Static Race Detection for C

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

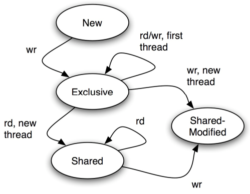
- The most common technique:
  - Locations r
  - Locks l
  - Correlation: r @ l
    - Location r is accessed when l is held
  - *Consistent correlation*
    - Any shared location is only ever correlated with one lock
      - We say that that lock *guards* that location
    - Implies race freedom
- Not 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
  - Locks_held(t) = set of locks held by thread t
  - For each r, set C(r) := { all locks }
  - On each access to r by thread t,
    - $C(r) := C(r) \cap \text{locks\_held}(t)$
    - If $C(r) = 0$, 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 synchornization
- ==> 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

Oops — We Can't Do This!

- Rice's Theorem: No computer program can precisely determine anything interesting about arbitrary source code
  - Does this program terminate?
  - Does this program produce value 42?
  - Does this program raise an exception?
  - Is this program correct?

The Art of Static Analysis

- Programmers don't write arbitrarily complicated programs
- Programmers have ways to control complexity
  - Otherwise they couldn't make sense of them
- Target: Be precise for the programs that programmers want to write
  - It's OK to forbid yucky code in the name of safety

Outline

- C locking idioms
- Alias analysis
  - An overview
  - Alias analysis via type systems
- Extend to infer correlations
- Making it work in practice for C
- Context-sensitivity via CFL reachability
- 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 thead-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->a, q->a), (p->b, q->b), ...
  - What about cyclic data structures?
    - (*p, p->next), (*p, p->next->next), ...

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:	Flow-insensitive:
p = &x; // (p, {&x})	(p, {&x, &y})
p = &y; // (p, {&y})	(x, &z)
*p = &z; // (p, {&y}), (y, {&z})	(y, &z)

A Simple Language

- We'll develop an alias analysis for ML
  - We'll talk about applying this to C later on

e ::= x	variables
| n	integers
| \x:t.e	functions
| e e	application
| if0 e then e else e	conditional
| let x = e in e	binding
| ref e	allocation
| !e	derefernce
| e := e	assignment

Aliasing in this Language

- ref creates an updatable reference
  - It's like malloc followed by initialization
- That pointer can be passed around the program

```
let x = ref 0 in
  let y = x in
    y := 3;   // updates !x
```

Label Flow for Points-to Analysis

- We're going to extend references with labels
  - e ::= ... | ref^{r} e | ...
  - Here r labels this particular memory allocation
    - Like malloc@257, identifies a line in the program
    - Drawn from a finite set of labels R
  - For now, programmers add these
- Goal of points-to analysis: determine set of labels a pointer may refer to

```
let x = ref^Rx 0 in
  let y = x in
    y := 3;  // y may point to { Rx }
```

Type-Based Alias Analysis

- We're going to build an alias analysis out of type inference
  - If you're familiar with ML type inference, that what we're going to do
- We'll use *labeled types* in our analysis
  - t ::= int | t → t | ref^{r} t
  - If we have !x or x := ..., we can decide what location x may point to by looking at its ref type

A Type Checking System

$$\frac{}{\text{A |-- x : A(x)}} \qquad \frac{}{\text{A |-- n : int}}$$

$$\frac{\text{A, x:t |-- e : t'}}{\text{A |-- \\x:t.e : t} \rightarrow \text{t'}} \qquad \frac{\text{A |-- e1 : t} \rightarrow \text{t'} \quad \text{A |-- e2 : t}}{\text{A |-- e1 e2 : t'}}$$

$$\frac{\text{A |-- e1 : int} \quad \text{A |-- e2 : t} \quad \text{A |-- e3 : t}}{\text{A |-- if0 e1 then e2 else e3 : t}}$$

A Type Checking System (cont'd)

$$\frac{\text{A |-- e : t}}{\text{A |-- ref}^{r}\text{ e : ref}^{r}\text{ t}}$$

$$\frac{\text{A |-- e : ref}^{r}\text{ t}}{\text{A |-- !e : t}}$$

$$\frac{\text{A |-- e1 : ref}^{r}\text{ t} \quad \text{A |-- e2 : t}}{\text{A |-- e1 := e2 : t}}$$

Example

```
let x = ref^Rx 0 in
  let y = x in
    y := 3;
```

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

Another Example

```
let x = ref^R 0 in
let y = ref^R 0 in
let w = ref^Rw 0 in
let z = if0 42 then x else y in
  z := 3;
```

- x and y both have type ref^{R} int
  - They must have this type because they are conflated by if
- At assignment, we write to location R
  - Notice that we don't know which of x, y we write to
  - But we do know that we don't affect w

Yet Another Example

let x = ref^{R} 3
let y = ref^{Ry} x
let z = ref^{R} 4
  y := z

- Both x and z have the same label
- y has type ref^{Ry} (ref^{R} int)
    - Notice we don't know after the assignment whether y points to x or z

Things to Notice

- We have a finite set of labels
    - One for each occurrence of ref in the program
    - A label may stand for more than one run-time loc
- Whenever two labels "meet" in the type system, they must be the same
    - Where does this happen in the rules?
- The system is flow-insensitive
    - Types don't change after assignment

The Need for Type Inference

- In practice, we don't have labeled programs
    - We need *inference*
- Given an unlabeled program that satisfies a standard type system, does there exist a valid labeling?
    - That labeling is our alias analysis

Type Checking vs. Type Inference

- Let's think about C's type system
    - C requires programmers to annotate function types
    - …but not other places
        - E.g., when you write down 3 + 4, you don't need to give that a type
    - So all type systems trade off programmer annotations vs. computed information
- Type checking = it's "obvious" how to check
- Type inference = it's "more work" to check

A Type Inference Algorithm

- We'll follow the standard approach
    - Introduce *label variables* a, which stand for unknowns
        - Now r may be either a constant R or a variable a
- Traverse the code of the unlabeled program
- Generate a set of *constraints*
- Solve the constraints to find a labeling
    - No solution ==> no valid labeling

Step 1: Introducing Labels

- Problem 1: In the ref rule, we don't know what label to assign to the ref
    - Solution: Introduce a fresh unknown
        - Why do we need to pick a variable rather than a constant?

$$\frac{A \vdash e : t \qquad a \text{ fresh}}{A \vdash \text{ref}^{a}\ e : \text{ref}^{a}\ t}$$

Step 1: Introducing Labels (cont'd)

- Problem 2: In the function rule, we don't know what type to give to the argument
  - Assume we are given a standard type s (no labels)
  - Make up a new type with fresh labels everywhere
    - We'll write this as fresh(s)

$$\frac{A,\ x{:}t\ |\text{--}\ e : t' \qquad t = \text{fresh}(s)}{A\ |\text{--}\ \backslash x{:}s.e : t \rightarrow t'}$$

Step 2: Adding Constraints

- Problem 3: Some rules implicitly require types to be equal
  - We will make this explicit with *equality constraints*

$$\frac{A\ |\text{--}\ e1 : \text{int} \quad A\ |\text{--}\ e2 : t2 \quad A\ |\text{--}\ e3 : t3 \\ t2 = t3}{A\ |\text{--}\ \text{if0 } e1 \text{ then } e2 \text{ else } e3 : t2}$$

Step 2: Adding Constraints (cont'd)

$$\frac{A\ |\text{--}\ e1 : \text{ref}^{r}\ t \quad A\ |\text{--}\ e2 : t2 \\ t = t2}{A\ |\text{--}\ e1 := e2 : t}$$

- Notice we're assuming that e1 is a ref
  - That was part of our assumption — we assumed the program was safe according to the standard types

Step 2: Adding Constraints (cont'd)

$$\frac{A\ |\text{--}\ e1 : t \rightarrow t' \quad A\ |\text{--}\ e2 : t2 \qquad t = t2}{A\ |\text{--}\ e1\ e2 : t'}$$

- Again, we're assuming e1 is a function

Constraint Resolution

- After applying the rules, we are left with a set of equality constraints
  - t1 = t2
- We'll solve the constraints via rewriting
  - We'll simplify more complex constraints into simpler constraints
  - S ==> S' rewrite constraints S to constraints S'

Constraint Resolution via Unification

- S + { int = int } ==> S
- S + { $t1 \rightarrow t2 = t1' \rightarrow t2'$ } ==>
  S + { t1' = t1 } + { t2 = t2' }
- S + { $\text{ref}^{a1}\ t1 = \text{ref}^{a2}\ t2$ } ==>
  S + { t1 = t2 } + { a1 = a2 }
- S + { mismatched constructors } ==> error
  - Can't happen if program correct w.r.t. std types
- Claim 1: This algorithm always terminates
- Claim 2: When it terminates, we are left with equalities among labels

Constraint Resolution via Unification (cont'd)

- Last step:
  - Computes sets of labels that are equal (e.g., using union-find)
  - Assign each equivalence class its own constant label

Example

let x = ref 0 in // x : ref^{a} int
let y = ref 0 in // y : ref^{b} int
let w = ref 0 in // w : ref^{c} int
let z = if0 42 then x else y in // z : ref^{a}, ref^{a} = ref^{b}
z := 3; // write to ref^{a}

- Solving constraint ref^{a} = ref^{b} yields a = b
- So we have two equivalence classes
  - {a,b} and {c}
  - Each one gets a label, e.g., R1 and R2

Example

let x = ref 0 in // x : ref^{R1} int
let y = ref 0 in // y : ref^{R1} int
let w = ref 0 in // w : ref^{R2} int
let z = if0 42 then x else y in // z : ref^{R1}
z := 3; // write to ref^{R1}

- Solving constraint ref^{a} = ref^{b} yields a = b
- So we have two equivalence classes
  - {a,b} and {c}
  - Each one gets a label, e.g., R1 and R2

Steensgaard's Analysis

- Flow-insensitive
- Context-insensitive
- Unification-based
  - = Steensgaard's Analysis
  - (In practice, Steensgaard's analysis includes stuff for type casts, etc)

- Properties
  - Very scalable
    - Complexity?
  - Somewhat imprecise

Limitation of Unification

- Modification of previous example:

let x = ref 0 in // x : ref^{R1} int
let y = ref 0 in // y : ref^{R1} int
let z = if0 42 then x else y in // z : ref^{R1}
z := 3; // write to ref^{R1}
x := 2; // write to ref^{R1}

- We're equating labels that may alias
  - Gives "backward flow" -- the fact that x and y are merged "downstream" (in z) causes x and y to be equivalent everywhere

---

Subtyping

- We can solve this problem using *subtyping*
  - Each label variable now stands for a *set* of labels
    - In unification, a variable could only stand for one label
  - We'll write [a] for the set represented by a
    - And [R] = {R} for a constant R

- Ex: let x have type ref^{a} int
  - Suppose [a] = {R1, R2}
  - Then x may point to location R1 or R2
  - ...and R1 and R2 may themselves stand for multiple locations

Labels on ref

- Slightly different approach to labeling
  - Assume that each ref has a unique constant label
    - Generate a fresh one for each syntactic occurrence
  - Add a fresh variable, and generate a *subtyping constraint* between the constant and variable
    - a1 ≤ a2 means [a1] ⊆ [a2]

$$\frac{A\ |\text{--}\ e : t \quad R \le a \quad a \text{ fresh}}{A\ |\text{--}\ \text{ref}^{R}\ e : \text{ref}^{a}\ t}$$

Subtype Inference

- Same basic approach as before
  - Walk over source code, generate constraints
  - Now want to allow subsets rather than equalities

$$\frac{A\ |\text{--}\ e1 : \text{int} \quad A\ |\text{--}\ e2 : \text{ref}^{r2}\ t \quad A\ |\text{--}\ e3 : \text{ref}^{r3}\ t \quad r2 \le r \quad r3 \le r}{A\ |\text{--}\ \text{if0 } e1 \text{ then } e2 \text{ else } e3 : \text{ref}^{r}\ t}$$

Subtyping Constraints

- Need to generalize to arbitrary types
  - Think of types as representing sets of values
    - E.g., int represents the set of integers
    - So ref^{r} int represents the set of pointers to integers that are labeled with [r]
  - Extend ≤ to a relation t ≤ t on types

$$\frac{}{\text{int} \le \text{int}} \qquad \frac{r1 \le r2 \quad \text{int} \le \text{int}}{\text{ref}^{r1}\ \text{int} \le \text{ref}^{r2}\ \text{int}}$$

Subsumption

- Add one new rule to the system
  - And leave remaining rules alone

$$\frac{A\ |\text{--}\ e : t \quad t \le t'}{A\ |\text{--}\ e : t'}$$

  - If we think that e has type t, and t is a subtype of t', then e also has type t'
  - We can use a subtype anywhere a supertype is expected

Example

let x = ref^{Rx} 0 in // x : ref^{a} int, Rx ≤ a
let y = ref^{Ry} 1 in // y : ref^{b} int, Ry ≤ b
let z = if 42 then x else y in
  x := 3

- At conditional, need types of x and y to match

$$\frac{A\ |\text{--}\ x : \text{ref}^{a}\ \text{int} \quad \dfrac{a \le c}{\text{ref}^{a}\ \text{int} \le \text{ref}^{c}\ \text{int}}}{A\ |\text{--}\ x : \text{ref}^{c}\ \text{int}}$$

- Thus we have z : ref^{c} int with a ≤ c and b ≤ c
  - Thus can pick a = {Rx}, b = {Ry}, c = {Rx, Ry}

Subtyping References (cont'd)

- Let's try generalizing to arbitrary types

$$\frac{r1 \le r2 \quad t1 \le t2}{\text{ref}^{r1}\ t1 \le \text{ref}^{r2}\ t2}$$

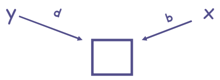
- This rule is broken

let x = ref^{Rx} ($\text{ref}^{Rx'}$ 0) in // x : ref^{a} (ref^{b} int), Rx' ≤ b
let y = x in // y : ref^{c} (ref^{d} int), b ≤ d
  y := ref^{Oops} 0 // Oops ≤ d
  !!x := 3 // dereference of b

- Can pick b = {Rx'}, d = {Rx', Oops}
  - Then write via b doesn't look like it's writing Oops

You've Got Aliasing!

- We have multiple names for the same memory location
  - But they have different types
  - *And* we can **write** into memory at different types



Solution #1: Java's Approach

- Java uses this subtyping rule
  - If S is a subclass of T, then S[] is a subclass of T[]
- Counterexample:
  - Foo[] a = new Foo[5];
  - Object[] b = a;
  - b[0] = new Object();
  - a[0].foo();
  - Write to b[0] forbidden at runtime, so last line cannot happen

Solution #2: Purely Static Approach

- Require equality "under" a ref

$$\frac{r1 \le r2 \quad t1 \le t2 \quad t2 \le t1}{ref^{r1}\ t1 \le ref^{r2}\ t2}$$

or

$$\frac{r1 \le r2 \quad t1 = t2}{ref^{r1}\ t1 \le ref^{r2}\ t2}$$

Subtyping on Function Types

- What about function types?

$$\frac{?}{t1 \to t2 \le t1' \to t2'}$$

- Recall: S is a subtype of T if an S can be used anywhere a T is expected
  - When can we replace a call "f x" with a call "g x"?

Replacing "f x" by "g x"

- When is $\underbrace{t1' \to t2'}_{g} \le \underbrace{t1 \to t2}_{f}$?
- Return type:
  - We are expecting t2 (f's return type)
  - So we can only return *at most* t2
  - $t2' \le t2$
- Example: A function that returns a pointer to {R1, R2} can be treated as a function that returns a pointer to {R1, R2, R3}

Replacing "f x" by "g x" (cont'd)

- When is $\underbrace{t1' \to t2'}_{g} \le \underbrace{t1 \to t2}_{f}$?
- Argument type:
  - We are supposed to accept t1 (f's argument type)
  - So we must accept *at least* t1
  - $t1 \le t1'$
- Example: A function that accepts a pointer to {R1, R2, R3} can be passed a pointer to {R1, R2}

Subtyping on Function Types

$$\frac{t1' \le t1 \quad t2 \le t2'}{t1 \to t2 \le t1' \to t2'}$$

- We say that → is
  - *Covariant* in the range (subtyping dir the same)
  - *Contravariant* in the domain (subtyping dir flips)

Where We Are

- We've built a unification-based alias analysis
- We've built a subtyping-based alias analysis
  - But it's still only a checking system
- Next steps
  - Turning this into inference
  - Adding context-sensitivity

The Problem: Subsumption

$$\frac{A \mid\text{--}\ e : t \quad t \le t'}{A \mid\text{--}\ e : t'}$$

- We're allowed to apply this rule at any time
  - Makes it hard to develop a deterministic algorithm
  - Type checking is not *syntax driven*
- Fortunately, we don't have that many choices
  - For each expression e, we need to decide
    - Do we apply the "regular" rule for e?
    - Or do we apply subsumption (how many times)?

Getting Rid of Subsumption

- Lemma: Multiple sequential uses of subsumption can be collapsed into a single use
  - Proof: Transitivity of ≤
- So now we need only apply subsumption once after each expression

---

Getting Rid of Subsumption (cont'd)

- We can get rid of the separate subsumption rule
  - Integrate into the rest of the rules

$$\frac{A \mid\text{--}\ e1 : t \to t' \quad A \mid\text{--}\ e2 : t2 \quad t = t2}{A \mid\text{--}\ e1\ e2 : t'}$$

becomes

$$\frac{A \mid\text{--}\ e1 : t \to t' \quad A \mid\text{--}\ e2 : t2 \quad t2 \le t}{A \mid\text{--}\ e1\ e2 : t'}$$

- Apply the same reasoning to the other rules
  - We're left with a purely syntax-directed system

Constraint Resolution: Step 1

- $S + \{ int \le int \} \Longrightarrow S$
- $S + \{ t1 \to t2 \le t1' \to t2' \} \Longrightarrow$
  $S + \{ t1' \le t1 \} + \{ t2 \le t2' \}$
- $S + \{ ref^{r1}\ t1 \le ref^{r2}\ t2 \} \Longrightarrow$
  $S + \{ t1 \le t2 \} + \{ t2 \le t1 \} + \{ r1 \le r2 \}$
- S + { mismatched constructors } ==> error

Constraint Resolution: Step 2

- Our type system is called a *structural subtyping system*
  - If $t \leq t'$, then t and t' have the same shape
- When we're done with step 1, we're left with constraints of the form $r1 \leq r2$
  - Where $r1$ and $r2$ are constants R or variables a
  - This is called an *atomic subtyping system*
  - That's because there's no "structure" left

Finding a Least Solution

- Our goal: compute a least solution to the remaining constraints
  - For each variable, compute a minimal set of constants satisfying the constraints
- One more rewriting rule: transitive closure
  - $S + \{ r1 \leq r2 \} + \{ r2 \leq r3 \}$ +==> $\{ r1 \leq r3 \}$
    - +==> means add rhs constraint without removing lhs constraints
  - Apply this rule until no new constraints generated
  - Then $[a] = \{ R \mid R \leq a$ is a constraint in $S \}$

Graph Reachability

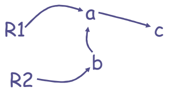
- Think of a constraint as a directed edge

$R1 \leq a$

$R2 \leq b$

$a \leq c$

$b \leq a$



  - Use graph reachability to compute solution
    - Compute set of constants that reach each variable
      - E.g., $[c] = [a] = \{R1, R2\}$, $[b] = \{R2\}$
    - Complexity?

Andersen's Analysis

- Flow-insensitive
- Context-insensitive
- Subtyping-based
  - = Andersen's analysis
  - ~= 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 | n | \x:t.e | e e | if0 e then e else e
  | newlock^L create a new lock
  | ref^R e allocate "shared" memory
  | $!^e$ e dereference with a lock held
  | e $:=^e$ e assign with a lock held
  t ::= int | t → t | lock l | 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 = newlock^{L1} in
let k2 = newlock^{L2} in
let x = ref^{Rx} 0 in
let y = ref^{Ry} 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 ==> program is consistently correlated
    - No solution ==> 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 labels for them
  - Standard labeling rule, standard subtyping
  - Warning: this is broken! Will fix later...

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

$$\frac{l1 \leq l2}{\text{lock } l1 \leq \text{lock } l2}$$

Correlation Constraints for Locations

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

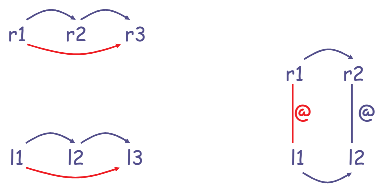
$$\frac{A \mid\text{--}\ e1 : \text{ref}^r\ t \quad A \mid\text{--}\ e2 : \text{lock } l \quad r @ l}{A \mid\text{--}\ !^{e2} e1 : t}$$

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

Constraint Resolution

- Apply subtyping until only atomic constraints
  - r1 ≤ r2 — location subtyping
  - l1 ≤ l2 — lock subtyping
  - r @ l — correlation
- Now apply three rewriting rules
  - S + { r1 ≤ r2 } + { r2 ≤ r3 } +==> { r1 ≤ r3 }
  - S + { l1 ≤ l2 } + { l2 ≤ l3 } +==> { l1 ≤ l3 }
  - S + { r1 ≤ r2 } + { l1 ≤ l2 } + { r2 @ l2 } +==> { 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 ≤ r and l ≤ l

Constraint Resolution, Graphically



Consistent Correlation

- Next define the *correlation set* of a location
  - S(R) = { L | 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 ≤ m
let k2 = newlock^{L2} in // k2 : lock n, L2 ≤ n
let x = ref^{Rx} 0 in // x : ref^{a}(int), Rx ≤ a
let y = ref^{Ry} 1 in // y : ref^{b}(int), Ry ≤ 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 = newlock^{L}
let x = ref^{Rx} 0
let y = ref^{Ry} 0
let z = if0 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 = newlock^{L1}
let k2 = newlock^{L2}
let k = if0 42 then k1 else k2
let x = ref^{Rx} 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 = newlock^{L}
let k2 = newlock^{L} // fine according to rules
let k = if0 42 then k1 else k2
let x = ref^{Rx} 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 = newlock^{L}
  let k2 = newlock^{L}
  - 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 creates

f ::= 0	no effect
| eff	effect variable
| {l}	lock l was allocated
| f + f	union of effects
| f ⊕ f	*disjoint union* of effects

Type Rules with Effects

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

Judgments now assign a type and effect

Type Rules with Effects (cont'd)

$$\frac{}{A \mid\!\text{--}\ x : A(x);\ 0}$$

$$\frac{A \mid\!\text{--}\ e1 : \text{ref}^r\ t;\ f1 \qquad A \mid\!\text{--}\ e2 : t;\ f2}{A \mid\!\text{--}\ e1 := e2 : t;\ f1 \oplus f2}$$

Prevents >1 alloc

$$\frac{A \mid\!\text{--}\ e1 : \text{int};\ f1 \qquad A \mid\!\text{--}\ e2 : t;\ f2 \qquad A \mid\!\text{--}\ e3 : t;\ f3}{A \mid\!\text{--}\ \text{if0 } e1 \text{ then } e2 \text{ else } e3 : t;\ f1 \oplus (f2 + f3)}$$

Only one branch taken

Rule for Functions

- Is the following rule correct?

$$\frac{A, x{:}t \mid\!\text{--}\ e : t';\ f}{A \mid\!\text{--}\ \backslash x{:}t.e : t \rightarrow t';\ f}$$

  - 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

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

$$\frac{A, x{:}t \mid\!\text{--}\ e : t';\ f}{A \mid\!\text{--}\ \backslash x{:}t.e : t \xrightarrow{f} t';\ 0}$$

$$\frac{A \mid\!\text{--}\ e1 : t \xrightarrow{f} t';\ f1 \qquad A \mid\!\text{--}\ e2 : t;\ f2}{A \mid\!\text{--}\ e1\ e2 : t';\ f1 \oplus f2 \oplus f}$$

One Minor Catch

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

$$\frac{A \mid\!\text{--}\ e : t;\ f \qquad t \leq t' \qquad f \leq \text{eff}}{A \mid\!\text{--}\ e : t';\ \text{eff}}$$

Safe to assume have more effects

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

Another Minor Catch

- We don't have types with effects on them

Standard type

$$\frac{A, x{:}s \mid\!\text{--}\ e : t';\ f \qquad t = \text{fresh}(s)}{A \mid\!\text{--}\ \backslash x{:}s.e : t \xrightarrow{f} t';\ 0}$$

Fresh label variables and effect variables

Effect Constraints

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

Effect Constraint Resolution

- Step 1: Close lock constraints
  - $S + \{ l1 \leq l2 \} + \{ l2 \leq l3 \}$ +==> $\{ l1 \leq l3 \}$
- Step 2: Count!

occurs(l, 0) = 0
occurs(l, {l}) = 1
occurs(l, {l'}) = 0   l != l'
occurs(l, f1 ⊕ f2) = occurs(l, f1) + occurs(l, f2)
occurs(l, f1 + f2) = max(occurs(l, f1), occurs(l, f2))
occurs(l, eff) = max occurs(l, f) for $f \leq eff$

  - For each effect f and for every lock l, make sure that occurs occurs(l, f) ≤ 1

Example

let k1 = newlock^{L}
let k2 = newlock^{L} // violates disjoint union
let k = if0 42 then k1 else k2   // k1, k2 have same type
let x = ref^{Rx} 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

Variables in C

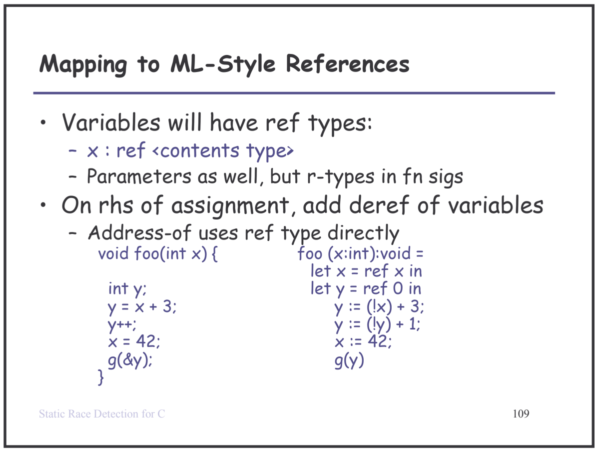
- The first (easiest) problem: C doesn't use ref
  - It has malloc for memory on the heap
  - But local variables on the stack are also updateable:

```
void foo(int x) {
  int y;
  y = x + 3;
  y++;
  x = 42;
}
```

- The C types aren't quite enough
  - 3 : int, but can't update 3!

L-Types and R-Types

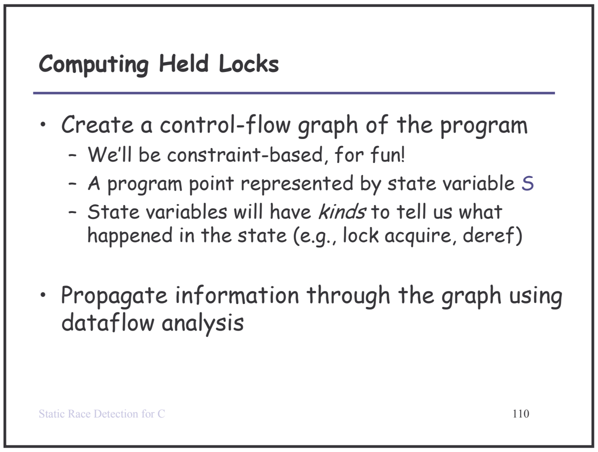
- C hides important information:
  - Variables behave different in l- and r-positions
    - l = left-hand-side of assignment, r = rhs
  - On lhs of assignment, x refers to *location* x
  - On rhs of assignment, x refers to *contents of location* x



Mapping to ML-Style References

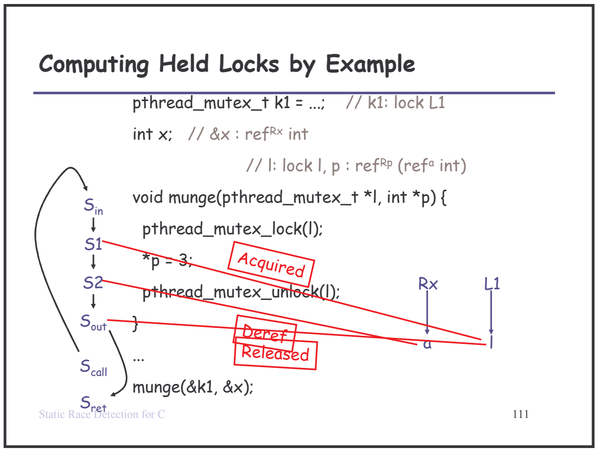
- Variables will have ref types:
  - x : ref <contents type>
  - Parameters as well, but r-types in fn sigs
- On rhs of assignment, add deref of variables
  - Address-of uses ref type directly

```
void foo(int x) {              foo (x:int):void =
                                 let x = ref x in
  int y;                         let y = ref 0 in
  y = x + 3;                         y := (!x) + 3;
  y++;                               y := (!y) + 1;
  x = 42;                            x := 42;
  g(&y);                             g(y)
}
```



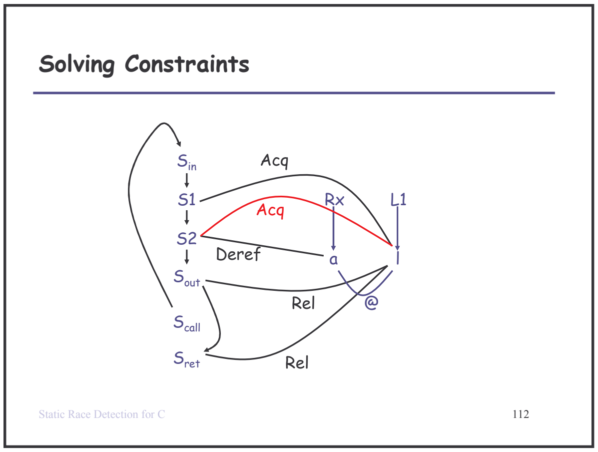
Computing Held Locks

- 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

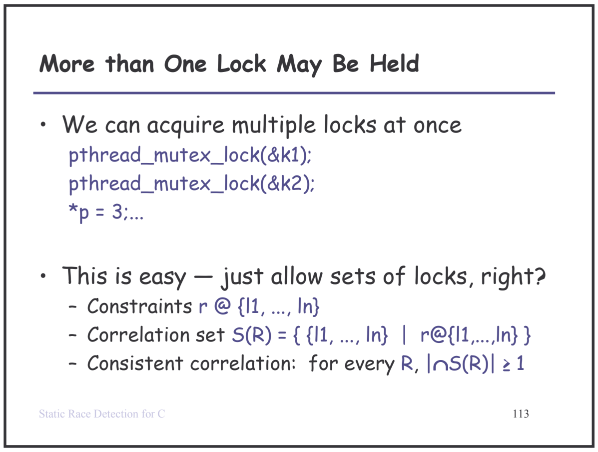


Computing Held Locks by Example

```
pthread_mutex_t k1 = ...;    // k1: lock L1
int x;   // &x : ref^Rx int
                    // l: lock l, p : ref^Rp (ref^a int)
void munge(pthread_mutex_t *l, int *p) {
  pthread_mutex_lock(l);
  *p = 3;
  pthread_mutex_unlock(l);
}
...
munge(&k1, &x);
```



Solving Constraints

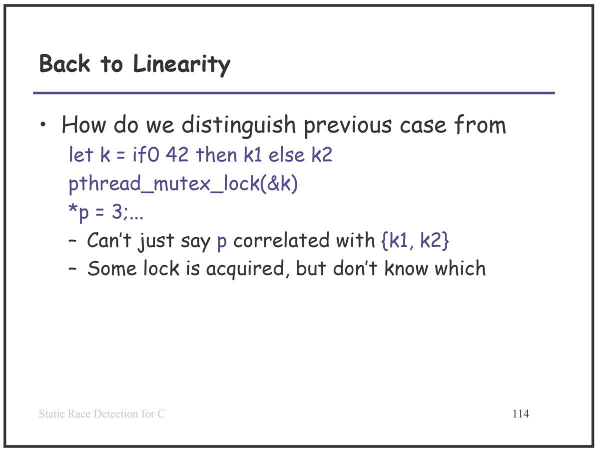


More than One Lock May Be Held

- 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 @ {l1, ..., ln}
  - Correlation set S(R) = { {l1, ..., ln} | r@{l1,...,ln} }
  - 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
  - ==> Can never mix different locks together

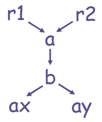
Context-Sensitivity

Limitations of Subtyping

- Subtyping gives us a kind of *polymorphism*
  - A *polymorphic* type represents multiple types
  - In a subtyping system, t represents t and all of t's subtypes
- As we saw, this flexibility helps make the analysis more precise
  - But it isn't always enough…

Limitations of Subtype Polymorphism

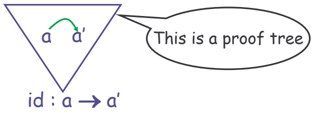
- Let's look at the identity function on int ptrs:
  - let id = \x:ref^a int . x
  - So id has type ref^a int → ref^b int
- Now consider the following:
  - let x = id (ref^{r1} 0)
  - let y = id (ref^{r2} 0)



  - It looks like ax and ay point to {r1, r2}
  - This is a *context-insensitive* analysis

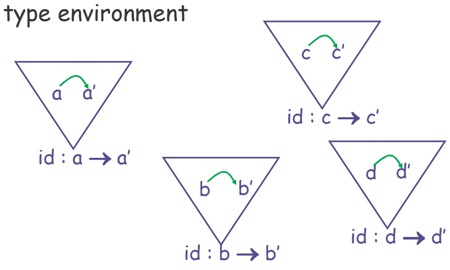
The Observation of Parametric Polymorphism

- Type inference on id yields a proof like this:



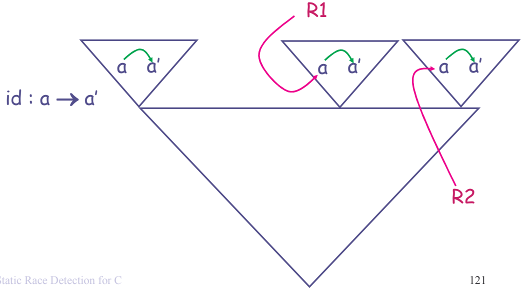
The Observation of Parametric Polymorphism

- We can duplicate this proof *for any* a,a', *in any* type environment



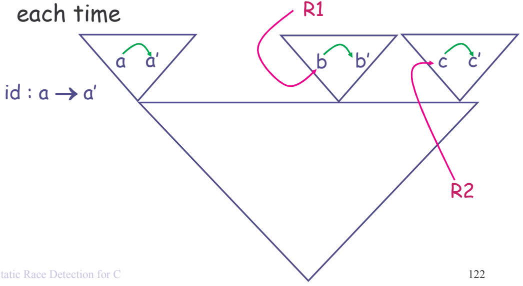
The Observation of Parametric Polymorphism

- Thus when we use id…



The Observation of Parametric Polymorphism

- We can "inline" its type, with a different a each time



Hindley-Milner Style Polymorphism

- Standard type rules (not quite for our system)
  - Generalize at let

$$\frac{A \vdash e1 : t1 \quad A, f : \forall \mathbf{a}.t1 \vdash e2 : t2 \quad \mathbf{a} = fv(t1) - fv(A)}{A \vdash \text{let } f = e1 \text{ in } e2 : t2}$$

- Instantiate at uses

Take the original type

$$\frac{A(f) = \forall \mathbf{a}.t1}{A \vdash f : t1[\mathbf{t}\backslash\mathbf{a}]}$$

Substitute bound vars (arbitrarily)

Polymorphically Constrained Types

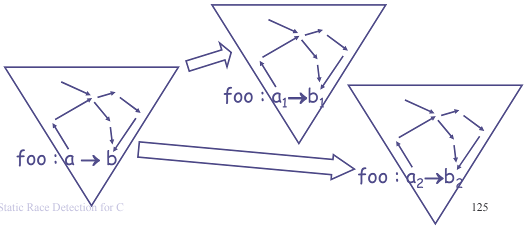
- Notice that we inlined not only the *type* (as in ML), but also the *constraints*
- We need polymorphically constrained types

$$x : \forall \mathbf{a}.t \text{ where } C$$

  - For any labels **a** where constraints C hold, x has type t

Polymorphically Constrainted Types

- Must copy constraints at each instantiation
  - Looks inefficient
  - (And hard to implement)



Comparison to Type Polymorphism

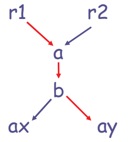
- ML-style polymorphic type inference is EXPTIME-hard
  - In practice, it's fine
  - Bad case can't happen here, because we're polymorphic *only* in the labels
    - That's because we'll apply this to C

A Better Solution: CFL Reachability

- Can reduce this to another problem
  - Equivalent to the constraint-copying formulation
  - Supports polymorphic recursion in qualifiers
  - It's easy to implement
  - It's efficient: $O(n^3)$
    - Previous best algorithm $O(n^8)$ [Mossin, PhD thesis]
- Idea due to Horwitz, Reps, and Sagiv [POPL'95], and Rehof, Fahndrich, and Das [POPL'01]

The Problem Restated: Unrealizable Paths

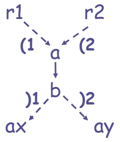
let id = \x:ref^a int . x
let x = id (ref^{r1} 0)
let y = id (ref^{r2} 0)



- No execution can exhibit that particular call/return sequence

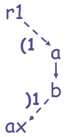
Only Propagate Along Realizable Paths

let id = \x:ref^a int . x
let x = id^1 (ref^{r1} 0)
let y = id^2 (ref^{r2} 0)



- Add edge labels for calls and returns
  - Only propagate along *valid* paths whose returns balance calls

Parenthesis Edges



- Paren edges represent substitutions
  - id : ∀a, b . a → b where a ≤ b
  - let x = id^1 (ref^{r1} 0)
- At call 1 to id, we instantiate type of id
  - $(a \to b)\underbrace{[r1\backslash a, ax\backslash b]}_{\text{Renaming for call 1}} = r1 \to ax$
- Edges with)1 or (1 represent renaming 1
  - $b \xrightarrow{)1} ax$ b instantiated to ax, and b flows to ax
  - $r1 \xrightarrow{(1} a$ a instantiated to r1, and r1 flows to a

Instantiation Constraints

- Edges with parentheses are called *instantiation constraints*
- They represent:
  - A renaming
  - Plus a "flow"
- We can extend instantiation constraints from labels to types in the standard way

Propagating Instantiation Constraints

- $S + \{ \text{int} \xrightarrow{)i} \text{int} \}$ ==> S
- $S + \{ \text{int} \xrightarrow{(i} \text{int} \}$ ==> S

- $S + \{ \text{ref}^{r1}\ t1 \xrightarrow{(i} \text{ref}^{r2}\ t2 \}$ ==>
  $S + \{ r1 \xrightarrow{(i} r2 \} + \{ t1 \xrightarrow{(i} t2 \} + \{ t2 \xrightarrow{)i} t1 \}$

- $S + \{ \text{ref}^{r1}\ t1 \xrightarrow{)i} \text{ref}^{r2}\ t2 \}$ ==>
  $S + \{ r1 \xrightarrow{)i} r2 \} + \{ t1 \xrightarrow{)i} t2 \} + \{ t2 \xrightarrow{(i} t1 \}$

Propagating Instantiation Constraints (cont'd)

- $S + \{ t1 \rightarrow t2 \dashrightarrow^{)i} t1' \rightarrow t2' \}$ ==>
  $S + \{ t2 \dashrightarrow^{)i} t2' \} + \{ t1' \dashrightarrow^{(i} t1 \}$
- $S + \{ t1 \rightarrow t2 \dashrightarrow^{(i} t1' \rightarrow t2' \}$ ==>
  $S + \{ t2 \dashrightarrow^{(i} t2' \} + \{ t1' \dashrightarrow^{)i} t1 \}$

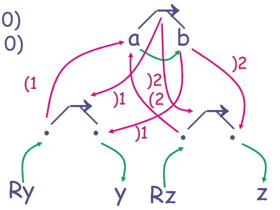
Type Rule for Instantiation

- Now when we mention the name of a function, we'll instantiate it using the following rule

$$\frac{A(f) = t \quad t' = fresh(t) \quad t \dashrightarrow^{)i} t'}{A \mid\text{--}\ f_i : t'}$$

A Simple Example

let id = \x.x in
  let y = id_1 (ref^{Ry} 0)
  let z = id_2 (ref^{Rz} 0)



Two Observations

- We *are* doing constraint copying
  - Notice the edge from c to a got "copied" to Ry to y
    - We didn't draw the transitive edge, but we could have
- This algorithm can be made demand-driven
  - We only need to worry about paths from constant qualifiers
  - Good implications for scalability in practice

CFL Reachability

- We're trying to find paths through the graph whose edges are a language in some grammar
  - Called the *CFL Reachability* problem
  - Computable in cubic time

Grammar for Matched Paths

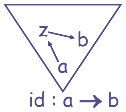
M ::= (i M)i	for any i
| M M	
| d	regular subtyping edge
|	empty

- Also can include other paths, depending on application

Global Variables

- Consider the following identity function

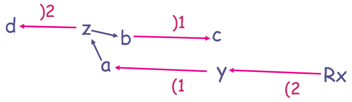
  let id = \x . (z := x; !z)
  - Here z is a global variable
- Typing of id, roughly speaking:



id : a → b

Global Variables

let foo = \y. ((id^{1} y); !z) in
foo^{2} (ref^{Rx} 0)
(Apply id to y, then return the value y via z)

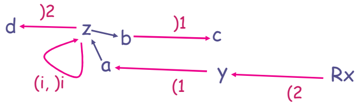


- Uh oh! (2 (1)2 is not a valid flow path
  - But Rx may certainly reach d

Thou Shalt Not Quantify a Global Variable

- We violated a basic rule of polymorphism
  - We generalized a variable free in the environment
  - In effect, we duplicated z at each instantiation
- Solution: Don't do that!

Our Example Again



- We want anything flowing into z, on any path, to flow out in any way
  - Add a self-loop to z that consumes any mismatched parentheses

---

Typing Rules, Fixed

- Track unquantifiable vars at generalization

$$\frac{A \mid\text{--}\ e1 : t1 \quad A, x : (t1, \mathbf{b}) \mid\text{--}\ e2 : t2 \quad \mathbf{b} = fv(A)}{A \mid\text{--}\ \text{let } x = e1 \text{ in } e2 : t2}$$

- Add self-loops at instantiation

$$\frac{\begin{array}{c} A(f) = (t, \mathbf{b}) \quad t' = fresh(t) \quad t \xrightarrow{)^i} t' \\ \mathbf{b} \xrightarrow{)^i} \mathbf{b} \quad \mathbf{b} \xrightarrow{(^i} \mathbf{b} \end{array}}{A \mid\text{--}\ f_i : t'}$$

Label Constants

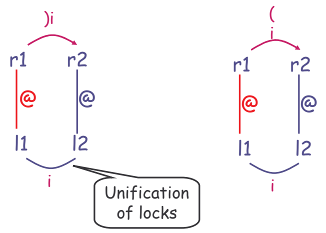
- Also use self-loops for label constants
  - They're global everywhere

Efficiency

- Constraint generation yields O(n) constraints
  - Same as before
  - Important for scalability
- Context-free language reachability is O(n^3)
  - But a few tricks make it practical (not much slowdown in analysis times)
- For more details, see
  - Rehof + Fahndrich, POPL'01

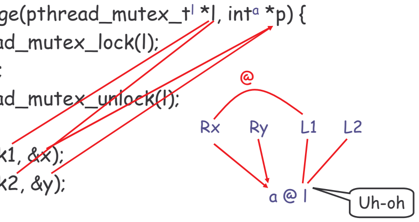
Adapting to Correlation

- Previous propagation rule, but match ()'s



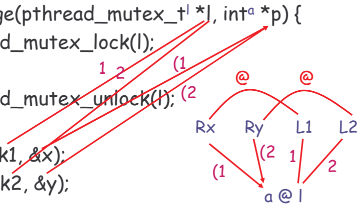
Example

```
pthread_mutex_t k1^{L1} = ..., k2^{L2} = ...;
int x^{Rx}, y^{Ry};
void munge(pthread_mutex_t^{l} *l, int^{a} *p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
munge(&k1, &x);
munge(&k2, &y);
```



Example: Using Context-Sensitivity

```
pthread_mutex_t k1^{L1} = ..., k2^{L2} = ...;
int x^{Rx}, y^{Ry};
void munge(pthread_mutex_t^{l} *l, int^{a} *p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
munge^{1}(&k1, &x);
munge^{2}(&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:

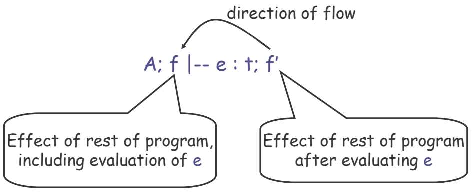
```
struct foo *p = malloc(…);
// initialize *p
fork(<something with p>);  // p becomes shared
// parent no longer uses p
```

  - If we compute
    <effects of parent> ∩ <effects of child>
    then we'll see p in both, and decide it's shared

Continuation Effects

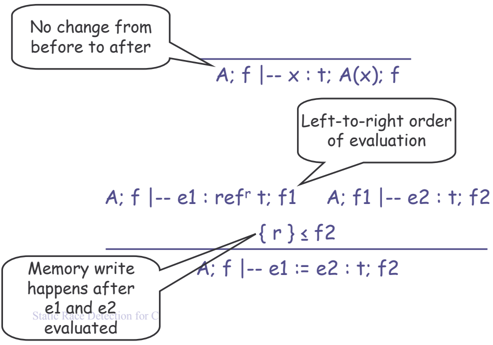
- Continuation effects capture the effect of the remainder of the computation
  - I.e., of the continuation
  - So in our previous example, we would see that in the parent's continuation after the fork, there are no effects
- Effects on locations
  - f ::= 0 | { r } | eff | f + f
    - Empty, locations,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

Judgments



A; f |-- e : t; f'

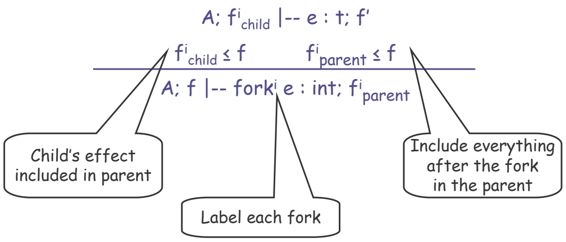
Type Rules



$$\frac{}{A;\ f\ |\text{--}\ x : t;\ A(x);\ f}$$

$$\frac{A;\ f\ |\text{--}\ e1 : ref^{r}\ t;\ f1 \qquad A;\ f1\ |\text{--}\ e2 : t;\ f2 \qquad \{ r \} \leq f2}{A;\ f\ |\text{--}\ e1 := e2 : t;\ f2}$$

Rule for Fork



$$\frac{A;\ f^{i}_{child}\ |\text{--}\ e : t;\ f' \qquad f^{i}_{child} \leq f \qquad f^{i}_{parent} \leq f}{A;\ f\ |\text{--}\ fork^{i}\ e : int;\ f^{i}_{parent}}$$

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
  - $Si = S(f^i_{child}) \cap S(f^i_{parent})$
- And all the shared locations are
  - shared = $\cup_i$ Si

Including Child's Effect in Parent

- Consider:

  let x = ref^{Rx} 0 in

  fork^1 (!x);

  fork^2 (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

void* and Aggregates

Error Messages are Important

```
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)
```

Experimental Results

Benchmark	Size (kloc)	Time	Warn	Unguraded	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

Experimental Results

Benchmark	Size (kloc)	Time	Warn	Unguraded	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*	8	2	1
slip	22.7	16.5s*	19	1	0
hp100	20.3	31.8s*	23	2	0

* = disabled linearity checks

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