessed and so are correlated with no locks.

The right side of Figure 3 shows a graph of the constraints that our analysis generates for this example code. Each label in the program forms a node in the graph, and labeled, directed edges indicate data flow. Location flow edges corresponding to a function call are labeled with $(i$ for the parameters at call site i, and any return values (not shown) are labeled with $)i$. Locks are modeled with unification in our system, and we label such edges simply with the call site, with the direction of the arrow into the type that was instantiated. For example, both `L1` and `x` are passed in at call site 1, so they connect to the parameters using edges labeled with $(1$. Undirected edges represent correlation. In this case, the body of `munge` requires that `l` and `p` are correlated.

After generating constraints we perform constraint resolution to propagate correlation constraints context-sensitively through the call graph. In this example, we copy `munge`'s correlation constraint out to each of the call sites, resulting in the three correlation constraints shown with dashed edges:

$$x \rhd L1 \qquad y \rhd L2 \qquad z \rhd L2$$

It is easy to see that these constraints are consistently correlated according to Definition 2.

2.2 Type System

We use a type and effect system for generating constraints C to check for consistent correlation. Our type system proves judgments of the form $C; \Gamma \vdash e : \tau; \varepsilon$, which means that expression e has type τ and effect ε under type assumptions Γ and constraint set C.

Figure 4 gives the type language and constraints used by our analysis. Types include integers, pairs, function types annotated with an effect ε, lock types with a label ℓ, and reference types with a label ρ. Effects are used to enforce linearity for locks (see below), and consist of the empty effect $\emptyset$, a singleton effect $\{\ell\}$, effect variables χ which are solved for during resolution, and both disjoint and non-disjoint unions of effects $\varepsilon \uplus \varepsilon'$ and $\varepsilon \cup \varepsilon'$, respectively. $\lambda_{\rhd}$ models context-sensitivity over labels using polytypes σ, introduced by `let` and `fix`. In our type language, polytype $(\forall.\tau, \vec{l})$ represents a universally quantified type, where τ is the base type and $\vec{l}$ is the set of *non*-quantified labels [24, 40]. Finally, C is a set of atomic constraints c. Within the type rules, the judgment $C \vdash c$ indicates that c can be proven by the constraint set C; in our algorithm, such judgments cause us to "generate" constraint c and add it C.

Effects Effects ε form an important part of $\lambda_{\rhd}$'s type system by enforcing linearity for lock labels. Roughly speaking, a lock label ℓ is linear if it never represents two different run-time locks that could reside in the same storage or are simultaneously live. To understand why this is important, consider the following code, where hypothetical types and generated constraints are marked in comments, eliding the constraints for the references to locks. We use $e_1; e_2$ as the standard abbreviation for $(\lambda x.e_2)\ e_1$ where $x \notin fv(e_2)$.

```
let l = ref newlock in      // l : ref^ρ' (lock ℓ)
let x = ref 0 in            // x : ref^ρ int
  x :=^!l 1;                // ρ ▷ ℓ
  l := newlock;
  x :=^!l 2                 // ρ ▷ ℓ
```

This code violates consistent correlation because `x` is correlated with two different run-time locks due to the assignment. However, to give `l` a consistent type, ℓ is used to model both locks, violating linearity. As a result, the constraints mistakenly suggest the program is safe, because ρ is only ever correlated with ℓ.

We now turn to the monomorphic type rules for $\lambda_{\rhd}$, shown in Figure 5. The [Newlock] rule in this system requires that when we create a lock labeled ℓ we generate an effect $\{\ell\}$. The other rules, like [Pair], join the effects of their subexpressions with disjoint union $\uplus$, thus requiring that chosen lock labels not conflict. For example, with the given labeling, the above code has the effect $\{\ell\} \uplus \{\ell\}$. We implicitly require that disjoint unions are truly disjoint—during constraint resolution, we will check that this holds—and thus we would forbid `L1` and `L2` from being given the same label. On the other hand, location labels ρ, introduced in the rule [Ref] for typing memory allocation, do not add to the effect as memory locations need not be linear.

Some other type-based systems for race detection [13, 22] and related systems for modeling dynamic memory allocation [46] avoid the need for this kind of effect by forcing newly-allocated locks (and/or locations) to be valid only within a lexical scope. That is, `newlock` is replaced with a construct `newlock` x `in` e, which at run time generates a new lock and substitutes it for x within e. When typing this construct, x's label ℓ is only valid in the expression e, ensuring the allocated lock cannot escape. Therefore subsequent invocations of the same `newlock` x `in` e (e.g., within a recursive function) cannot be confused. We can achieve the same

$$[\text{Id}]\ \frac{}{C;\Gamma, x:\tau \vdash x:\tau;\emptyset}$$

$$[\text{Int}]\ \frac{}{C;\Gamma \vdash n:\mathit{int};\emptyset}$$

$$[\text{Lam}]\ \frac{C;\Gamma, x:\tau \vdash e:\tau';\varepsilon \quad C \vdash \varepsilon \le \chi \quad \chi \text{ fresh}}{C;\Gamma \vdash \lambda x.e : \tau \to^{\chi} \tau';\emptyset}$$

$$[\text{App}]\ \frac{\begin{array}{c}C;\Gamma \vdash e_1 : \tau \to^{\varepsilon} \tau';\varepsilon_1 \\ C;\Gamma \vdash e_2 : \tau;\varepsilon_2\end{array}}{C;\Gamma \vdash e_1\ e_2 : \tau';\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

$$[\text{Pair}]\ \frac{C;\Gamma \vdash e_1:\tau_1;\varepsilon_1 \quad C;\Gamma \vdash e_2:\tau_2;\varepsilon_2}{C;\Gamma \vdash (e_1,e_2):\tau_1 \times \tau_2;\varepsilon_1 \uplus \varepsilon_2}$$

$$[\text{Proj}]\ \frac{C;\Gamma \vdash e:\tau_1 \times \tau_2;\varepsilon \quad j=1,2}{C;\Gamma \vdash e.j:\tau_j;\varepsilon}$$

$$[\text{Sub}]\ \frac{C;\Gamma \vdash e:\tau_1;\varepsilon \quad C \vdash \tau_1 \le \tau_2}{C;\Gamma \vdash e:\tau_2;\varepsilon}$$

$$[\text{Cond}]\ \frac{\begin{array}{c}C;\Gamma \vdash e_0 : \mathit{int};\varepsilon_0 \\ C;\Gamma \vdash e_1:\tau;\varepsilon_1 \quad C;\Gamma \vdash e_2:\tau;\varepsilon_2\end{array}}{C;\Gamma \vdash \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau;\varepsilon_0 \uplus (\varepsilon_1 \cup \varepsilon_2)}$$

$$[\text{Newlock}]\ \frac{\ell \text{ fresh}}{C;\Gamma \vdash \texttt{newlock} : \mathit{lock}\ \ell;\{\ell\}}$$

$$[\text{Ref}]\ \frac{C;\Gamma \vdash e:\tau;\varepsilon \quad \rho \text{ fresh}}{C;\Gamma \vdash \texttt{ref}\ e : \mathit{ref}^{\rho}\ \tau;\varepsilon}$$

$$[\text{Deref}]\ \frac{\begin{array}{c}C;\Gamma \vdash e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 \quad C;\Gamma \vdash e_2 : \mathit{lock}\ \ell;\varepsilon_2 \\ C \vdash \rho \rhd \ell\end{array}}{C;\Gamma \vdash !^{e_2} e_1 : \tau;\varepsilon_1 \uplus \varepsilon_2}$$

$$[\text{Assign}]\ \frac{\begin{array}{cc}C;\Gamma \vdash e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 & C;\Gamma \vdash e_2:\tau;\varepsilon_2 \\ C;\Gamma \vdash e_3 : \mathit{lock}\ \ell;\varepsilon_3 & C \vdash \rho \rhd \ell\end{array}}{C;\Gamma \vdash e_1 :=^{e_3} e_2 : \tau;\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_3}$$

Figure 5. $\lambda_{\triangleright}$ Monomorphic Rules

effect using the [Down] rule, described below, and our approach matches the usage of `newlock` as it occurs in practice.

Typing Rules Turning to the remaining rules in Figure 5, [Id], [Int], and [Proj] are standard. [Lam] types a function definition, and the effect on the function arrow is the effect of the body. Notice that we always place effect variables χ on function arrows; this ensures constraints involving effects always have a variable on their right-hand side, simplifying constraint resolution. In [App] we apply a function e_1 to argument e_2, and the effect includes the effect of evaluating e_1, the effect of evaluating e_2, and the effect of the function body.

The [Sub] rule and subtyping rules, shown in Figure 7(a), are also standard. Note that in rule [Sub-Lock], we require ℓ and ℓ' to be equal. Thus we have no subtyping on lock labels, which makes it easier to enforce linearity by forcing lock labels that "flow" together to be unified. The rules in Figure 7(a) can be seen as judgments for reducing subtyping on types to constraints on labels, and during constraint resolution we assume that all subtyping constraints have been reduced in this way and thus eliminated.

[Cond] is mostly standard, except we use a non-disjoint union to join the effects of the two branches, since only one of e_1 or e_2 will be executed at run time. [Deref] accesses a location e_1 while holding lock e_2, and generates a correlation constraint between the lock and location label, as does [Assign].

$$[\text{Let}]\ \frac{\begin{array}{c}C;\Gamma \vdash v_1:\tau_1;\emptyset \quad \vec{l} = \mathit{fl}(\Gamma) \\ C;\Gamma, f:(\forall.\tau_1,\vec{l}) \vdash e_2:\tau_2;\varepsilon\end{array}}{C;\Gamma \vdash \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2;\varepsilon}$$

$$[\text{Inst}]\ \frac{C \vdash \tau \preceq^{i}_{+} \tau' \quad C \vdash \vec{l} \preceq^{i}_{\pm} \vec{l}}{C;\Gamma, f:(\forall.\tau,\vec{l}) \vdash f^{i} : \tau';\emptyset}$$

$$[\text{Fix}]\ \frac{\begin{array}{cc}C;\Gamma, f:(\forall.\tau,\vec{l}) \vdash v:\tau';\emptyset & \vec{l} = \mathit{fl}(\Gamma) \\ C \vdash \tau' \le \tau & C \vdash \tau \preceq^{i}_{+} \tau'' \\ C \vdash \vec{l} \preceq^{i}_{\pm} \vec{l} & C \vdash \mathit{effect}(\tau) = \emptyset\end{array}}{C;\Gamma \vdash \texttt{fix}\ f.v : \tau'';\emptyset}$$

$$[\text{Down}]\ \frac{\begin{array}{cc}C;\Gamma \vdash e:\tau;\varepsilon & \vec{l} = \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau) \\ C \vdash \varepsilon \le_{\vec{l}} \chi & \chi \text{ fresh}\end{array}}{C;\Gamma \vdash e:\tau;\chi}$$

Figure 6. $\lambda_{\triangleright}$ Polymorphic Rules (plus [Down])

Polymorphism Figure 6 gives the rules for polymorphism. [Let] introduces polytypes. As is standard we only generalize the types of values. In [Let] the name f is bound to a quantified type where $\vec{l}$ is the set of free labels of Γ, i.e., the labels that cannot be generalized.

In [Inst], we use *instantiation constraints* to model a type instantiation. The constraint $\tau \preceq^{i}_{+} \tau'$ means that there exists some substitution ϕ_i such that $\phi_i(\tau) = \tau'$, i.e., that at the use of f labeled by index i in the program, τ is instantiated to τ'. We also generate the constraint $\vec{l} \preceq^{i}_{\pm} \vec{l}$, which requires that all of the variables we could not quantify are renamed to themselves by ϕ_i, i.e., they are not instantiated.

The subscript $+$'s and $-$'s in an instantiation constraint are *polarities*, which represent the direction of subtyping through a constraint, either covariant ($+$) or contravariant ($-$). Instantiation constraints correspond to the edges labeled with parentheses in Figure 3. A constraint $\rho \preceq^{i}_{+} \rho'$ corresponds to an output (i.e., a return value), and in constraint graphs we draw it as a directed edge $\rho \to^{)i} \rho'$. A constraint $\rho \preceq^{i}_{-} \rho'$ corresponds to an input (i.e., a parameter), and we draw it with a directed edge $\rho' \to^{(i} \rho$. We draw a constraint $\ell \preceq^{i} \ell'$ as an edge $\ell' \to^{i} \ell$, where there is no direction of flow since lock labels are unified but the arrow indicates the reverse direction of instantiation.

Instantiation constraints on types can be reduced to instantiation constraints on labels, as shown in Figure 7(b). In these rules we use p to stand for an arbitrary polarity, and in [Inst-Fun] we flip the direction of polarity for the function domain with the notation $\bar{p}$. For example, to generate the graph in Figure 3, we generated three instantiation constraints

$$\begin{array}{l}(\texttt{1} \times \texttt{p}) \to \mathit{int} \ \preceq^{1}_{+}\ (\texttt{L1} \times \texttt{x}) \to \mathit{int} \\ (\texttt{1} \times \texttt{p}) \to \mathit{int} \ \preceq^{2}_{+}\ (\texttt{L2} \times \texttt{y}) \to \mathit{int} \\ (\texttt{1} \times \texttt{p}) \to \mathit{int} \ \preceq^{3}_{+}\ (\texttt{L2} \times \texttt{z}) \to \mathit{int}\end{array}$$

corresponding to the three instantiations and calls of `munge`. For full details on polarities, see Rehof et al [40].

Hiding Effects [Fix] introduces polymorphic recursion, which is decidable for label flow [33, 40]. However, in our system we instantiate effects, which because they contain disjoint unions may grow without bound if a recursive function allocates a lock. Thus in [Fix], we require that recursive functions have an empty effect on their top-most arrow with the constraint $\mathit{effect}(\tau) = \emptyset$.

$$\text{[Sub-Int]}\ \frac{}{C \vdash \mathit{int} \leq \mathit{int}}$$

$$\text{[Sub-Pair]}\ \frac{C \vdash \tau_1 \leq \tau_1' \quad C \vdash \tau_2 \leq \tau_2'}{C \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$\text{[Sub-Lock]}\ \frac{C \vdash \ell = \ell'}{C \vdash \mathit{lock}\ \ell \leq \mathit{lock}\ \ell'}$$

$$\text{[Sub-Ref]}\ \frac{C \vdash \rho \leq \rho' \quad C \vdash \tau \leq \tau' \quad C \vdash \tau' \leq \tau}{C \vdash \mathit{ref}^{\rho}\ \tau \leq \mathit{ref}^{\rho'}\ \tau'}$$

$$\text{[Sub-Fun]}\ \frac{C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \quad C \vdash \varepsilon_1 \leq \varepsilon_2}{C \vdash \tau_1 \to^{\varepsilon_1} \tau_1' \leq \tau_2 \to^{\varepsilon_2} \tau_2'}$$

(a) Subtyping

$$\text{[Inst-Int]}\ \frac{}{C \vdash \mathit{int} \preceq^i \mathit{int}}$$

$$\text{[Inst-Pair]}\ \frac{C \vdash \tau_1 \preceq_p^i \tau_1' \quad C \vdash \tau_2 \preceq_p^i \tau_2'}{C \vdash \tau_1 \times \tau_2 \preceq_p^i \tau_1' \times \tau_2'}$$

$$\text{[Inst-Lock]}\ \frac{C \vdash \ell \preceq^i \ell'}{C \vdash \mathit{lock}\ \ell \preceq_p^i \mathit{lock}\ \ell'}$$

$$\text{[Inst-Ref]}\ \frac{C \vdash \rho \preceq_p^i \rho' \quad C \vdash \tau \preceq_{\pm}^i \tau'}{C \vdash \mathit{ref}^{\rho}\ \tau \preceq_p^i \mathit{ref}^{\rho'}\ \tau'}$$

$$\text{[Inst-Fun]}\ \frac{C \vdash \tau_1 \preceq_p^i \tau_2 \quad C \vdash \tau_1' \preceq_p^i \tau_2' \quad C \vdash \varepsilon_1 \preceq^i \varepsilon_2}{C \vdash \tau_1 \to^{\varepsilon_1} \tau_1' \preceq_p^i \tau_2 \to^{\varepsilon_2} \tau_2'}$$

(b) Instantiation

Figure 7. Subtyping and Instantiation Constraints

This is a strong restriction, and we would like to be able infer correlations for recursive functions that allocate locks. For example, consider the following two code snippets:

```
fix f.λx.                      let y = ref 0 in
  let l = newlock in           fix f.λx.
  let y = ref 0 in               let l = newlock in
    y :=^l 42;                     y :=^l 42;
    ... f 0 ...                    ... f 0 ...
```

Here f is a recursive function that creates a lock `l` and accesses a location `y`. In both cases the lock does not escape the function, and therefore the linear labels corresponding to the locks in different iterations of the function cannot interfere. However, in the second case the location `y` is allocated outside the function, meaning that with each iteration it will be accessed with a different lock held, violating consistent correlation. We want to allow the first case while rejecting the second.

Thus we add a final rule [Down] to our type system to hide effects on lock labels that are purely local to a block of code [21]. In [Down], we generate a "filtering" constraint $\varepsilon \leq_{\vec{l}} \chi$, which means that χ should contain labels in ε that *escape* through $\vec{l}$, but not necessarily any other label. We determine escaping during constraint resolution. Formally, $C \vdash \mathit{escapes}(l, \vec{l})$, where l is either a ρ or ℓ, if

$$l \in \vec{l} \ \vee\ \exists c, l'.\ \Big(C \vdash c\ \wedge\ l, l' \in c\ \wedge\ C \vdash \mathit{escapes}(l', \vec{l})\Big)$$

$$\begin{array}{rcl}
C \cup \{\ell = \ell'\} & \Rightarrow & C[\ell \mapsto \ell'] \\
C \cup \{\rho_0 \leq \rho_1\} \cup \{\rho_1 \leq \rho_2\} & \cup \Rightarrow & \{\rho_0 \leq \rho_2\} \\
C \cup \{\ell_0 \preceq^i \ell_1\} \cup \{\ell_0 \preceq^i \ell_2\} & \Rightarrow & C[\ell_2 \mapsto \ell_1] \cup \{\ell_0 \preceq^i \ell_1\} \\
C \cup \{\rho_1 \preceq_-^i \rho_0\} \cup \{\rho_1 \leq \rho_2\} \cup \{\rho_2 \preceq_+^i \rho_3\} & \cup \Rightarrow & \{\rho_0 \leq \rho_3\}
\end{array}$$

(a) Flow of lock and location labels

$$\begin{array}{rcl}
C \cup \{\rho \leq \rho'\} \cup \{\rho' \rhd \ell\} & \cup \Rightarrow & \{\rho \rhd \ell\} \\
C \cup \{\rho \preceq_p^i \rho'\} \cup \{\rho \rhd \ell\} \cup \{\ell \preceq^i \ell'\} & \cup \Rightarrow & \{\rho' \rhd \ell'\}
\end{array}$$

(b) Correlation propagation

$$\begin{array}{rcl}
C \cup \{\emptyset \leq \chi\} & \Rightarrow & C \\
C \cup \{\varepsilon \cup \varepsilon' \leq \chi\} & \Rightarrow & C \cup \{\varepsilon \leq \chi\} \cup \{\varepsilon' \leq \chi\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq \chi'\} & \cup \Rightarrow & \{\varepsilon \leq \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq_{\vec{l}} \chi'\} & \cup \Rightarrow & \{\varepsilon \leq_{\vec{l}} \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \preceq^i \chi'\} & \cup \Rightarrow & \{\varepsilon \preceq^i \chi'\} \\
\hline
C \cup \{\emptyset \preceq^i \chi\} & \Rightarrow & C \\
C \cup \{\{\ell\} \preceq^i \chi\} & \Rightarrow & C \cup \{\ell \preceq^i \ell'\} \cup \{\{\ell'\} \leq \chi\} \\
 & & \quad \ell' \text{ fresh} \\
C \cup \{\varepsilon \uplus \varepsilon' \preceq^i \chi_0\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi'\} \\
 & & \quad \cup \{\chi \uplus \chi' \leq \chi_0\} \\
 & & \quad \chi, \chi' \text{ fresh} \\
C \cup \{\varepsilon \cup \varepsilon' \preceq^i \chi\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi\} \\
C \cup \{\chi \preceq^i \chi'\} \cup \{\chi \preceq^i \chi''\} & \Rightarrow & C[\chi' \mapsto \chi''] \cup \{\chi \preceq^i \chi''\} \\
\hline
C \cup \{\emptyset \leq_{\vec{l}} \chi\} & \Rightarrow & C \\
C \cup \{\{\ell\} \leq_{\vec{l}} \chi\} & \Rightarrow & C \cup \{\{\ell\} \leq \chi\} \\
 & & \quad \text{if } C \vdash \mathit{escapes}(\ell, \vec{l}) \\
C \cup \{\varepsilon \uplus \varepsilon' \leq_{\vec{l}} \chi_0\} & \Rightarrow & C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi'\} \\
 & & \quad \cup \{\chi \uplus \chi' \leq \chi_0\} \\
C \cup \{\varepsilon \cup \varepsilon' \leq_{\vec{l}} \chi\} & \Rightarrow & C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi\}
\end{array}$$

(c) Effect propagation

Figure 8. Constraint Resolution

In other words, l escapes through $\vec{l}$ if it is in $\vec{l}$ or if it appears in a constraint in C with an l' that escapes in $\vec{l}$. For example, if $\rho \rhd \ell$ and ρ escapes, then ℓ escapes. This prevents `l` from being hidden in our second example above, while in the first example we can apply [Down] to hide the allocation effect successfully. Although [Down] is not a syntax-directed rule, it is only useful to apply it to terms whose effect may be duplicated in the type system. Hence we can make the system syntax-directed by assuming that [Down] is always applied once to e in rule [Lam], so that the effect on the function arrow has as much hidden as possible. Also note that we can easily encode the lexically-scoped lock allocation primitive `newlock` x `in` e as $(\lambda x.e)$ `newlock` and applying [Down] to the application.

Uses of [Down] are rare in C programs in our experience, which tend to use global locks. Some C programs also store locks in data structures, and in this case [Down] allows us to hide locks that are created and then packed inside of an existential type (Section 3.3) that contains the only reference to them.

2.3 Constraint Resolution

After we have applied the rules in Figures 5, 6, and 7 to a $\lambda_{\rhd}$ program, we are left with a set of constraints C. To check that a program is consistently correlated, we first reduce the constraints C into a *solved form*, from which we can easily extract correlations between locks and locations.

Figure 8 gives a series of left-to-right rewrite rules that we apply exhaustively to the constraints to compute their solution. Figure 8(a) gives rules to compute the "flow" of locations and

locks; part (b) gives the rules for propagating correlations; and part (c) propagates effects so that we can check that disjoint unions are truly disjoint. The rules in part (a) are mostly standard, while parts (b) and (c) are new. Here, $C \cup \Rightarrow C'$ means $C \Rightarrow C \cup C'$.

The first rule of part (a) resolves equality constraints on lock labels and the second transitively closes subtyping constraints on location labels. The next rule is the standard semi-unification rule [24]: If a lock label ℓ_0 is instantiated at site i to two different lock labels ℓ_1 and ℓ_2, then ℓ_1 and ℓ_2 must be equal (because the substitution at site i has to substitute for ℓ_0 consistently). The final rule is for "matched flow." Recall the [Inst] rule from Figure 6: if f has polytype $(\forall.\, \mathit{ref}^{\rho_1}\ \tau_1 \rightarrow^{\emptyset} \mathit{ref}^{\rho_2}\ \tau_1, \emptyset)$, then instantiating this polytype at site i to the type $\mathit{ref}^{\rho_0}\ \tau_1 \rightarrow^{\emptyset} \mathit{ref}^{\rho_3}\ \tau_1$ requires that C contain instantiation constraints $\rho_1 \preceq^i_- \rho_0$ and $\rho_2 \preceq^i_+ \rho_3$ (according to [Inst-Fun] and [Inst-Ref]). The negative constraint corresponds to context-sensitive flow from the caller's argument to the function's parameter while the positive constraint corresponds to the returned value. Say that f is the identity function; then C would contain the constraint $\rho_1 \leq \rho_2$, indicating the function's parameter flows to its returned value. Thus the argument at site i should flow to the value returned at site i, and so the matched flow rule permits the addition of a flow edge $\rho_0 \leq \rho_3$. For a full discussion of this rule, see Rehof et al [40].

In the correlation propagation rules in part (b), the first rule says that if location ρ flows to a location ρ' that is correlated with ℓ, then ρ is correlated with ℓ also. Notice that there is no similar rule for flow on the right-hand side of a correlation, because we unify lock labels. The next rule propagates correlations at instantiation sites. Similarly to location propagation, if we have a correlation constraint $\rho \rhd \ell$ on the labels in a polymorphic function, and we instantiate ℓ to ℓ' and ρ to ρ' at some site i, then we propagate the correlation to ℓ' and ρ'. For example, Figure 3 depicts the following three constraints, among others (recall an edge $l' \rightarrow^{(i}\ l$ in the figure corresponds to a constraint $l \preceq^i_- l'$):

$$\texttt{l} \preceq^1_- \texttt{L1} \qquad \texttt{p} \preceq^1_- \texttt{x} \qquad \texttt{p} \rhd \texttt{l}$$

Using our resolution rule yields the constraint $\texttt{x} \rhd \texttt{L1}$, shown in Figure 3 with a dashed line. Note that the polarity of the instantiation constraint on ρ is irrelevant for this propagation step, because locks can correlate with both inputs (parameters) and output (returns).

Part (c), presented as three blocks of rules, propagates effect constraints. The first block of rules discards useless effect subtyping, replaces standard unions by two separate constraints, and computes transitivity of subtyping on effects. The next block of rules handles instantiation constraints. The constraint $\emptyset \preceq^i \chi$ can be discarded, because it places no constraint on χ. (It is not even the case that χ must be empty, because it may have subtyping constraints on it from other effects.) In the next rule we model instantiation of a function with a single effect $\{\ell\}$. In our system, each time we call a function that invokes `newlock` we wish to treat the locks from different calls differently. Thus we create a fresh lock label ℓ' that flows to χ and require that ℓ is instantiated to ℓ'. The remaining rules copy disjoint unions across an instantiation site, expand non-disjoint unions, and require that effect variables are instantiated consistently.

The last block of rules propagates effects across filtering constraints. The only interesting rule is the second one, which propagates an effect $\{\ell\}$ to χ only if ℓ escapes in the set $\vec{l}$; this corresponds to "hiding" effects χ that are only used within a lexical scope.

After applying the rewrite rules, there are three conditions we need to check. First, we need to ensure that all disjoint unions formed during type inference and constraint resolution are truly disjoint. We define $\mathit{occurs}(\ell, \varepsilon)$ to be the number of times label ℓ occurs disjointly in ε:

$$\begin{array}{rcl}
\mathit{occurs}(\ell, \emptyset) & = & 0 \\
\mathit{occurs}(\ell, \chi) & = & \max_{\varepsilon \leq \chi} \mathit{occurs}(\ell, \varepsilon) \\
\mathit{occurs}(\ell, \{\ell\}) & = & 1 \\
\mathit{occurs}(\ell, \{\ell'\}) & = & 0 \qquad \ell \neq \ell' \\
\mathit{occurs}(\ell, \varepsilon \uplus \varepsilon') & = & \mathit{occurs}(\ell, \varepsilon) + \mathit{occurs}(\ell, \varepsilon') \\
\mathit{occurs}(\ell, \varepsilon \cup \varepsilon') & = & \max(\mathit{occurs}(\ell, \varepsilon), \mathit{occurs}(\ell, \varepsilon'))
\end{array}$$

We require for every effect ε created during type inference (including constraint resolution), and for all ℓ, that $\mathit{occurs}(\ell, \varepsilon) \leq 1$. We enforce the constraint $\mathit{effect}(\tau) = \emptyset$ by extracting the effect ε from the function type τ and ensuring that $\mathit{occurs}(\ell, \varepsilon) = 0$ for all ℓ.

Finally, we ensure that locations are consistently correlated with locks. We compute $S(C, \rho)$ for all locations ρ and check that it has size ≤ 1. This computation is easy with the constraints in solved form; we simply walk through all the correlation constraints generated in Figure 8(b) to count how many different lock labels appear correlated with each location ρ.

We now analyze the running time of our algorithm for each part of constraint resolution. Let n be the number of constraints generated by walking over the source code of the program. Then the rules in Figure 8(a) take time $O(n^3)$ [40], as do the rules in Figure 8(b), since given n constraints there can be only $O(n^2)$ correlations among locations and locks mentioned in the constraints. Constraint resolution rules like those given in parts (a) and (b) have been shown to be efficient in practice [9].

There exist constraint sets C for which the rules in Figure 8(c) will not terminate. This is because a cycle in the instantiation constraints might result in a single effect being repeatedly copied and renamed. We believe that this cannot occur in our type system, however, because we forbid recursive functions from having effects. Even so, effect propagation can still be $O(2^n)$, because a single effect might be copied through a chain of instantiations that double the effect each time.

Soundness We have proven that a version of our type system $\lambda^{cp}_{\rhd}$ based on polymorphically constrained types [33] is sound, and that the system presented here reduces to that system. We define a call-by-value operational semantics as a series of rewriting rules, using evaluation contexts $\mathbb{E}$ to define evaluation order, as is standard. The evaluation rule for `newlock` generates a fresh lock constant L, and `ref` v generates a fresh location constant R. We extend labels l to include L and R and define typing rules for them. We also introduce *allocation constraints* $L \leq^1 \ell$ to indicate that lock variable ℓ has been allocated as constant L. We then refine $S(C, \rho)$ to $S_g(C, \rho)$, which only refers to concrete lock labels:

$$S_g(C, \rho) = \{L \mid C \vdash \rho \rhd \ell \wedge C \vdash L \leq^1 \ell\}$$

Thus $S_g(C, \rho)$ is the set of concrete locks correlated with ρ in C.

Next we define valid evaluation steps, which are those such that if a location R is accessed with lock L, then $L \in S_g(C, R)$.

DEFINITION 3 (Valid Evaluation). *We write $C \vdash e \longrightarrow e'$ iff $e \equiv \mathbb{E}[!^{[L]}\, v^R]$ or $e \equiv \mathbb{E}[v'^R :=^{[L]} v]$ implies $L \in S_g(C, R)$.*

Notice that this still allows a location to be correlated with more than one lock. We define an auxiliary judgment $\varepsilon \vdash_{ok} C$, which holds if in C all locations are consistently correlated and no lock labels in ε have been allocated.

We write $\vdash_{cp}$ for the type judgment in $\lambda^{cp}_{\rhd}$. We then show preservation, which implies soundness.

LEMMA 1 (Preservation). *If $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$ where $\varepsilon \vdash_{ok} C$ and $e \longrightarrow e'$, then there exists some C', ε', such that $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$; and $C' \vdash C$; and $C' \vdash e \longrightarrow e'$; and $\varepsilon' \vdash_{ok} C'$; and $C'; \Gamma \vdash_{cp} e' : \tau; \varepsilon'$.*

(The proof is by induction on $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$.) This lemma shows that if we begin with a consistently correlated constraint system and take a step for an expression e whose effect is ε, then the evaluation is valid. Moreover, there is some consistently correlated C' that entails C, where C' may contain additional constraints if the evaluation step allocated any locks or locations. Notice that since C' entails C, any correlations that hold in C also hold in C'. Since at each evaluation step we preserve existing correlations and maintain consistent correlation, a well-typed program is always consistently correlated during evaluation.

Finally, we can prove that we can reduce judgments in $\lambda_{\triangleright}$ to $\lambda_{\triangleright}^{cp}$. This reduction-based proof technique follows Fähndrich et al [12].

LEMMA 2 (Reduction). *Given a derivation of* $C; \Gamma \vdash e : \tau; \varepsilon$, *then* $C^*; \Gamma^* \vdash_{cp} e : \tau; \varepsilon^*$.

where C^* is the set of constraints closed according to the rules in Figure 8(a) and (b), ε^* is the set of locks in ε according to the rules in Figure 8(c), and Γ^* is a translation from $\lambda_{\triangleright}$ to $\lambda_{\triangleright}^{cp}$ type assumptions.

Full proofs can be found in a forthcoming technical report.

3. LOCKSMITH: Race Detection for C

LOCKSMITH applies the ideas of Section 2 to the full C programming language. We implemented LOCKSMITH using CIL [35] as a C front-end and using BANSHEE [30] to encode portions of the constraint graph and to apply the resolution rules in Figure 8(a). We use our own constraint solver for the rest of the analysis.

LOCKSMITH is structured as a set of modules implementing different phases of the analysis. The first phase traverses source code and generates constraints akin to $\lambda_{\triangleright}$ constraints. However, while $\lambda_{\triangleright}$ programs specify a correlation between a lock and a location explicitly, in C such correlations must be inferred. Using some additional constraint forms, LOCKSMITH infers which locks are held at each program point, and generates correlations accordingly to detect potential races. As an optimization, LOCKSMITH includes a middle phase to compute which locations are always thread-local and therefore can be ignored for purposes of checking correlation. LOCKSMITH also includes two additional features to improve precision for C. We support existential types, to model locks stored in data structures, and we try to model pointers to `void` precisely and structures efficiently.

3.1 Flow-Sensitive Race Detection

LOCKSMITH extends $\lambda_{\triangleright}$ type judgments to include *state variables* ψ [21] to model the flow-sensitive events needed to infer correlations. Judgments include both an input and an output state variable, representing the point just before and just after, respectively, execution of the expression. Function types also have an input and output ψ, to represent the initial and final states of the function. Control flow from state ψ to state ψ' is indicated by a *control flow constraint* $\psi \leq \psi'$, and we also include instantiation on states, written $\psi \preceq_p^i \psi'$. Each state is assigned a *kind* that describes how that state differs from preceding states.

As an example, the typing rule for acquiring a lock is

$$[\text{Acquire}]\frac{\psi; C; \Gamma \vdash e : \mathit{lock}\ \ell; \psi'; \varepsilon \qquad \psi'' \text{ fresh} \quad C \vdash \psi' \leq \psi'' \qquad C \vdash \psi'' : \texttt{Acquire}(\ell)}{\psi; C; \Gamma \vdash \texttt{acquire}\ e : \mathit{int}; \psi''; \varepsilon}$$

This rule says that to infer a type for `acquire` e beginning in state ψ, we infer a labeled lock type for e, whose evaluation produces the state ψ'. We create a new state ψ'' that immediately follows ψ' and in which ℓ is acquired. A similar rule for `release` e annotates states with kind $\texttt{Release}(\ell)$, and a rule for dereferences and assignments annotates states with kind $\texttt{Deref}(\rho)$ for reads of writes to location ρ. An example of control-flow constraints is shown below in Section 3.3.

Computing Held Locks Given a control-flow constraint graph, LOCKSMITH computes the locks held at each program point represented by a state ψ. Assume for the moment that all locks are linear. Then at a node ψ such that $C \vdash \psi : \texttt{Acquire}(\ell)$, the lock ℓ is clearly held. We iteratively propagate this fact forward through constraints $\psi \leq \psi'$ (and likewise for instantiation constraints) stopping propagation at any node ψ for which $C \vdash \psi : \texttt{Release}(\ell)$. At joins we intersect the sets of acquired locks. This continues until we reach a fixed point.

In essence this analysis computes the set of locks that *must* be acquired at each program point. Notice that because the analysis is necessarily conservative, we may decide at a program point that lock ℓ is not held even if it is at run time. This is safe because if our analysis inaccurately determines that a lock is released, at worst it will report a data race where no race is possible.

At function calls, denoted by another kind of ψ variable, we "split" the set of locks. At a split, we propagate the state of ℓ to the function's input state only if that function actually changes the state of (acquires or releases) ℓ, since otherwise the function must be polymorphic in ℓ's state. (Which locks are (transitively) mentioned by a function is determined by a standard, context-sensitive effect analysis.) The state of other locks is added to the output state of the function upon return. This is similar to *Merge* nodes in CQual [21]. Crucially, this optimization ensures we do not conflate lock states at calls to library functions such as `printf`. At instantiation sites $\psi \preceq_p^i \psi'$, we use the renaming defined by $\preceq_+^i$ to copy the states of any locks in the domain of the substitution corresponding to i from ψ to ψ', and vice-versa for $\preceq_-^i$ constraints.

Inferring Correlations and Finding Races Now, for each state variable ψ of kind $\texttt{Deref}(\rho)$, we generate a correlation constraint $\rho \triangleright \{\ell_1, \ldots, \ell_n\}$, where the ℓ_i are the set of locks held at ψ. (We have extended correlation constraints to include a set of locks rather than a single lock.)

Given the correlation constraints, we could apply the rules in Figure 8(b) to infer all correlations. However, because we "split" lock states at function calls, this would result in many false alarms, since a correlation constraint $\rho \triangleright \{\ell_1, \ldots, \ell_n\}$ generated inside a function really means that locks ℓ_i are held *in addition* to any locks held at the function's callers. Thus in LOCKSMITH, rather than inserting each correlation constraint into a global set C, we define a family of constraint sets C_ψ, one per ψ, and propagate them backwards along the control-flow constraint graph. When we reach a split node in the constraint graph, we add to each correlation constraint any held locks that were split off previously.

When we are done propagating, we check for consistent correlation among the correlation constraints $C_\psi^{\texttt{main}}$ which correspond to the initial state ψ of `main()`. As correlation constraints now refer to lock sets, we define

$$S(C, \rho) = \{\{\ell_1, \ldots, \ell_n\} \mid C \vdash \rho \triangleright \{\ell_1, \ldots, \ell_n\}\}$$

A location labeled ρ is consistently correlated if

$$|\bigcap S(C, \rho)| \geq 1$$

i.e., if there is at least one lock held every time ρ is accessed.

In the discussion thus far we have assumed that all locks are linear, but as per discussion in Section 2 this could be unsound. Rather than forbid non-linear locks, in LOCKSMITH we treat them as always released. Therefore non-linear locks are never included in correlation constraints, and so they do not prevent races from being reported. Our implementation currently allows most linearity checking to be optionally disabled, as we have found it is not very helpful in practice and can have a steep performance penalty. Our

```
int x;

void *f(...) {
  int *p = (int *) malloc(...);
  *p = x;
}

int main(void) {
  x = 42;
  pthread_create(..., f, ...);
  pthread_create(..., f, ...);
}
```

Figure 9. Example of Sharing Analysis

implementation also omits the right disjunct of the *escapes*$(\ell, \vec{l})$ check used for [Down] because it is not supported by BANSHEE. While possible, it is highly improbable that this omission will result in missed races.

3.2 Shared Locations

As an optimization, LOCKSMITH generates correlation constraints at states `Deref`(ρ) only when ρ may be thread-shared. Thus thread-local data need not be consistently correlated, which in practice substantially improves the precision and efficiency of LOCKSMITH.

We use several techniques to infer sharing. Our core technique is based on *continuation effects*. In the standard approach, the effect of an expression e denotes those locations read and written by e. In our approach, each expression has both input and output effects, ϵ_i and ϵ_o, where ϵ_o denotes the locations read and written in the program executed after e (including forked threads), while ϵ_i contains ϵ_o and those locations read and written in e itself. We compute continuation effects context-sensitively using BANSHEE.

When a new thread is created, we determine the locations it might share with its parent (or other threads the parent forks) as follows. Let ϵ_t be the input effect of the child thread, and let $\epsilon*$ be the *input closure* of a continuation effect ϵ, defined as all those locations ρ' that could flow to locations $\rho \in \epsilon$. Then $S = \epsilon_t^* \cap \epsilon_o^*$ is the possibly-shared locations due to the fork, where ϵ_o is the output effect of the parent. We prune S further to only mention ρ if it is written in either ϵ_t^* or ϵ_o^* (so that read-only access is not a race). For each `Deref`(ρ) state, we generate a correlation constraint for ρ if

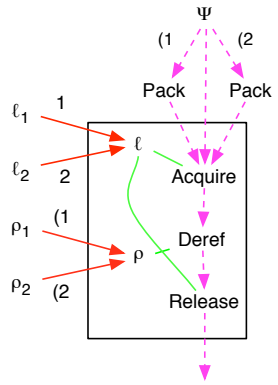
$$\rho^* \cap \bigcup_{\text{all } S} S \neq \emptyset$$

We make two improvements to this basic technique. First, rather than intersect ρ^* with *all* S, we consider only those S due to the forking of the current thread, its ancestors, or any child threads created by the current thread prior to the dereference of ρ; all dereferences in `main` prior to forking the first thread are considered unshared. This allows data to be accessed thread locally without protection, and only once it becomes shared must it be consistently correlated. Second, we apply a *Down-Fork* rule to further filter from S locations that do not escape a forked thread, and thus cannot be shared with its parent. In particular, suppose we spawn a thread t that may access location ρ. Then we observe that if $\rho^* \cap fl(\Gamma)^* = \emptyset$, where $fl(\Gamma)$ is the set of free labels in the types of variables visible at the point of the fork, then ρ is not visible outside the child thread and thus cannot be shared.

To see the benefit of these techniques, consider the code in Figure 9. This program initializes a global variable `x` and then forks two threads (using `pthread_create`) that invoke the function `f`, which reads `x` and writes it to freshly-allocated storage. Sharing analysis determines that `x` is never written after it becomes shared,

```
let m = newlock^ℓ1 in
let x = ref^ρ1 1 in
let p =
  if0 b then
    pack^1 (m, x)
  else
    pack^2 (newlock^ℓ2, ref^ρ2 2)
in
  unpack (l, r) = p in
    acquire l;
    r := 3;
    release l
```

(a) Source code



(b) Constraint graph

Figure 10. Existential Quantification

and hence does not require consistent correlation. Also notice that both copies of `f` allocate a location at the same syntactic position in the program. Thus our analysis assigns both allocations the same location ρ. A naive analysis would determine that ρ is shared because it is accessed by both child threads. Using Down-Fork, however, we observe that ρ does not escape the body of `f` and hence is thread-local.

Finally, our implementation also includes a *uniqueness* analysis. We perform a very basic, intraprocedural, flow-sensitive alias analysis to determine when local variables definitely point to thread-local memory. For example, consider the following code:

```
int* x = (int *) malloc(sizeof(int));
*x = 2;
lock(l);
shared = x; /* becomes shared */
```

Here `x` points to newly-allocated memory that is subsequently initialized. Then `x` is assigned to `shared`—a variable visible to another thread—after acquiring lock `l`. The assignment causes `*x` to be an alias of `shared`; if this code occurs in a routine run by multiple threads, our earlier sharing analysis will think that `x` is always thread-shared. But our local uniqueness analysis observes that at the write to `*x`, the variable `x` has not yet escaped, and hence the write is ignored for purposes of correlation.

3.3 Existential Quantification for Data Structures

In applying our system to C programs, we found several examples where locks are stored in heap data structures along with the data they protect. Standard context-sensitive analyses typically merge all elements of the same data structure into an indistinguishable "blob," which would cause us to lose track of the identities of locations and the linearities of locks in data structures. In this subsection we briefly sketch an approach to solving this problem that has proven effective for one of our benchmarks.

As an example, consider the program in Figure 10(a). This program first binds m to a new lock labeled ℓ_1, and then binds x to a new reference labeled ρ_1 (here for convenience we mark labels in the source code directly). The program then sets p to be one of two pairs. The *pack* operation alerts our analysis that the pairs should be treated abstractly so that we can conflate them without losing correlations. Next the program *unpacks* p and acquires the pair's lock before dereferencing its pointer.

Notice that although r may be either ρ_1 or ρ_2 at runtime, and l may be either ℓ_1 or ℓ_2, in either case the correct lock will be acquired. Because we used *pack* before the data structure was

conflated, our analysis gives p the type

$$\exists \ell, \rho[\rho \rhd \{\ell\}].\ \mathit{lock}\ \ell \times \mathit{ref}^{\rho}\ \mathit{int}$$

meaning that p contains some lock ℓ and some location ρ where ℓ and ρ are correlated.

One key novelty of LOCKSMITH is that, given a program with `pack` and `unpack` annotations, we perform *inference* on existential types using constraint resolution rules similar to those in Figure 8. Figure 10(b) shows the constraint graph for our example. Rather than give resolution rules explicitly, we discuss the algorithm informally on this example. Existential inference using this basic technique is sound for the related problem of label flow [38].

In this figure, we represent data flow from labels ℓ_i and ρ_i to the packed labels ℓ and ρ with directed edges annotated with the `pack` site. It is no coincidence that this is the same notation used for universal quantification in Figure 3—it is the duality of universal and existential quantification that lets us use similar techniques for both. The remaining edges show the states at the various program points. Initially we are in some state ψ. Then we pack one of the two pairs, represented by a split labeled with (i for pack site i. Within the unpack (shown in the box), we acquire lock l, dereference r, and then release l. At the dereference site, lock l is held, and so we generate a constraint $r \rhd \{l\}$ (not shown in the graph). We propagate this correlation constraint using matched flow as in Figure 8(b) and generate two constraints, $\rho_1 \rhd \{\ell_1\}$ and $\rho_2 \rhd \{\ell_2\}$. Had we not used existential quantification here, we would not have been able to track correlation precisely, because ℓ_1 and ℓ_2 would have been non-linear, and there would have been no way to tell which goes with ρ_1 and which goes with ρ_2.

LOCKSMITH supports existential types for `structs`. To use existentials, the programmer annotates aggregates that can be packed to indicate which fields should have bound types after packing. We extend C with a special `pack(x)` statement that makes `x`'s type existentially quantified. For unpacking, the programmer inserts `start_unpack(x)` and `end_unpack(x)` statements, which begin and end the scope of the unpack, possibly non-lexically. In Section 4, we show that existential quantification is useful for one of our benchmarks. We needed to add a total of 29 pack, unpack, and field annotations to the program that could benefit, and 3 of the 12 `start_unpack` operations are not lexically scoped.

3.4 Analysis of `void*` and Aggregates

We aim to be sound, and so we use a number of techniques to conservatively model the unsafe aspects of C without losing too much precision. At type casts of dissimilar types we conflate locations in the actual and cast-to type. However, for `void*` pointers, we instead maintain a set of non-`void*` types that are cast to or from them [29]. Any type cast to or from that `void*` at the same type is unified with the matching type stored in the `void*`. This technique enables us to model the common case when `void*` is used for polymorphism, which is important because the POSIX `pthread_create` routine takes a `void*` argument that is passed into the function called when the new thread starts. However, using this model of `void*` is unsound, because it does not handle up- or downcasts of `struct` types and assumes programmers use `void*` safely. Nevertheless, we have found it effective in practice, and it makes the output of LOCKSMITH much easier to interpret by reducing conflation.

Some `struct` types in C programs may have many fields (we have seen cases of 100 or more), many of which themselves have `struct` types containing many fields. For precision, we wish to assign fresh labels to fields of different instances of the same `struct` type. However, if we model these types naively by representing all fields of all instances of aggregates, then the analysis becomes very inefficient. Instead, we represent structure fields lazily [29], so that we only model those fields that are actually used. This optimization does not affect precision, but can provide a significant speedup.

Benchmark	Size (KLOC)	Time	Warn.	Unguarded	Races
aget	1.6	0.8s	15	15	15
ctrace	1.8	0.9s	8	8	2
pfscan	1.7	0.7s	5	0	0
engine	1.5	1.2s	7	0	0
smtprc	6.1	6.0s	46	1	1
knot	1.7	1.5s	12	8	8

Table 1. Summary of Experimental Results: POSIX Apps

4. Experiments

We evaluated LOCKSMITH on a modest set of benchmarks, including several small applications and medium-sized Linux kernel drivers. We conducted our experiments on a dual Xeon@2.8GHz PC with 3.5GB of RAM, running RedHat Enterprise Linux, kernel version 2.4.21. LOCKSMITH was compiled using OCaml 3.08.1, with all C code (BANSHEE and the OCaml runtime system) compiled with `gcc` 3.2.3-53 at optimization level `-O2`. Reported elapsed times are the median of 7 runs.

4.1 POSIX Threads Applications

We selected several multi-threaded programs largely gathered from `sourceforge.net`. *Aget* is an FTP client in which multiple threads download chunks of a file. *Ctrace* is a library for tracing the execution of multi-threaded programs; we analyzed a sample application that came with its distribution. *Pfscan* is a multithreaded file scanner that combines the functionality of `find`, `xargs`, and `fgrep`. *Engine* issues requests to several search engines in parallel and collates the responses. *Smtprc* is an open mail relay scanner that looks for potential configuration problems. Finally, *knot* is a multi-threaded webserver distributed with the Capriccio user-level threads package [48].

In our experiments we measured the number of warnings reported by LOCKSMITH and how many of those warnings correspond to races. Here is a somewhat simplified warning taken from aget:

```
Possible data race on
  &bwritten(aget_comb.c:943)
References:
  dereference at aget_comb.c:1079
  locks acquired at dereference:
    &bwritten_mutex(aget_comb.c:996)
  in: FORK at aget_comb.c:468 ->
    http_get aget_comb.c:468

  dereference at aget_comb.c:984
  locks acquired at dereference:
    (none)
  in: FORK at aget_comb.c:193 ->
    signal_waiter(aget_comb.c:193) ->
    sigalrm_handler(aget_comb.c:957)
```

The first part indicates where the data that might be accessed in race is allocated, in this case the global variable `bwritten` defined at line 943. The second part lists where that location may be dereferenced, along with the locks held at that point and the context-sensitive control-flow path that led to the dereference. Above we show two (out of many) accesses. The first is in a thread running the function `http_get` with the mutex `&bwritten_mutex` held. The second access is in another thread running `signal_waiter`, which has called the function `sigalrm_handler`; this takes place with no lock held, violating consistent correlation. In practice this situation could arise when the user terminates aget abruptly with a

Benchmark	Size (KLOC)	Time	Warn.	Unguarded	Races
plip	19.1	24.9s	11	2	1
eql	16.5	3.2s	3	0	0
3c501	17.4	240.1s	24	2	2
sundance	19.9	98.2s	3	1	0
sis900	20.4	61.0s[1]	8	2	1
slip	22.7	16.5s[1]	19	1	0
hp100	20.3	31.8s[1]	23	2	0

Table 2. Summary of Experimental Results: Linux Drivers

signal, which causes it to save its current state to disk. The race on `bwritten` could cause it to be confused when the program restarts.

Table 1 shows the experimental results for these application programs. We merge multi-file programs into a single C file using the CIL merger, which eliminates duplicate and unused declarations. The table lists the size of the merged program in lines of preprocessed code. Since the application programs use the standard C library, we constructed a stub file containing function definitions that model the data flow and effects of libc routines used in our benchmarks, totaling roughly 400 LOC (not counted in the table).

The third column shows the total number of warnings of potential races issued by LOCKSMITH. The "Unguarded" column lists the number of warnings that constitute true violations of consistent correlation. Some of these may not be races because a shared location is protected using other techniques. The last column lists the number of true races, in which data could be accessed simultaneously by two threads and one of the accesses is a write.

For aget, most of the races are similar to the example above. For knot, all the races seem to be benign; most are due to unprotected accesses to global variables used to gather statistics. Ctrace uses semaphores to communicate among threads, so while 6 guarded-by violations reported are legitimate, the data is not subject to races. The two real races are on global variables; one is benign, but the other is used to communicate information between threads, and a race could cause messages to be lost. Finally, smtprc has one race that occurs when a reaper thread sets a global counter that is also set and read by the main thread, which could cause unpredictable behavior.

LOCKSMITH also reported a number of false alarms, mostly arising from two coding idioms that LOCKSMITH does not handle. One is when a parent thread accesses previously shared data after its child threads have died (3 for engine, 1 for pfscan), as determined by `pthread_mutex_join` or other signaling mechanisms. Another is when a global data structure points to thread-local data, indexed by thread identifier (4 in engine, 44 in smtprc). The remaining false alarms could be handled with some improvements to the local sharing analysis (3 for knot), and to allowable idioms of existential initialization (3 for pfscan).

We also tried to run LOCKSMITH on several larger programs, but were ultimately unsuccessful due to resource exhaustion. We do not believe these problems are fundamental, and plan to continue to investigate how LOCKSMITH can be applied to larger programs.

4.2 Device Drivers

In addition to application code, we applied LOCKSMITH to a set of Linux device drivers. We found that determining synchronization assumptions for device drivers is challenging because the internal Linux API is complex and sparsely documented. Complicating matters, earlier versions of the kernel used a single spin lock ("the big kernel lock" or BKL) to prevent parallel access within the kernel, and remnants of this discipline remain. For example, as far as we can tell, character driver operations are always called with the BKL held, removing the need for multi-processor synchronization.

Therefore, we chose to apply LOCKSMITH to network device drivers, which must use internal locking and are relatively well-documented. We focused on seven drivers from the 2.6.12 kernel: plip (parallel-line IP), slip (serial line IP), eql (network traffic equalizer), and sis900, 3c501, hp100 and sundance (ethernet card drivers). We constructed stub files with a special `main` routine that simulates the kernel's concurrent interaction with the driver via interrupts, timeouts, and user process-induced calls. LOCKSMITH models kernel spin locks in the same way it models POSIX mutexes. We ran LOCKSMITH with some linearity checking disabled, because currently our semi-unification algorithm does not terminate for some drivers. We believe this can be fixed by implementing an extended occurs check [24].

Table 2 lists our results. We found a total of 4 races. The races in plip and sis900 are both benign races on counters. One race in 3c501 is a presumably benign race on a debugging flag. The other race in 3c501 occurs on a flag that tracks whether the driver is transmitting. We believe a race on this flag could cause errors if it occurs in the middle of a send operation. The guarded-by violations that were not races were due to the use of *atomic operations* which are always thread safe. These are implemented with inline assembly code that LOCKSMITH processes conservatively.

The main source of false alarms in drivers is due to conflation and other conservatism due to type casts. This causes LOCKSMITH mistakenly to think locations could be shared when they are not. Several of the false alarms could be addressed using existentials, in principle, to model a lock stored in a data structure, but we currently cannot check the initialization pattern. Finally, in many cases synchronization is context dependent, employing state variables and other non-lock-based forms. It was difficult to tell in these cases whether a race existed or not, since we are not kernel experts, so we considered them to be non-races in the table.

4.3 Per-feature Effectiveness

We examined the various features described in Section 3 for improving the precision of the analysis. Table 3 shows the number of warnings issued by the tool depending on which techniques are enabled, along with the corresponding running time. In particular, we measured the cumulative effectiveness of four techniques: (1) our technique for modeling a `void*` cast to/from a single type precisely (Void), as compared to conflating all locations at all levels of a type cast to `void*`; (2) the use of Down-Fork to reduce false sharing (DownF); (3) flow-sensitive uniqueness analysis of local variables (Uniq); and (4) using existential quantification to model locks local to data structures (Exist), which only affects the knot benchmark. All but the last feature are fully automatic, while existentials require manual insertion of packs and unpacks; we used 29 annotations for knot.

For a more visual comparison, we show the normalized effect of each technique on precision in Figure 5. The non-black portion of each bar is the scaled improvement due to the addition of that particular feature. For example, we can see that for aget, 60% (26 of 43) warnings were eliminated due to precise handling of `void*` while an additional 4.5% (2 of 43) were removed due to Down-Fork. Void and DownF are clearly the most useful overall. In contrast to Void and DownF, Exists and Uniq are useful only in a few cases, and may increase running time.

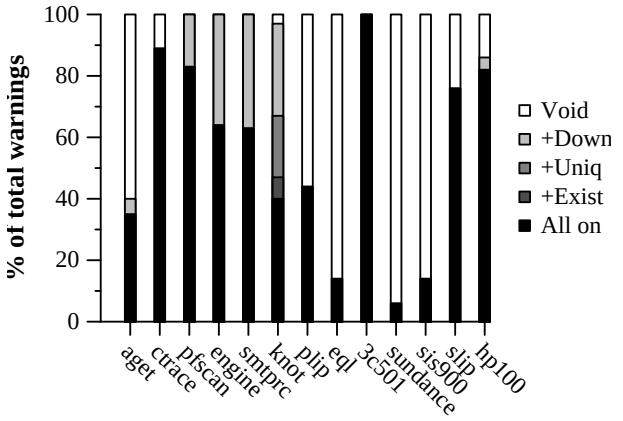
5. Related Work

A number of systems have been developed for detecting data races and other concurrency errors in multi-threaded programs, including dynamic analysis, static analysis, and hybrid systems.

[1] Did not perform linearity checking

Benchmark	All off	+Void	+DownF	+Uniq	+Exist
aget	**43** 1.5s	**17** 0.9s	**15** 0.8s	**15** 0.8s	**15** 0.8s
ctrace	**9** 1.1s	**8** 0.9s	**8** 0.9s	**8** 0.9s	**8** 0.9s
pfscan	**6** 0.7s	**6** 0.7s	**5** 0.7s	**5** 0.7s	**5** 0.7s
engine	**11** 1.0s	**11** 1.0s	**7** 1.0s	**7** 1.2s	**7** 1.2s
smtprc	**73** 5.6s	**73** 5.8s	**46** 5.0s	**46** 6.0s	**46** 6.0s
knot	**30** 1.2s	**29** 1.1s	**20** 1.0s	**14** 0.9s	**12** 1.5s
plip	**25** 27.5s	**11** 23.8s	**11** 24.0s	**11** 24.9s	**11** 24.9s
eql	**22** 2.9s	**3** 3.1s	**3** 3.1s	**3** 3.2s	**3** 3.2s
3c501	**24** 233.4s	**24** 238.3s	**24** 238.7s	**24** 240.1s	**24** 240.1s
sundance	**52** 53.4s	**3** 98.5s	**3** 99.6s	**3** 98.2s	**3** 98.2s
sis900	**57** 40.5s	**8** 59.6s	**8** 60.5s	**8** 61.0s	**8** 61.0s
slip	**25** 7.7s	**19** 16.2s	**19** 16.4s	**19** 16.5s	**19** 16.5s
hp100	**28** 18.2s	**24** 31.1s	**23** 31.6s	**23** 31.8s	**23** 31.8s

Table 3. Summary of per-feature effects



Dynamic systems such as Eraser [44] instrument a program to find data races at run time and require no annotations. The efficiency and precision of dynamic systems can be improved with static analysis [7, 36, 1]. Dynamic systems are fast and easy to use, but cannot prove the absence of races, and require comprehensive test suites.

Researchers have developed type checking systems against races [13] for several languages, including Java [14], Java variants [6], and Cyclone [22]. In general, systems based on type checking perform very well, but require a significant number of programmer annotations, which can be time consuming when checking large code bases [10, 15]. Static race detection in ESC/Java [19],which employs a theorem prover, similarly requires many annotations.

Some researchers have developed tools to automatically infer the annotations needed by the Java-based type checking systems just mentioned. Most target Java 1.4, which simplifies the problem by permitting only lexically-acquired locks via `synchronized` statements, whereas C (and Java 1.5) programs may acquire and release locks at any program point. Houdini [15] can infer types for the original race-free Java system [14], but lacks context-sensitivity. More recently Agarwal and Stoller [2] and Rose et al [43] have developed algorithms that infer types based on dynamic traces, but these require sizeable test suites to avoid excessive false alarms. Flanagan and Freund [17] have proposed a system for inference which is formulated to support parameterized classes and dependent types. Though the problem is NP-complete, their SAT-based approach can analyze 30K lines of Java code in 46 minutes. Von Praun and Gross's dataflow-based system also requires no annotations and performs well, checking 2000-line programs in a few seconds.

Naik, Aiken, and Whaley present a race detection system for Java [34]. Their system scales well to large Java programs and has found a number of races. They use a cloning-based alias analysis, and hence their approach does not suffer the summarization problem mentioned in Section 2.1 for other context-sensitive analyses. Analyzing Java 1.4 avoids some problems we encountered analyzing C code, such as flow sensitive locking, low-level pointer operations, and unsafe type casts. They also omit linearity checking, which we include in $\lambda_\triangleright$ but occasionally disable in LOCKSMITH.

Several completely automatic static analyses have been developed for finding races in C code. Polyspace [27] is a proprietary tool that uses abstract interpretation to find data races (and other problems). The Blast model checker has been used to find data races in programs written in NesC, a variant of C [26]. Race checking is not limited to checking for consistent correlation and can be state dependent, but is limited to checking global variables and can be quite expensive. Seidl et al [45] propose a framework for analyzing multi-threaded programs that interact through global variables. Using their framework they develop a race detection system for C and apply it to a small set of benchmarks, finding a number of data races. It is unclear whether their analysis supports context sensitivity and how it models data structures. RacerX [10] does not soundly model some features of C for better scalability and to reduce false alarms, but may miss races as a result. KISS [39] builds on model checking techniques, and has been shown to find many races, but ignores possible thread interleavings, possibly missing the most subtle bugs.

Work that detects violations of *atomicity*, either dynamically [16] or statically [20, 18] typically requires a program to be free of races.

Our analysis is based on ideas initially explored by Reps et al [41] and Rehof and Fähndrich [40], who showed how to encode context-sensitive analysis as a context-free language reachability problem. Our support for existential types is related to `restrict` or `focus` for alias analysis [3, 11]. Our flow-sensitive analysis is a significant extension of our previous work on flow-sensitive type qualifiers [21], which used a similar flow-sensitive constraint graph. Both systems can be seen as inference for a variant of the calculus of capabilities [8].

Correlation between locks and locations is similar to correlation between regions and pointers, and several researchers have looked at the problem of region inference, including the Tofte and Birkedal system for the ML Kit [47]. Henglein et al [25] use a control-flow-sensitive and context-sensitive type system to check that regions with non-lexical allocation and deallocation are used correctly. Our treatment of lock allocation is similar to Henglein et al's treatment of region allocation, but our formal system supports higher-order functions, and we present a constraint-based inference algorithm.

6. Conclusion

We have developed a tool, LOCKSMITH, that aims to prove the absence of data races in a C program. The core component of LOCKSMITH is a context-sensitive *correlation analysis* that determines whether there exists a lock that is held consistently each time a memory location is accessed. This paper formalizes correlation

analysis as a constraint-based type and effect system for a simple language $\lambda_{\triangleright}$ which we have proven sound. A novel feature of our formalism is its use of effects to ensure that dynamically-allocated locks can be accurately tracked, with a means to safely hide effects to better support recursive functions. LOCKSMITH uses a series of techniques to scale correlation analysis to the full C language, including flow-sensitive state tracking, existential types, sharing analysis, and heuristics to model type casts to and from `void*`. When applied to a set of benchmarks, LOCKSMITH discovered a number of real data races with a modest rate of false alarms. We are continuing to explore how to scale LOCKSMITH to large code bases.

Acknowledgments

This research was supposed in part by NSF CCF-0346989, CCF-0430118, and CCF-0524036. We thank Dan Grossman, Greg Morrisett, Boniface Hicks, Nik Swamy, and the anonymous reviewers for their helpful comments. We also thank Will Dogan, Iulian Neamtiu, and Pavlos Papageorgiou for help with the Linux drivers.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *ASE*, 2005.

[2] R. Agarwal and S. D. Stoller. Type Inference for Parameterized Race-Free Java. In *VMCAI*, 2004.

[3] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *PLDI*, 2003.

[4] A. Alexandrescu, H. Boehm, K. Henney, B. Hutchings, D. Lea, and B. Pugh. Memory model for multithreaded C++: Issues, 2005. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1777.pdf`.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[6] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, 2002.

[8] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *POPL*, 1999.

[9] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *SAS*, 2001.

[10] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[11] M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, 2002.

[12] M. Fähndrich, J. Rehof, and M. Das. From Polymorphic Subtyping to CFL Reachability: Context-Sensitive Flow Analysis Using Instantiation Constraints. Technical Report MSR-TR-99-84, Microsoft Research, 1999.

[13] C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.

[14] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[15] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *PASTE*, 2001.

[16] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL*, 2004.

[17] C. Flanagan and S. N. Freund. Type Inference Against Races. In *SAS*, 2004.

[18] C. Flanagan, S. N. Freund, and M. Lifshin. Type Inference for Atomicity. In *TLDI*, 2005.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[20] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, 2003.

[21] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, 2002.

[22] D. Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, 2003.

[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[24] F. Henglein. Type Inference with Polymorphic Recursion. *TOPLAS*, 15(2), 1993.

[25] F. Henglein, H. Makholm, and H. Niss. A Direct Approach to Control-Flow Sensitive Region-Based Memory Management. In *PPDP*, 2001.

[26] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.

[27] C. Hote. Run-Time Error Detection Through Semantic Analysis, 2004. `http://www.polyspace.com/pdf/Semantics_Analysis.pdf`.

[28] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

[29] R. Johnson and D. Wagner. Finding User/Kernel Bugs With Type Inference. In *USENIX Security*, 2004.

[30] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*. London, United Kingdom, 2005.

[31] N. Leveson and C. S. Turner. An investigation of the therac-25 accidents, July 1993.

[32] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *POPL*, 1996.

[33] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[34] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, 2006. To appear.

[35] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *ICCC*, 2002.

[36] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[37] K. Poulsen. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, 2004.

[38] P. Pratikakis, M. Hicks, and J. S. Foster. Existential Label Flow Inference via CFL Reachability. Technical Report CS-TR-4700, Department of Computer Science, UMD, 2005. Forthcoming.

[39] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, 2004.

[40] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL*, 2001.

[41] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

[42] J. C. Reynolds. Towards a Grainless Semantics for Shared Variable Concurrency. In *POPL*, 2004.

[43] J. Rose, N. Swamy, and M. Hicks. Dynamic inference of polymorphic lock types. *Science of Computer Programming*, 2005.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, 1997.

[45] H. Seidl, V. Vene, and M. Müller-Olm. Global Invariants for Analyzing Multi-threaded Applications. In *Proc. of Estonian Academy of Sciences: Phys., Math.*, volume 52, pages 413–436, 2003.

[46] F. Smith, D. Walker, and G. Morrisett. Alias Types. In *ESOP*, 2000.

[47] M. Tofte and L. Birkedal. A Region Inference Algorithm. *TOPLAS*, 20(4), 1998.

[48] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

[49] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL*, 1999.