

Static Race Detection for C using Locksmith

Jeff Foster
University of Maryland

Introduction

- Concurrent programming is hard
 - Google for “notoriously difficult” and “concurrency”
 - 58,300 hits
- One particular problem: *data races*
 - Two threads access the same location “simultaneously,” and one access is a write

Consequences of Data Races

- Data races cause real problems
 - 2003 Northeastern US blackout
 - One of the “top ten bugs of all time” due to races
 - <http://www.wired.com/news/technology/bugs/1,69355-0.html>
 - 1985-1987 Therac-25 medical accelerator
- Race-free programs are easier to understand
 - Many semantics for concurrent languages assume correct synchronization
 - It's hard to define a memory model that supports unsynchronized accesses
 - C.f. The Java Memory Model, recent added to Java Spec

Avoiding Data Races

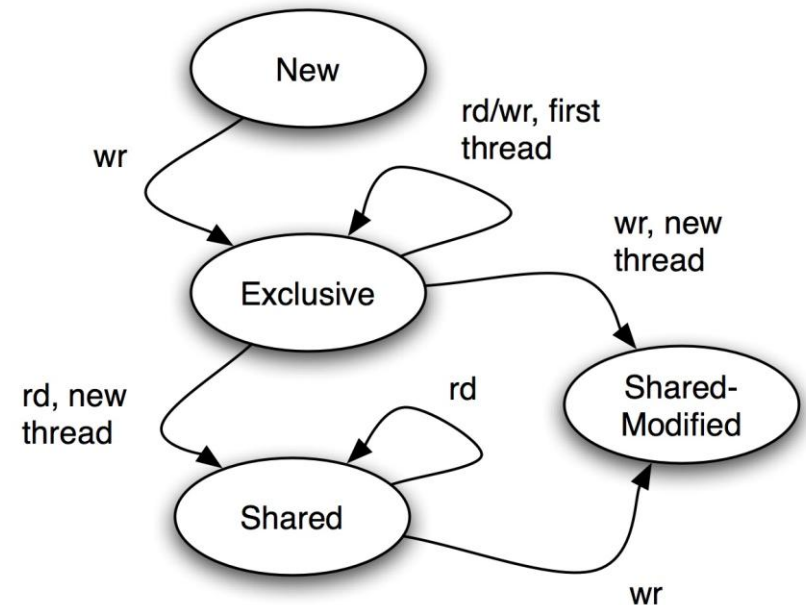
- The most common technique:
 - Locations r
 - Locks l
 - Correlation: $r @ l$
 - Location r is accessed when l is held
 - *Consistent correlation*
 - Any shared location is only ever correlated with one lock
 - We say that that lock *guards* that location
 - Implies race freedom
- Not the only technique for avoiding races!
 - But it's simple, easy to understand, and common

Eraser [Savage et al, TOCS 1997]

- A dynamic tool for detecting data races based on this technique
 - $\text{Locks_held}(t)$ = set of locks held by thread t
 - For each r , set $C(r) := \{ \text{all locks} \}$
 - On each access to r by thread t ,
 - $C(r) := C(r) \cap \text{locks_held}(t)$
 - If $C(r) = \emptyset$, issue a warning

An Improvement

- Unsynchronized *reads* of a shared location are OK
 - As long as no on writes to the field after it becomes shared
- Track state of each field
 - Only enforce locking protocol when location shared and written



Safety and Liveness Tradeoffs

- Programs should be *safe*, so that they do not have data races
 - Adding locking is one way to achieve safety
 - (Note: not the only way)
- Programs should be *live*, so that they make progress
 - *Removing* locking is one way to achieve liveness!

Data Races in Practice

- Programmers worry about performance
 - A good reason to write a concurrent program!
 - Hence want to avoid unnecessary synchronization
- ==> Ok to do unsafe things that “don’t matter”
 - Update a counter
 - Often value does not need to be exact
 - But what if it’s a reference count, or something critical?
 - Algorithm works ok with a stale value
 - The algorithm will “eventually” see the newest values
 - Need deep reasoning here, about algorithm and platform
 - And others

Concurrent Programming in C

- Many important C programs are concurrent
 - E.g., Linux, web servers, etc
- Concurrency is usually provided by a library
 - Not baked into the language
 - But there is a POSIX thread specification
 - Linux kernel uses its own model, but close

A Static Analysis Against Races

- Goal: Develop a tool for determining whether a C program is race-free
- Design criteria:
 - Be sound: Complain if there is a race
 - Handle locking idioms commonly-used in C programs
 - Don't require many annotations
 - In particular, do *not* require the program to describe which locations are guarded by what locks
 - Scale to large programs
- Can reuse many of the type qualifiers ideas!

Outline

- C locking idioms
- Alias analysis = type qualifier inference
- Extend to infer correlations
- Making it work in practice for C
- Using alias analysis to detect sharing

A Hypothetical Program: Part 1

```
lock_t log_lock; /* guards logfd, bw */
int logfd, bw = 0;
void log(char *msg) {
    int len = strlen(msg);
    lock(&log_lock);
    bw += len;
    write(logfd, msg, len);
    unlock(&log_lock);
}
```

Acquires `log_lock` to protect access to `logfd`, `bw`

However, assumes caller has necessary locks to guard `*msg`

A Hypothetical Program: Part 2

```
struct job {  
    lock_t j_lock; /* guards worklist and cnt */  
    struct job *next;  
    void *worklist;  
    unsigned cnt;  
};  
lock_t list_lock; /* guards list backbone */  
struct job *joblist;
```

Data structures can include locks

Sometimes locks guard individual elements, sometimes they guard sets of elements (and sometimes even more complex)

A Hypothetical Program: Part 3

```
void logger() { ...  
    lock(&list_lock);  
    for (j = joblist; j != NULL; j = j->next) {  
        cnt++;  
        if (trylock(&j->job_lock)) {  
            sprintf(msg, "...", cnt, j->cnt);  
            log(msg);  
            unlock(&j->job_lock);  
        }  
    }  
    unlock(&list_lock); ... }
```

trylock returns false (and does not block) if lock already held
locking appears at arbitrary program points

A Hypothetical Program: Part 4

```
int main(int argc, char **argv) {...  
    for (i = 0; i < n; i++) {  
        struct job *x = malloc(sizeof(struct job));  
        /* initialize x */  
        fork(worker, x);  
    }  
}
```

x is thread-local during initialization, and only becomes shared once thread is forked
and all of this happens within a loop

Summary: Key Idioms

- Locks can be acquired or released anywhere
 - Not like synchronized blocks in Java
- Locks protect static data and heap data
 - And locks themselves are both global and in data structures
- Functions can be polymorphic in the relationship between locks and locations
- Much data is thread-local
 - Either always, or up until a particular point
 - No locking needed while thread-local

Other Possible Idioms (Not Handled)

- Locking can be *path-sensitive*
 - `if (foo) lock(&x) ... if (foo) unlock(&x)`
- Reader/writer locking
- Ownership of data may be transferred
 - E.g., thread-local data gets put into a shared buffer, then pulled out, at which point it becomes thread-local to another thread

First Task: Understand Pointers

- We need to know a lot about pointers to build a tool to handle these idioms
 - We need to know which locations are accessed
 - We need to know what locks are being acquired and released
 - We need to know which locations are shared and which are thread local
- The solution: Perform an alias analysis

Alias Analysis

Introduction

- *Aliasing* occurs when different names refer to the same thing
 - Typically, we only care for imperative programs
 - The usual culprit: pointers
- A core building block for other analyses
 - ...`*p = 3;` // What does `p` point to?
- Useful for many languages
 - C — lots of pointers all over the place
 - Java — “objects” point to updatable memory
 - ML — ML has updatable references

May Alias Analysis

- p and q *may alias* if it's possible that p and q might point to the same address
- If not (p may alias q), then a write through p does not affect memory pointed to by q
 - ... $*p = 3$; $x = *q$; // write through p doesn't affect x
- Most conservative may alias analysis?
 - Everything may alias everything else

Must Alias Analysis

- p and q *must alias* if p and q do point to the same address
 - If p must alias q , then p and q refer to the same memory
 - ... $*p = 3$; $x = *q$; // x is 3
- What's the most conservative must alias analysis?
 - Nothing must alias anything

Early Alias Analysis (Landi and Ryder)

- Expressed as computing alias pairs
 - E.g., $(*p, *q)$ means p and q may point to same memory
- Issues?
 - There could be many alias pairs
 - $(*p, *q), (p \rightarrow a, q \rightarrow a), (p \rightarrow b, q \rightarrow b), \dots$
 - What about cyclic data structures?
 - $(*p, p \rightarrow \text{next}), (*p, p \rightarrow \text{next} \rightarrow \text{next}), \dots$

Points-to Analysis (Emami, Ghiya, Hendren)

- Determine set of locations **p** may point to
 - E.g., **(p, {&x})** means **p** may point to the location **x**
 - To decide if **p** and **q** alias, see if their points-to sets overlap
- More compact representation
- Need to name locations in the program
 - Pick a finite set of possible location names
 - No problem with cyclic structures
 - **x = malloc(...);** // where does **x** point to?
 - **(x, {malloc@257})** "the malloc at line 257"

Flow-Sensitivity

- An analysis is *flow-sensitive* if it tracks state changes
 - E.g., data flow analysis is flow-sensitive
- An analysis is *flow-insensitive* if it discards the order of statements
 - E.g., type systems are flow-insensitive
- Flow-sensitivity is much more expensive, but also more precise

Example

```
p = &x;  
p = &y;  
*p = &z;
```

Flow-sensitive:

```
p = &x; // (p, {&x})  
p = &y; // (p, {&y})  
*p = &z; // (p, {&y}), (y, {&z})
```

Flow-insensitive:

```
(p, {&x, &y})  
(x, &z)  
(y, &z)
```

Points-to Analysis via Type Qualifiers

- Create a type qualifier constant for each memory location in the program
 - `x = malloc(...)` becomes `x = annot(malloc@257, ...)`
 - `malloc@257` = "The memory allocated at line 257"
 - `int x` becomes `x = annot(x@42, ref 0)`
 - `x@42` = "The memory allocated for x at line 42"
- Perform type qualifier inference
 - Will never fail—no inconsistent qualifiers
- At each dereference `*p`, compute the set of qualifiers that may reach p

Example

```
let x = refRx 0 in  
  let y = x in  
y := 3;
```

- x has type $\text{ref}^{Rx}\text{int}$
- y has the same type as x
- Therefore at assignment, we know which location y refers to

Another Example

```
let x = refRx0 in  
let y = refRy0 in  
let w = refRw 0 in  
let z = if 0 42 then x else y in  
z := 3;
```

- Constraints $R_x \leq R_z$, $R_y \leq R_z$
- At assignment, we write to location R_z
 - Notice that we don't know which of x, y we write to
 - But we do know that we don't affect w

Andersen's Analysis

- Flow-insensitive
- Context-insensitive
- Subtyping-based
 - = Andersen's analysis
 - \sim = Das's "one-level flow"
- Properties
 - Still very scalable in practice
 - Much less coarse than Steensgaard's analysis
 - Can still be improved (will see later)

Back to Race Detection

Programming Against Races

- Recall our model:
 - Locations r
 - Locks l
 - Correlation: $r @ l$
 - Location r is accessed when l is held
 - *Consistent correlation*
 - Any shared location is only ever correlated with one lock
 - We say that that lock *guards* that location
 - Implies race freedom

Applying Alias Analysis

- Recall our model:
 - Locations r
 - Drawn from a set of constant labels R , plus variables a
 - We'll get these from (may) alias analysis
 - Locks l
 - Hm...need to think about these
 - Draw from a set of constant lock labels L , plus variables m
 - Correlation: $r @ l$
 - Hm...need to associate locks and locations somehow
 - Let's punt this part

Lambda-Corr

- A small language with “locations” and “locks”

$e ::= x \mid n \mid \backslash x:t.e \mid e e \mid \text{if0 } e \text{ then } e \text{ else } e$
 $\mid \text{newlock}^L$ create a new lock
 $\mid \text{ref}^R e$ allocate “shared” memory
 $\mid !^e e$ dereference with a lock held
 $\mid e :=^e e$ assign with a lock held
 $t ::= \text{int} \mid t \rightarrow t \mid \text{lock } l \mid \text{ref}^r t$

- No acquire and release
 - All accesses have explicit annotations (superscript) of the lock
 - This expression evaluates to the lock to hold
- No thread creation
 - ref creates “shared” memory
 - Assume any access needs to hold the right lock

Example

```
let k1 = newlockL1 in
let k2 = newlockL2 in
let x = refRx 0 in
let y = refRy 1 in
  x :=k1 3;
  x :=k1 4;      // ok — Rx always accessed with L1
  y :=k1 5;
  y :=k2 6      // bad — Ry sometimes accessed
                  with L1 or L2
```

Type Inference for Races

- We'll follow the same approach as before
 - Traverse the source code of the program
 - Generate constraints
 - Solve the constraints
 - Solution \Rightarrow program is consistently correlated
 - No solution \Rightarrow potential race
 - Notice that in alias analysis, there was always a solution
- For now, all rules except for locks and deref, assignment will be the same

Type Rule for Locks

- For now, locks will work just like references
 - Different set of qualifiers for them
 - Standard labeling rule, standard subtyping
 - Warning: this is broken! Will fix later...

$$\frac{L \leq m \quad m \text{ fresh}}{A \vdash \text{newlock}^L : \text{lock } m}$$

$$\frac{l_1 \leq l_2}{\text{lock } l_1 \leq \text{lock } l_2}$$

Correlation Constraints for Locations

- Generate a *correlation constraint* $r @ l$ when location r is accessed with lock l held

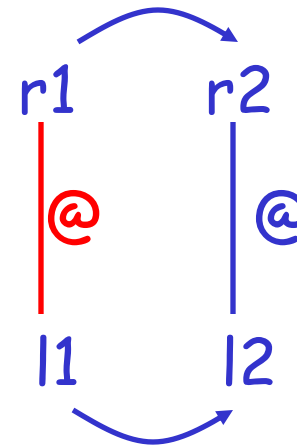
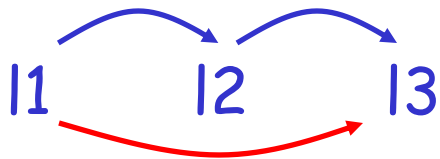
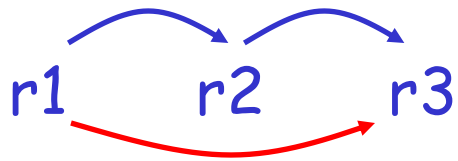
$$\frac{A \vdash e1 : \text{ref}^r t \quad A \vdash e2 : \text{lock } l \quad r @ l}{A \vdash !^{e2} e1 : t}$$

$$\frac{A \vdash e1 : \text{ref}^r t \quad A \vdash e2 : t \quad A \vdash e3 : \text{lock } l \quad r @ l}{A \vdash e1 :=^{e3} e2 : t}$$

Constraint Resolution

- Apply subtyping until only atomic constraints
 - $r1 \leq r2$ — location subtyping
 - $l1 \leq l2$ — lock subtyping
 - $r @ l$ — correlation
- Now apply three rewriting rules
 - $S + \{ r1 \leq r2 \} + \{ r2 \leq r3 \} + \Rightarrow \{ r1 \leq r3 \}$
 - $S + \{ l1 \leq l2 \} + \{ l2 \leq l3 \} + \Rightarrow \{ l1 \leq l3 \}$
 - $S + \{ r1 \leq r2 \} + \{ l1 \leq l2 \} + \{ r2 @ l2 \} + \Rightarrow \{ r1 @ l1 \}$
 - If $r1$ “flows to” $r2$ and $l1$ “flows to” $l2$ and $r2$ and $l2$ are correlated, then so are $r1$ and $r2$
 - Note: $r \leq r$ and $l \leq l$

Constraint Resolution, Graphically



Consistent Correlation

- Next define the *correlation set* of a location
 - $S(R) = \{ L \mid R @ L \}$
 - The correlation set of R is the set of locks L that are correlated with it after applying all the rewrite rules
 - Notice that both of these are constants
- Consistent correlation: for every R , $|S(R)| = 1$
 - Means location only ever accessed with one lock

Example

let k1 = newlock ^{L1} in	// k1 : lock m, $L1 \leq m$
let k2 = newlock ^{L2} in	// k2 : lock n, $L2 \leq n$
let x = ref ^{Rx} 0 in	// x : ref ^a (int), $Rx \leq a$
let y = ref ^{Ry} 1 in	// y : ref ^b (int), $Ry \leq b$
x := ^{k1} 3;	// a @ m
x := ^{k1} 4;	// a @ m
y := ^{k1} 5;	// b @ m
y := ^{k2} 6	// b @ n

- Applying last constraint resolution rule yields
 - $\{ Rx @ L1 \} + \{ Rx @ L1 \} + \{ Ry @ L1 \} + \{ Ry @ L2 \}$
 - Inconsistent correlation for Ry

Consequences of May Alias Analysis

- We used may aliasing for locations and locks
 - One of these is okay, and the other is not

May Aliasing of Locations

```
let k1 = newlockL  
let x = refRx 0  
let y = refRy 0  
let z = if 0 42 then x else y  
      z :=k1 3
```

- Constraint solving yields $\{ Rx @ L \} + \{ Ry @ L \}$
- Thus any two locations that may alias must be protected by the same lock
- This seems fairly reasonable, and it is sound

May Aliasing of Locks

```
let k1 = newlockL1  
let k2 = newlockL2  
let k = if0 42 then k1 else k2  
let x = refRx 0  
  x :=k 3; x :=k1 4
```

- { Rx @ L1 } + { Rx @ L2 } + { Rx @ L1 }
- Thus Rx is inconsistently correlated
- That's not so bad — we're just rejecting an odd program

May Aliasing of Locks (cont'd)

```
let k1 = newlockL  
let k2 = newlockL      // fine according to rules  
let k = if0 42 then k1 else k2  
let x = refRx 0  
      x :=k 3; x :=k1 4
```

- { Rx @ L } + { Rx @ L } + { Rx @ L }
- Uh-oh! Rx is consistently correlated, but there's a potential "race"
 - Note that k and k1 are different locks at run time
- Allocating a lock in a loop yields same problem

The Need for Must Information

- The problem was that we need to know exactly what lock was "held" at the assignment
 - It's no good to know that some lock in a set was held, because then we don't know anything
 - We need to ensure that the same lock is *always* held on access
- We need *must alias* analysis for locks
 - Static analysis needs to know exactly which run-time lock is represented by each static lock label

Must Aliasing via Linearity

- Must aliasing not as well-studied as may
 - Many early alias analysis papers mention it
 - Later ones focus on may alias
 - Recall this is really used for “must not”
- One popular technique: linearity
 - We want each static lock label to stand for exactly one run-time location
 - I.e., we want lock labels to be *linear*
 - Term comes from linear logic
 - “Linear” in our context is a little different

Enforcing Linearity

- Consider the bad example again
 - `let k1 = newlockL`
 - `let k2 = newlockL`
 - Need to prevent lock labels from being reused
- Solution: remember `newlockd` labels
 - And prevent another `newlock` with the same label
 - We can do this by adding *effects* to our type system

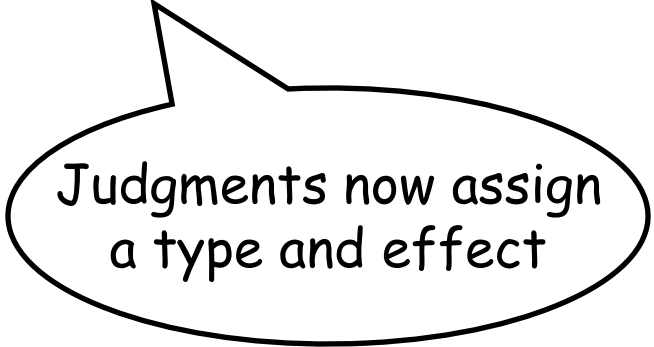
Effects

- An *effect* captures some stateful property
 - Typically, which memory has been read or written
 - We'll use these kinds of effects soon
 - In this case, track what locks have been created

$f ::= 0$	no effect
eff	effect variable
$\{l\}$	lock l was allocated
$f + f$	union of effects
$f \oplus f$	<i>disjoint union of effects</i>

Type Rules with Effects

$$\frac{L \leq m \quad m \text{ fresh}}{A \vdash\text{--} \text{newlock}^L : \text{lock } m; \{m\}}$$



Judgments now assign
a type and effect

Type Rules with Effects (cont'd)

$$\frac{}{A \dashv\vdash x : A(x); 0}$$

$$\frac{A \dashv\vdash e1 : \text{ref}^r t; f1 \quad A \dashv\vdash e2 : t; f2}{A \dashv\vdash e1 := e2 : t; f1 \oplus f2}$$

Prevents >1 alloc

$$\frac{A \dashv\vdash e1 : \text{int}; f1 \quad A \dashv\vdash e2 : t; f2 \quad A \dashv\vdash e3 : t; f3}{A \dashv\vdash \text{if0 } e1 \text{ then } e2 \text{ else } e3 : t; f1 \oplus (f2 + f3)}$$

Only one branch taken

Rule for Functions

- Is the following rule correct?

$$\frac{A, x:t \vdash e : t'; f}{A \vdash \lambda x:t. e : t \rightarrow t'; f}$$

- No!
- The fn's effect doesn't occur when it's defined
 - It occurs when the function is called
- So we need to remember the effect of a function

Correct Rule for Functions

- Extend types to have effects on arrows

$t ::= \text{int} \mid t \rightarrow^f t \mid \text{lock} \mid \text{ref}^r t$

$$\frac{A, x:t \vdash e : t'; f}{A \vdash \lambda x:t. e : t \rightarrow^f t'; 0}$$

$$\frac{A \vdash e1 : t \rightarrow^f t'; f1 \quad A \vdash e2 : t; f2}{A \vdash e1 e2 : t'; f1 \oplus f2 \oplus f}$$

One Minor Catch

- What if two function types need to be equal?
 - Can use subsumption rule

$$\frac{A \vdash e : t; f \quad t \leq t' \quad f \leq \text{eff}}{A \vdash e : t'; \text{eff}}$$

- We always use a variable as an upper bound
- Otherwise how would we solve constraints like
 - $\{L1\} + \{L2\} + f \leq \{L1\} + g + h$?

Safe to assume
have more effects

Another Minor Catch

- We don't have types with effects on them

Standard type

$$\frac{A, x:s \vdash e : t'; f \quad t = \text{fresh}(s)}{A \vdash \lambda x:s. e : t \rightarrow^f t'; 0}$$

Fresh label variables and
effect variables

Effect Constraints

- The same old story!
 - Walk over the program
 - Generate constraints
 - $r1 \leq r2$
 - $l1 \leq l2$
 - $f \leq \text{eff}$
 - Effects include disjoint unions
 - Solution \Rightarrow locks can be treated linearly
 - No solution \Rightarrow reject program

Effect Constraint Resolution

- Step 1: Close lock constraints
 - $S + \{ l1 \leq l2 \} + \{ l2 \leq l3 \} + \Rightarrow \{ l1 \leq l3 \}$
- Step 2: Count!
 - $\text{occurs}(l, 0) = 0$
 - $\text{occurs}(l, \{l\}) = 1$
 - $\text{occurs}(l, \{l'\}) = 0 \quad l \neq l'$
 - $\text{occurs}(l, f1 \oplus f2) = \text{occurs}(l, f1) + \text{occurs}(l, f2)$
 - $\text{occurs}(l, f1 + f2) = \max(\text{occurs}(l, f1), \text{occurs}(l, f2))$
 - $\text{occurs}(l, \text{eff}) = \max \text{occurs}(l, f) \text{ for } f \leq \text{eff}$
 - For each effect f and for every lock l , make sure that $\text{occurs}(l, f) \leq 1$

Example

```
let k1 = newlockL  
let k2 = newlockL      // violates disjoint union  
let k = if0 42 then k1 else k2  // k1, k2 have same type  
let x = refRx 0  
    x :=k 3; x :=k1 4
```

- Example is now forbidden
- Still not quite enough, though, as we'll see...

Applying this in Practice

- That's the core system
 - But need a bit more to handle those cases we saw way back at the beginning of lecture
- In C,
 1. We need to deal with C
 2. Held locks are not given by the programmer
 - Locks can be acquired or released anywhere
 - More than one lock can be held at a time
 3. Functions can be polymorphic in the relationship between locks and locations
 4. Much data is thread-local

Computing Held Locks

- Create a control-flow graph of the program
 - We'll be constraint-based, for fun!
 - A program point represented by state variable **S**
 - State variables will have *kinds* to tell us what happened in the state (e.g., lock acquire, deref)
- Propagate information through the graph using dataflow analysis

Computing Held Locks by Example

```
pthread_mutex_t k1 = ...; // k1: lock L1
```

```
int x; // &x : refRx int
```

```
// l: lock l, p : refRp (refa int)
```

```
void munge(pthread_mutex_t *l, int *p) {
```

```
    pthread_mutex_lock(l);
```

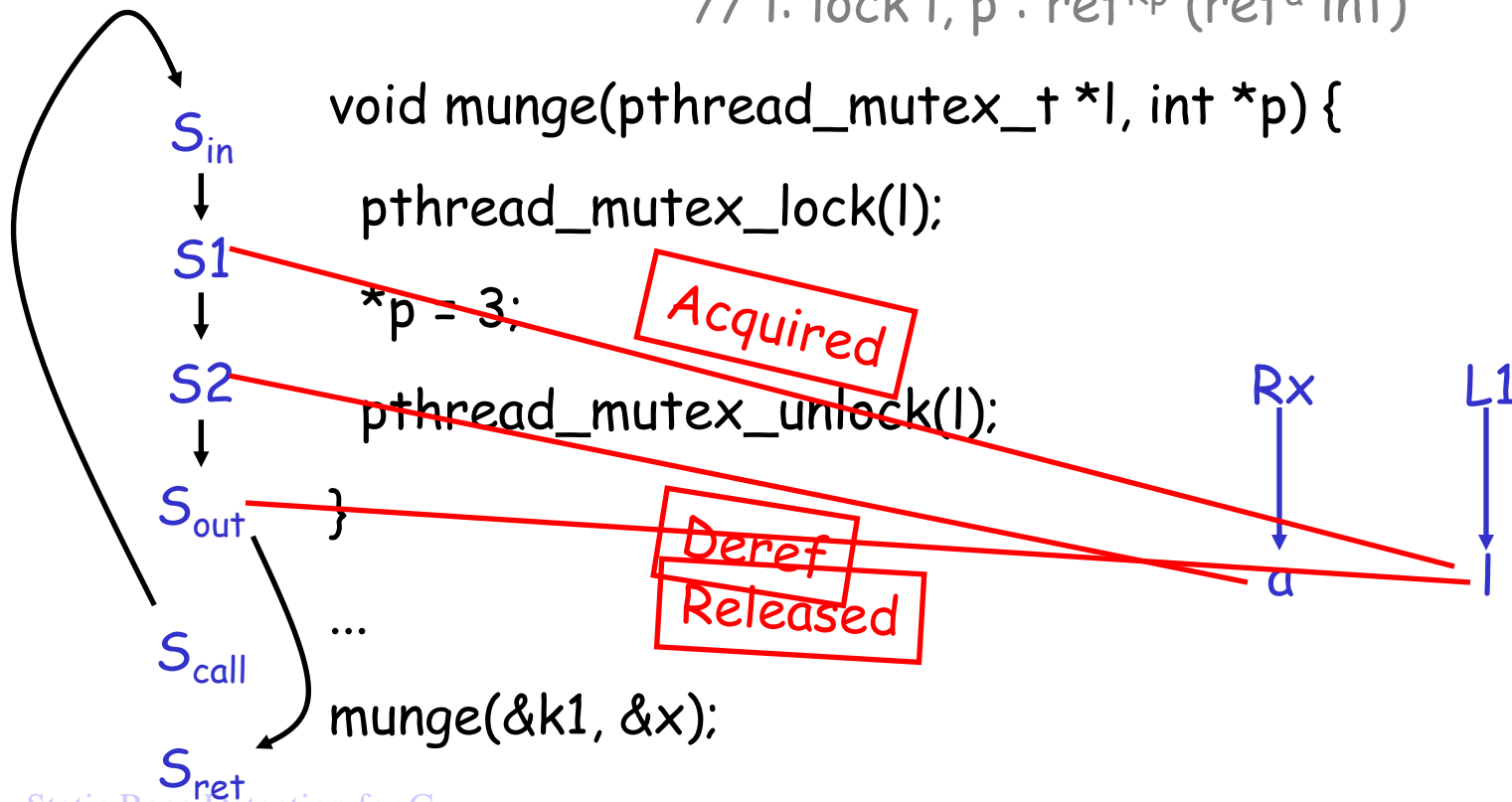
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

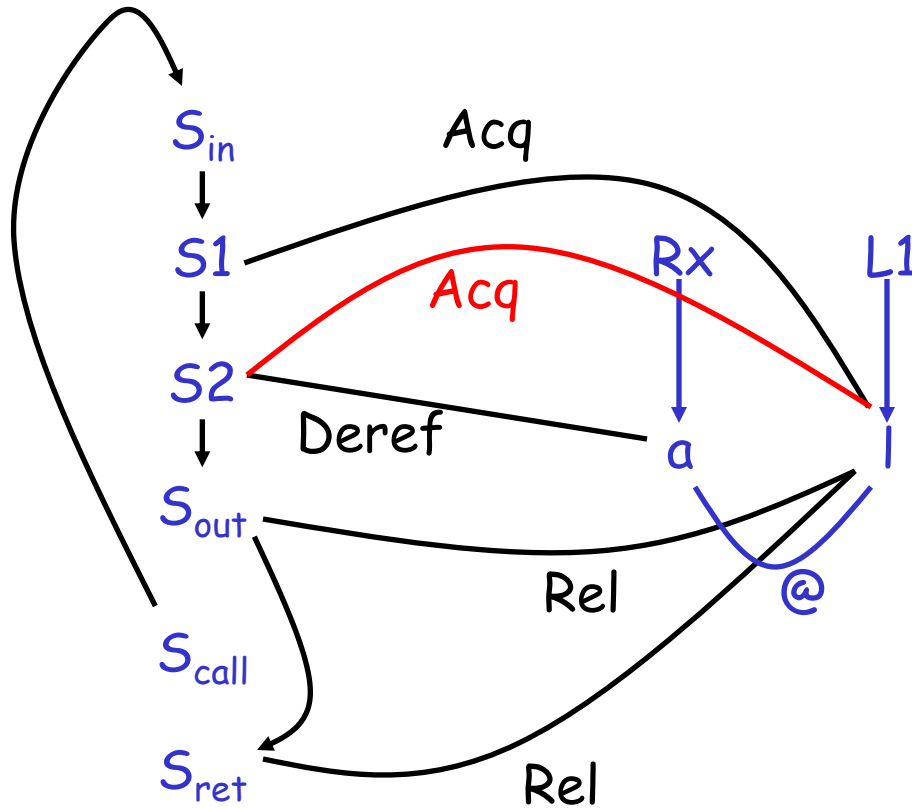
```
}
```

```
...
```

```
munge(&k1, &x);
```



Solving Constraints



More than One Lock May Be Held

- We can acquire multiple locks at once
`pthread_mutex_lock(&k1);`
`pthread_mutex_lock(&k2);`
`*p = 3;...`
- This is easy — just allow sets of locks, right?
 - Constraints $r @ \{l_1, \dots, l_n\}$
 - Correlation set $S(R) = \{ \{l_1, \dots, l_n\} \mid r @ \{l_1, \dots, l_n\} \}$
 - Consistent correlation: for every R , $|\cap S(R)| \geq 1$

Back to Linearity

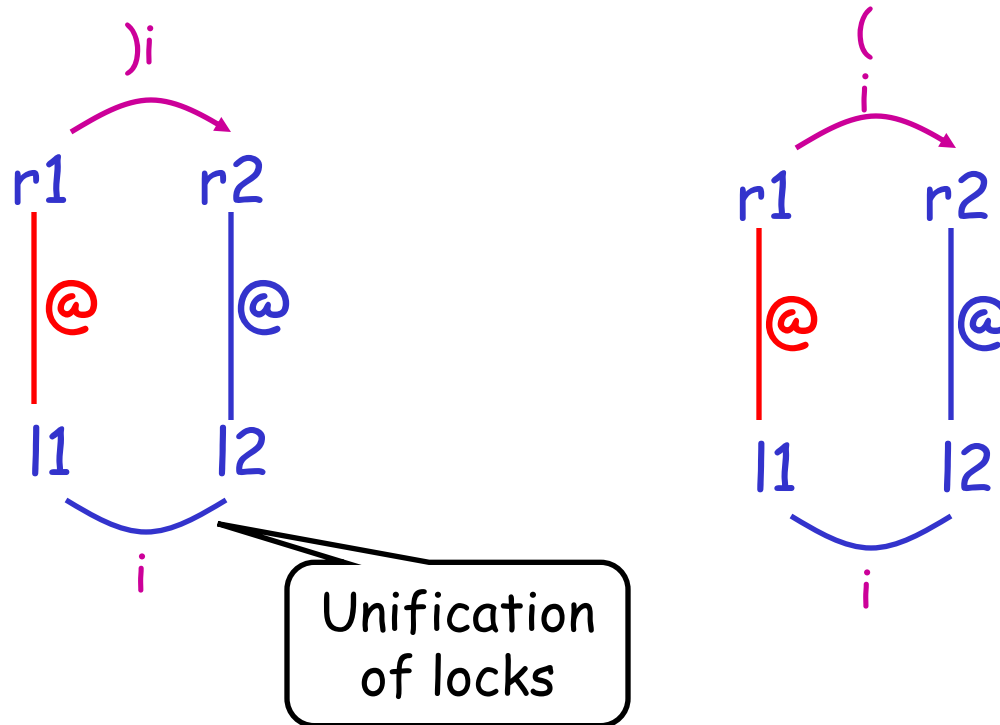
- How do we distinguish previous case from
let k = if0 42 then k1 else k2
pthread_mutex_lock(&k)
*p = 3;...
 - Can't just say p correlated with {k1, k2}
 - Some lock is acquired, but don't know which

Solutions (Pick One)

- Acquiring a lock l representing more than one concrete lock L is a no-op
 - We're only interested in races, so okay to forget that we've acquired a lock
- Get rid of subtyping on locks
 - Interpret \leq as unification on locks
 - Unifying two disjoint locks not allowed
 - Disjoint unions prevent same lock from being allocated twice
 - \Rightarrow Can never mix different locks together

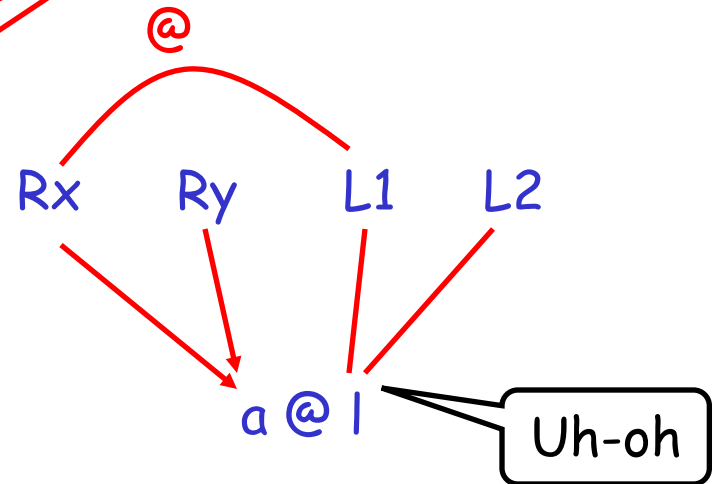
Context-Sensitive Correlation Inference

- Previous propagation rules, but match ()'s



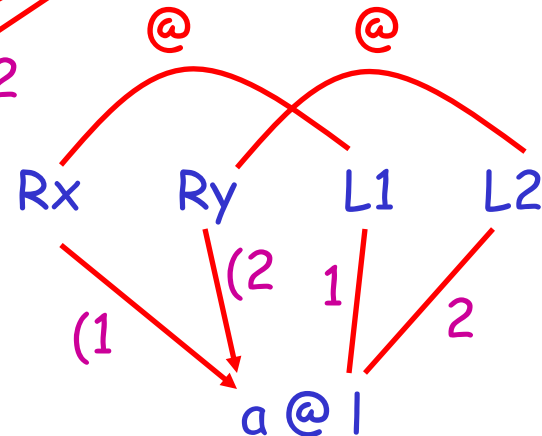
Example

```
pthread_mutex_t k1L1 = ..., k2L2 = ...;  
int xRx, yRy;  
void munge(pthread_mutex_tl *l, inta *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
munge(&k1, &x);  
munge(&k2, &y);
```



Example: Using Context-Sensitivity

```
pthread_mutex_t k1L1 = ..., k2L2 = ...;  
int xRx, yRy;  
void munge(pthread_mutex_tl *l, inta *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
munge1(&k1, &x);  
munge2(&k2, &y);
```



Sharing Inference

Thread-Local Data

- Even in multi-threaded programs, lots of data is thread local
 - No need to worry about synchronization
 - A good design principle
- We've assumed so far that everything is shared
 - Much too conservative

Sharing Inference

- Use alias analysis to find shared locations
- Basic idea:
 - Determine what locations each thread may access
 - Hm, looks like an effect system...
 - Shared locations are those accessed by more than one thread
 - Intersect effects of each thread
 - Don't forget to include the parent thread

Initialization

- A common pattern:

```
struct foo *p = malloc(...);
```

```
// initialize *p
```

```
fork(<something with p>); // p becomes shared
```

```
// parent no longer uses p
```

- If we compute

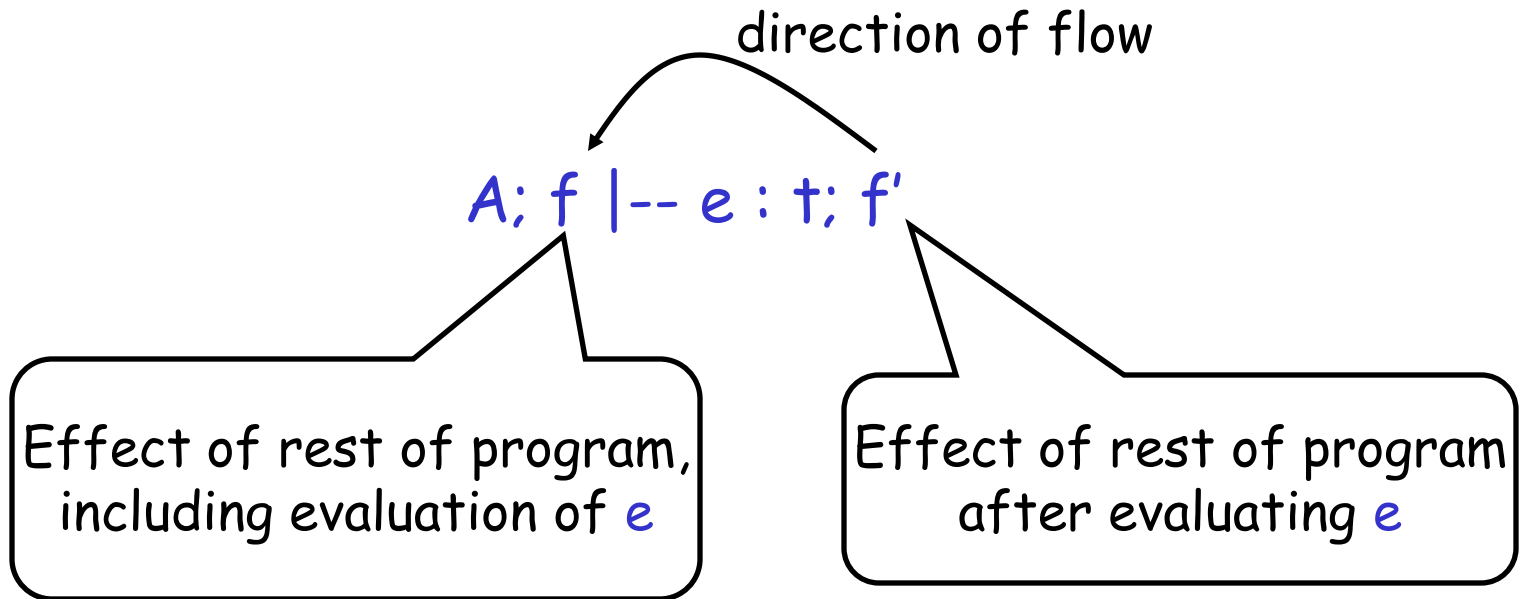
$\langle \text{effects of parent} \rangle \cap \langle \text{effects of child} \rangle$

then we'll see **p** in both, and decide it's shared

Continuation Effects

- Continuation effects capture the effect of the remainder of the computation
 - I.e., of the continuation
 - So in our previous example, we would see that in the parent's continuation after the fork, there are no effects
- Effects on locations
 - $f ::= 0 \mid \{r\} \mid \text{eff} \mid f + f$
 - Empty, locations, variables, union

Judgments



Type Rules

No change from
before to after

$A; f \vdash\!\!\vdash x : t; A(x); f$

Left-to-right order
of evaluation

$A; f \vdash\!\!\vdash e1 : \text{ref}^r t; f1 \quad A; f1 \vdash\!\!\vdash e2 : t; f2$

$\{ r \} \leq f2$

Memory write
happens after
e1 and e2
evaluated

$A; f \vdash\!\!\vdash e1 := e2 : t; f2$

Rule for Fork

$$\frac{A; f_{\text{child}}^i \dashv\vdash e : t; f' \quad f_{\text{child}}^i \leq f \quad f_{\text{parent}}^i \leq f}{A; f \dashv\vdash \text{fork}^i e : \text{int}; f_{\text{parent}}^i}$$

Child's effect
included in parent

Label each fork

Include everything
after the fork
in the parent

Computing Sharing

- Resolve effect constraints
 - Same old constraint propagation
 - Let $S(f)$ = set of locations in effect f
- Then the shared locations at fork^i are
 - $S_i = S(f_{\text{child}}^i) \cap S(f_{\text{parent}}^i)$
- And all the shared locations are
 - $\text{shared} = \cup_i S_i$

Including Child's Effect in Parent

- Consider:
 `let x = refRx 0 in`
 `fork1 (!x);`
 `fork2 (x:=2);`
- Then if we didn't include child's effects in parent, we wouldn't see that parallel child threads share data

Race Detection, Results

Trylock

- In most cases, just syntactically recognize
`if (trylock(&l)) { ... } else { ... }`
- Recall that lock states are flow-sensitive
 - So just assume `l` acquired in true branch, and unchanged in false branch
- Can get slightly fancier if result of `trylock` stored in an integer
 - ...which is what CIL will transform the program to

Locks in Data Structures

- Alias analysis conflates nodes of data structs
 - Locks in data structures not likely to be linear
- Our solution: *existential quantification*

```
∃l,r [ r @ l ]. struct job {  
    lock_tl j_lock;  
    struct job *next;  
    unsignedr cnt;  
};
```

Locks in Data Structures

- Must “unpack” existential type to use it

```
for (j = joblist; j != NULL; j = j->next) {  
    unpack (j){                      // (different syntax in practice)  
        lock(&j->j_lock);  
        j->cnt++;  
        unlock(&j->j_lock);  
    }  
}
```
- Restrictions on unpacking
 - Can only **unpack** one node at a time
 - Nothing unpacked may escape the **unpack** scope
 - ==> Only working with that one node
 - ==> Safe to assume lock is linear

void* and Aggregates

Current Results

Benchmark	Size (LOC)	Time (sec)	Warnings	Unguarded	Races
aget	1,914	0.85	62	31	31
ctrace	2,212	0.59	10	9	2
engine	2,608	0.88	7	0	0
knot	1,985	0.78	12	8	8
pfscan	1,948	0.46	6	0	0
smtprc	8,624	5.37	46	1	1
3c501	17,443	9.18	15	5	4
eql	16,568	21.38	35	0	0
hp100	20,370	143.23	14	9	8
plip	19,141	19.14	42	11	11
sis900	20,428	71.03	6	0	0
slip	22,693	16.99	3	0	0
sundance	19,951	106.79	5	1	1
synclink	24,691	1521.07	139	2	0
wavelan	20,099	19.70	10	1	1

Conclusion

- Alias analysis is a key building block
 - Lots and lots of stuff is variations on it
- We can perform race detection on *C* code
 - Bring out the toolkit of constraint-based analysis
 - Scales somewhat, still needs improvement
 - Handles idioms common to *C*
 - Including some things we didn't have time for