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

Cf. The Java Memory Model, recent addition 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) \ 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 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

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

```c
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

```c
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

```c
void logger() { ...
  lock(&list_lock);
  for (j = joblist; j != NULL; j = j->next) {
    cnt++;
    if (trylock(&j->job_lock)) {
      printf(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

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(a); ... if (foo) unlock(a)
- 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

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=ao, q=ao), (p=ib, q=ib), ...
  - 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, (dx)) 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

\[
\begin{align*}
p &= \&x; \\
p &= \&y; \\
*p &= \&z;
\end{align*}
\]

Flow-sensitive:  
(\(p, (\&x, \&y)\))

Flow-insensitive:  
(\(x, \&z\))

A Simple Language

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

        e ::=  \begin{align*}
        n & \quad \text{variables} \\
        \langle \text{f.t.e} \rangle & \quad \text{integers} \\
        \langle e \rangle & \quad \text{functions} \\
        \langle \text{if}\ e\ \text{then}\ e\ \text{else}\ e \rangle & \quad \text{application} \\
        \langle \text{let}\ x = e \text{in}\ e \rangle & \quad \text{conditional} \\
        \langle \text{let}\ e\ =\ e \rangle & \quad \text{binding} \\
        \langle \text{ref}\ e \rangle & \quad \text{allocation} \\
        \langle \text{le} \rangle & \quad \text{dereference} \\
        \langle e := e \rangle & \quad \text{assignment}
\end{align*}

Aliasing in this Language

- ref creates an updatable reference
  - It’s like malloc followed by initialization

- That pointer can be passed around the program

        \[
        \begin{align*}
        \text{let}\ x = \text{ref 0 in} \\
        \text{let}\ y = x\ \text{in} \\
        y := 3. \quad \text{// updates lx}
        \end{align*}
        \]
**Label Flow for Points-to Analysis**

- We’re going to extend references with labels
  - \( e := \ldots | \text{ref}^R \ e \ | \ldots \)
  - 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

  \[
  \text{let } x = \text{ref}^R \ 0 \ \text{in} \\
  \text{let } y = x \ \text{in} \\
  y := 3; /\ y \text{ may point to } (R^x) \\
  \]

---

**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’s what we’re going to do

- We’ll use **labeled types** in our analysis
  - \( t ::= \text{int} \ | \ t \ightarrow t \ | \text{ref}^R \ t \)
  - If we have \( l_x \) or \( x := \ldots \), we can decide what location \( x \) may point to by looking at its ref type

---

**A Type Checking System**

\[
\begin{align*}
A & \vdash x : A(x) & A & \vdash n : \text{int} \\
A, x : t \vdash e : t' & \rightarrow A \vdash \lambda x : e : t \rightarrow t' \\
A \vdash \lambda x : e : t \rightarrow t' & \rightarrow A \vdash \text{if} 0 \ e1 \ \text{then} \ e2 \ \text{else} \ e3 : t \\
A & \vdash \text{if} 0 \ e1 \ \text{then} \ e2 \ \text{else} \ e3 : t & A & \vdash e : t \\
A & \vdash \text{ref}^R \ e : \text{ref}^R \ t & A & \vdash \text{ref}^R \ t \\
A & \vdash e \ : \ t & A & \vdash e \ : \ t \\
A & \vdash e1 : \text{ref}^R \ t & A & \vdash e2 : t \\
A & \vdash e1 : e2 : t \\
A & \vdash e1 : e2 : t
\end{align*}
\]

---

**Example**

- Let \( x = \text{ref}^R \ 0 \ \text{in} \\
- Let \( y = x \ \text{in} \\
- \ y := 3; \)
  - \( x \) has type \( \text{ref}^R \text{int} \)
  - \( y \) must have the same type as \( x \)
  - Therefore at assignment, we know which location \( y \) refers to

---

**Another Example**

- Let \( x = \text{ref}^R \ 0 \ \text{in} \\
- Let \( y = \text{ref}^R \ 0 \ \text{in} \\
- Let \( w = \text{ref}^R \ 0 \ \text{in} \\
- Let \( z = \text{if} 0 \ 42 \ \text{then} \ x \ \text{else} \ y \ \text{in} \\
- z := 3; \)
  - \( x \) and \( y \) both have type \( \text{ref}^R \text{int} \)
    - 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 3
- let y = ref x
- let z = ref 4
  y := z

- Both x and z have the same label
- y has type ref (ref 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?

A |- e : t
A |- ref e : ref 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)

\[
A, x : t \vdash e : t' \quad t \leftarrow fresh(s)
\]

\[
A \vdash \lambda x : s . e : t \rightarrow t'
\]

Step 2: Adding Constraints

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

\[
A \vdash e_1 : int \quad A \vdash e_2 : t_2 \quad A \vdash e_3 : t_3
\]

\[
t_2 = t_3
\]

\[
A \vdash if \ e_1 \ then \ e_2 \ else \ e_3 : t_2
\]

Step 2: Adding Constraints (cont’d)

\[
A \vdash e_1 : \text{ref} \rightarrow t' \quad A \vdash e_2 : t_2 \quad t = t_2
\]

\[
A \vdash e_1 := e_2 : t
\]

• Notice we’re assuming that \( e_1 \) 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)

\[
A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t_2 \quad t = t_2
\]

\[
A \vdash e_1 e_2 : t'
\]

• Again, we’re assuming \( e_1 \) is a function

Constraint Resolution

• After applying the rules, we are left with a set of equality constraints
  - \( t_1 = t_2 \)

• We’ll solve the constraints via rewriting
  - We’ll simplify more complex constraints into simpler constraints
  - \( S \leftrightarrow S' \) rewrite constraints \( S \) to constraints \( S' \)

Constraint Resolution via Unification

• \( S + (\text{int} + \text{int}) \Rightarrow S \)
• \( S + (t_1 \rightarrow t_2 \leftarrow t_1 \rightarrow t_2) \Rightarrow \)
  \[
  S + (t_1 \leftarrow t_1) + (t_2 \leftarrow t_2)
  \]
• \( S + (\text{ref} \leftarrow t_1 \leftarrow \text{ref} \leftarrow t_2) \Rightarrow \)
  \[
  S + (t_1 \leftarrow t_2) + (a_1 \leftarrow a_2)
  \]
• \( S + (\text{mismatched constructors}) \Rightarrow \text{error} \)
  - 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 = if 0 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

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^a int
  let y = ref 0 in // y : ref^b int
  let z = if 0 42 then x else y in // z : ref^a
      z := 3; // write to ref^a
      x := 2; // write to ref^a
  ```

- 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
  - \( a \preceq a^2 \text{ means } [a] \subseteq [a^2] \)

\[
\begin{align*}
A & \vdash e : R \quad a \text{ a fresh} \\
A & \vdash \text{ref}^a e : \text{ref}^a t
\end{align*}
\]

**Subtype Inference**

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

\[
\begin{align*}
A & \vdash e_1 : \text{int} \\
A & \vdash e_2 : \text{ref}^r t \\
A & \vdash e_3 : \text{ref}^r t \\
r^2 & \subseteq r \\
r^3 & \subseteq r \\
A & \vdash \text{if} e_1 \text{ then } e_2 \text{ else } e_3 : \text{ref}^r t
\end{align*}
\]

**Subtyping Constraints**

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

\[
\begin{align*}
\text{int} & \preceq \text{int} \\
\text{ref}^a \text{int} & \preceq \text{ref}^b \text{int}
\end{align*}
\]

**Subsumption**

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

\[
\begin{align*}
A & \vdash e_1 : t \\
& \vdash t \subseteq t' \\
A & \vdash e_1 : t'
\end{align*}
\]

- 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**

```c
let x = ref^0x 0 in       // x : ref^a int, Rx \preceq a
let y = ref^x 1 in       // y : ref^b int, Ry \preceq b
let z = if 42 then x else y in
x := 3
```

- At conditional, need types of \( x \) and \( y \) to match \( a \preceq c \)

\[
\begin{align*}
A & \vdash x : \text{ref}^a \text{int} \\
& \vdash \text{ref}^a \text{int} \preceq \text{ref}^c \text{int} \\
A & \vdash x : \text{ref}^c \text{int}
\end{align*}
\]

- Thus we have \( z : \text{ref}^c \text{int} \) with \( a \preceq c \) and \( b \preceq c \)
- Thus can pick \( a = \text{Rx} \), \( b = \text{Ry} \), \( c = \text{Rx, Ry} \)

**Subtyping References (cont’d)**

- Let’s try generalizing to arbitrary types

\[
\begin{align*}
r^1 & \preceq r^2 \\
t_1 & \preceq t_2 \\
\text{ref}^r t_1 & \preceq \text{ref}^r t_2
\end{align*}
\]

- This rule is broken

```c
let x = ref^0x (ref^0x 0) in       // x : ref^a (ref^b int), Rx \preceq b
let y = x in       // y : ref^b (ref^d int), b \preceq d
y := ref^0x 0       // Oops \preceq d
lx := 3       // dereference of b
```

- Can pick \( b = \{\text{Rx}\} \), \( d = \{\text{Rx}^\prime, \text{Oops}\} \)
  - Then write \( \text{via} b \) doesn’t look like it’s writing \( \text{Oops} \)
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
  \[
  r_1 \leq r_2 \quad t_1 \leq t_2 \quad t_2 \leq t_1
  \]
  \[
  \text{ref}^2 t_1 \leq \text{ref}^2 t_2
  \]
  or
  \[
  r_1 \leq r_2 \quad t_1 \leq t_2
  \]
  \[
  \text{ref}^2 t_1 \leq \text{ref}^2 t_2
  \]

Subtyping on Function Types

- What about function types?

  \[
  \begin{array}{c}
  t_1 \rightarrow \text{ref}^2 t_2 \\
  \text{ref} \end{array}
  \]

- 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 \( t_1' \rightarrow t_2' \leq t_1 \rightarrow t_2 \) ?

- Return type:
  - We are expecting \( t_2 \) (f’s return type)
  - So we can only return at most \( t_2 \)
  - \( t_2 \leq t_2' \)

- Example: A function that returns a pointer to \( \{R_1, R_2\} \) can be treated as a function that returns a pointer to \( \{R_1, R_2, R_3\} \)

Replacing “f x” by “g x” (cont’d)

- When is \( t_1' \rightarrow t_2' \leq t_1 \rightarrow t_2 \) ?

- Argument type:
  - We are supposed to accept \( t_1 \) (f’s argument type)
  - So we must accept at least \( t_1 \)
  - \( t_1 \leq t_1' \)

- Example: A function that accepts a pointer to \( \{R_1, R_2, R_3\} \) can be passed a pointer to \( \{R_1, R_2\} \)
Subtyping on Function Types

\[ t_1' \leq t_1 \quad t_2 \leq t_2' \]
\[ t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2' \]

• We say that \( \rightarrow \) 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

\[ A \vdash e : t \leq t' \]
\[ A \vdash e : t' \]

• 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 \( \leq \)
• 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
    \[ A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t_2 \quad t = t_2 \]
    \[ A \vdash e_1 e_2 : t' \]
  becomes
    \[ A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t_2 \quad t_2 \leq t \]
    \[ A \vdash e_1 e_2 : t' \]
• Apply the same reasoning to the other rules
  - We’re left with a purely syntax-directed system

Constraint Resolution: Step 1

• \( S + \{ \text{int} \leq \text{int} \} \implies S \)
• \( S + \{ t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2' \} \implies \)
  \[ S + \{ t_1 \leq t_1' \} \}
• \( S + \{ \text{ref}^2 t_1 \leq \text{ref}^2 t_2 \} \implies \)
  \[ S + \{ t_1 \leq t_2 \} \}
• \( S + \{ \text{mismatched constructors} \} \implies \text{error} \)
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 \( r_1 \leq r_2 \)
  - Where \( r_1 \) and \( r_2 \) 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
  - \( S \cdot \{ r_1 \leq r_2 \} \cdot \{ r_2 \leq r_3 \} \Rightarrow \{ r_1 \leq r_3 \} \)
  - \( \Rightarrow \) means add the constraint without removing lhs constraints
  - Apply this rule until no new constraints generated
  - Then \( [a] \subseteq \{ R | R \leq a \} \text{ is a constraint in } S \)

Graph Reachability

- Think of a constraint as a directed edge
  - \( R_1 \leq a \)
  - \( R_2 \leq b \)
  - \( a \leq c \)
  - \( b \leq a \)
  - Use graph reachability to compute solution
    - Compute set of constants that reach each variable
      - E.g., \([c] \subset [a] = (R_1, R_2), [b] = (R_2)\)
    - Complexity?

Andersen’s Analysis

- Flow-insensitive
- Context-insensitive
- Subtyping-based
  - \( \Rightarrow \) Andersen’s analysis
  - \( \Rightarrow \) 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)

Programming Against Races

- Recall our model:
  - Locations \( r \)
  - Locks \( l \)
  - Correlation: \( r \bowtie l \)
  - 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

Back to Race Detection
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 $l$
      - $h$...need to think about these
      - Draw from a set of constant lock labels $L$, plus variables $m$
    - Correlation: $r \otimes l$
      - $h$...need to associate locks and locations somehow
      - Let’s punt this part

Lambda-Corr

- A small language with “locations” and “locks”
  - $e ::= x \mid n \mid x + e \mid e \mid if e e$ then $e$ else $e$
    - $\mathtt{newlock}$
      - create a new lock
    - $\mathtt{ref}\ e$
      - allocate “shared” memory
    - $\mathtt{h}\ e$
      - dereference with a lock held
    - $\mathtt{e} \mathtt{:=}\ e$
      - assign with a lock held
    - $t ::= \mathtt{int} \mid t \mathtt{:=} t \mid \mathtt{lock} l \mid \mathtt{ref}\ t$

- No acquire and release
  - All accesses have explicit annotations (superscript) of the lock
    - This expression evaluates to the lock to hold
- No thread creation
  - $\mathtt{ref}$ creates “shared” memory

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 labels for them
  - Standard labeling rule, standard subtyping
  - Warning: this is broken! Will fix later...

- $\mathtt{newlock}$
  - $\mathsf{fresh}$

\begin{align*}
(L \leq m & \quad m \text{ fresh}) \\
A &\vdash \mathtt{newlock}\ ! : \mathtt{lock} \, m \\
   &\vdash l1 \leq l2 \quad \text{lock} l1 \leq \text{lock} l2
\end{align*}

Correlation Constraints for Locations

- Generate a correlation constraint $r \otimes l$ when location $r$ is accessed with lock $l$ held

\begin{align*}
A \vdash e1 : \mathtt{ref} &\otimes t & A \vdash e2 : \mathtt{lock} &\otimes l & r \otimes l \\
A &\vdash \mathtt{ref}\ e1 : t
\end{align*}

\begin{align*}
A \vdash e1 : \mathtt{ref} &\otimes t & A \vdash e2 : t &\otimes l & A \vdash e3 : \mathtt{lock} &\otimes l & r \otimes l \\
A &\vdash e1 :\mathtt{ref}\ e2 : t
\end{align*}
Constraint Resolution

- Apply subtyping until only atomic constraints
  - \( r_1 \leq r_2 \) — location subtyping
  - \( l_i \leq l_j \) — lock subtyping
  - \( r @ l \) — correlation

- Now apply three rewriting rules
  - \( S \times (r_1 \leq r_2) \times (r_2 \leq r_3) \Rightarrow (r_1 \leq r_3) \)
  - \( S \times (l_i \leq l_j) \times (l_j \leq l_k) \Rightarrow (l_i \leq l_k) \)
  - If \( r_i \) "flows to" \( r_j \) and \( r_j \) "flows to" \( r_k \)
    - \( r_k \) are correlated, then so are \( r_i \) and \( r_k \)
    - Note: \( r \times r \) and \( l \times l \)

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 \( R \) 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

- \( k_1 = \text{newlock}^{k_1} \) in \( k_1 : \text{lock} \) \( m, L_1 \leq m \)
- \( k_2 = \text{newlock}^{k_2} \) in \( k_2 : \text{lock} \) \( n, L_2 \leq n \)
- \( x = \text{ref}^{Rx} 0 \) in \( x : \text{ref}^{\text{int}}(Rx) \leq a \)
- \( y = \text{ref}^{Ry} 1 \) in \( y : \text{ref}^{\text{int}}(Ry) \leq b \)
  - \( x := k_1 3; \)
  - \( x := k_4 4; \)
  - \( y := k_5 b; \)
  - \( y := k_6 b; \)

- Applying last constraint resolution rule yields
  - \( (Rx @ L_1) + (Rx @ L_1) + (Ry @ L_1) + (Ry @ L_2) \)
  - Inconsistent correlation for \( R_y \)

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

- \( k_1 = \text{newlock}^{k_1} \)
- \( x = \text{ref}^{Rx} 0 \)
- \( y = \text{ref}^{Ry} 0 \)
- \( z = \text{if} 42 \text{ then } x \text{ else } y \)
  - \( z := k_3 \)

- 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^1
let k2 = newlock^2
let k = if 0 42 then k1 else k2
let x = refPx 0
\[ x := k \text{ 3}; x := k1 \text{ 4} \]

- \( (Rx @ L1) + (Rx @ L2) + (Rx @ L1) \)
- 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^1
let k2 = newlock^2 // fine according to rules
let k = if 0 42 then k1 else k2
let x = refPx 0
\[ x := k \text{ 3}; x := k1 \text{ 4} \]

- \( (Rx @ L) + (Rx @ L) + (Rx @ L) \)
- 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 runtime 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^1
  - let k2 = newlock^2
    - Need to prevent lock labels from being reused
- Solution: remember newlock^d 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

\[
\begin{align*}
f &::= 0 & \text{no effect} \\
| eff & \text{effect variable} \\
| (l) & \text{lock l was allocated} \\
| f + f & \text{union of effects} \\
| f \oplus f & \text{disjoint union of effects}
\end{align*}
\]
Type Rules with Effects

\[
L \leq m \quad m \text{ fresh} \\
A \vdash \text{newlock}^d : \text{lock}_m: (m)
\]

Judgments now assign a type and effect

Type Rules with Effects (cont’d)

\[
A \vdash x : A(x) : 0 \\
A \vdash e_1 : \text{ref}^t : t_1 \quad A \vdash e_2 : t : t_2 \\
A \vdash e_1 : e_2 : t : t_1 @ t_2 \\
\]

Prevents 1 alloc

\[
A \vdash e_1 : \text{int} : t_1 \quad A \vdash e_2 : t : t_2 \quad A \vdash e_3 : t : t_3 \\
A \vdash \text{if}0 e_1 \text{then} e_2 \text{else} e_3 : t : t_1 @ (t_2 @ t_3)
\]

Only one branch taken

Rule for Functions

• Is the following rule correct?

\[
A, x : t \vdash e : t ; t' \quad f \\
A \vdash \lambda x : t. e : t ; t' ; f
\]

- 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

\[
t ::= \text{int} | \text{t} \rightarrow t \mid \text{lock} \mid \text{ref}^t
\]

\[
A, x : t \vdash e : t ; t' \quad f \\
A \vdash \lambda x : t. e : t ; t' ; 0
\]

\[
A \vdash e_1 : t \rightarrow t' ; t_1 \quad A \vdash e_2 : t ; t_2 \\
A \vdash e_1 e_2 : t ; t_1 @ t_2 @ f
\]

One Minor Catch

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

\[
A \vdash e : t ; f \quad t \sqsupseteq t' ; f \quad \text{eff}
A \vdash e : t' ; \text{eff}
\]

- We always use a variable as an upper bound
- Otherwise how would we solve constraints like
  \[
  \{L_1\} \cdot \{L_2\} \cdot f : \{L_1\} \cdot g + h >
  \]

Safe to assume have more effects

Another Minor Catch

• We don’t have types with effects on them

\[
A, x : s \vdash e : t ; t = \text{fresh}(s) \\
A \vdash \lambda x : s. e : t ; t = \text{effect}(s)
\]

Fresh label variables and effect variables
### Effect Constraints

- The same old story!
  - Walk over the program
  - Generate constraints
    - \( r_1 \leq r_2 \)
    - \( l_1 \leq l_2 \)
    - \( f \leq \text{eff} \)
    - Effects include disjoint unions
  - Solution \( \Rightarrow \) locks can be treated linearity
  - No solution \( \Rightarrow \) reject program

### Effect Constraint Resolution

- **Step 1:** Close lock constraints
  - \( S \ast ( l_1 \leq l_2 ) \ast ( l_2 \leq l_3 ) \Rightarrow ( l_1 \leq l_3 ) \)
- **Step 2:** Count!
  - \( \text{occurr}(l, 0) = 0 \)
  - \( \text{occurr}(l, \{ l \}) = 1 \)
  - \( \text{occurr}(l, \{ f \}) = 1 \Rightarrow l = f \)
  - \( \text{occurr}(l, f_1 \oplus f_2) = \text{occurr}(l, f_1) + \text{occurr}(l, f_2) \)
  - \( \text{occurr}(l, f_1 \ast f_2) = \max(\text{occurr}(l, f_1), \text{occurr}(l, f_2)) \)
  - \( \text{occurr}(l, \text{eff}) = \max(\text{occurr}(l, f) \text{ for } f \neq \text{eff}) \)
  - For each effect \( f \) and for every lock \( l \), make sure that occurs \( \text{occurr}(l, f) \leq 1 \)

### Example

```plaintext
let k1 = newlock;
let k2 = newlock;  // violates disjoint union
let k = if 42 then k1 else k2; // k1, k2 have same type
let x = ref 0;
  x := 3; x := 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:
    ```c
    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:
  - `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

```c
void foo(int x) {
  let x = ref x in
  int y;
  let y = x + 3;
  y := (x) + 3;
  x := 42;
  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

```c
pthread_mutex_t k1 = ... // k1: lock L1
int x; // x: ref\(^{\text{rel}} \) int

// L: lock L, p: ref\(^{\text{rel}} \) (ref\(^{\text{rel}} \) int)
void muge(pthread_mutex_t *l, int *p) {
  pthread_mutex_lock(l);
  pthread_mutex_unlock(l);
}

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

Solving Constraints

![Graph showing constraints](image)

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 \( r @ \{1, \ldots, n\} \)
  - Correlation set \( S(r) = \{ \{1, \ldots, l\} | r @ \{1, \ldots, n\} \} \)
  - Consistent correlation: for every \( R \), \( |rS(R)| \geq 1 \)

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 \( \epsilon \) 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

Limitations of Subtyping

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

Context-Sensitivity

Limitations of Subtype Polymorphism

- Let’s look at the identity function on int pointers:
  - \( \text{let } \text{id} = \lambda \text{x}: \text{ref}^\text{int}. \text{x} \)
  - So \( \text{id} \) has type \( \text{ref}^\text{int} \rightarrow \text{ref}^\text{int} \)
- Now consider the following:
  - \( \text{let } x = \text{id}(\text{ref}^1 \text{O}) \)
  - \( \text{let } y = \text{id}(\text{ref}^2 \text{O}) \)
  - \( \text{It looks like } ax \text{ and } ay \text{ point to } (r1, r2) \)
  - This is a context-insensitive analysis

The Observation of Parametric Polymorphism

- Type inference on \( \text{id} \) yields a proof like this:

  ![Proof Tree](image)

  This is a proof tree

The Observation of Parametric Polymorphism

- We can duplicate this proof for any \( a, a' \), in any type environment

  ![Duplicate Proof](image)
The Observation of Parametric Polymorphism

- Thus when we use `id`...

```
\begin{center}
\begin{tikzpicture}
  \node (a) {a};
  \node (b) [right of=a] {a'};
  \node (c) [below of=a] {a};
  \node (d) [right of=c] {a'};
  \draw[->] (a) -- (b);
  \draw[->] (c) -- (d);
  \draw[->,red] (a) -- (c);
  \draw[->,red] (b) -- (d);
\end{tikzpicture}
\end{center}
```

The Observation of Parametric Polymorphism

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

```
\begin{center}
\begin{tikzpicture}
  \node (a) {a};
  \node (b) [right of=a] {a'};
  \node (c) [below of=a] {a};
  \node (d) [right of=c] {a'};
  \draw[->] (a) -- (b);
  \draw[->] (c) -- (d);
  \draw[->,red] (a) -- (c);
  \draw[->,red] (b) -- (d);
\end{tikzpicture}
\end{center}
```

Hindley-Milner Style Polymorphism

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

```
\begin{center}
\begin{align*}
A \vdash e_1 : t_1 & \quad A, f : \forall a t_1 \vdash e_2 : t_2 \quad a = \text{fv}(t_1) - \text{fv}(A) \\
A \vdash \text{let } f = e_1 \text{ in } e_2 : t_2
\end{align*}
\end{center}
```

- Instantiate at uses

```
\begin{center}
\begin{align*}
A(f) = \forall a t_1 \\
A \vdash f : t_1[a\backslash t]\quad \text{(arbitrarily)}
\end{align*}
\end{center}
```

Polymorphically Constrained Types

- Notice that we inlined not only the type (as in ML), but also the constraints

- We need polymorphically constrained types

```
\begin{center}
\begin{align*}
x : \forall a t & \text{ where } C
\end{align*}
\end{center}
```

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

Polymorphically Constrained Types

- Must copy constraints at each instantiation

```
\begin{center}
\begin{tikzpicture}
  \node (a) {a};
  \node (b) [right of=a] {a'};
  \node (c) [below of=a] {a};
  \node (d) [right of=c] {a'};
  \draw[->] (a) -- (b);
  \draw[->] (c) -- (d);
  \draw[->] (a) -- (c);
  \draw[->] (b) -- (d);
\end{tikzpicture}
\end{center}
```

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: $O(n^3)$ (Massin, PhD thesis)
- Idea due to Horwitz, Reps, and Sagiv [POPL’95], and Rehof, Fahndrich, and Das [POPL’01]

The Problem Restated: Unrealizable Paths

- No execution can exhibit that particular call/return sequence

Only Propagate Along Realizable Paths

- $\text{let } id = \lambda x: \text{ref } \text{int } . x$
- $\text{let } x = id  \text{ (ref } 0\text{ )}$
- $\text{let } y = id  \text{ (ref } 0\text{ )}$

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

Parenthesis Edges

- Paren edges represent substitutions
  - $\text{id : } a, b \rightarrow b \text{ where } a \leq b$
  - $\text{let } x = id  \text{ (ref } 0\text{ )}$
  - At call 1 to id, we instantiate type of id
    - $(a \rightarrow b)[r1/a, ax/b] = r1 \rightarrow ax$
    - Renaming for call 1
- Edges with $)$1 or (1 represent renaming 1
  - $b \rightarrow )1 ax b$ instantiated to $ax$, and $b$ flows to $ax$
  - $r1 \rightarrow )1 a$ 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

- $S + \{ \text{int } \rightarrow \text{int } \} \Rightarrow S$
- $S + \{ \text{int } \rightarrow \text{int } \} \Rightarrow S$
- $S + \{ \text{ref } t1 \rightarrow \text{ref } t2 \} \Rightarrow $
  - $S + \{ r1 \rightarrow r2 \} \Rightarrow (t1 \rightarrow t2) \Rightarrow (t2 \rightarrow t1)$
- $S + \{ \text{ref } t1 \rightarrow \text{ref } t2 \} \Rightarrow$
  - $S + \{ r1 \rightarrow r2 \} \Rightarrow (t1 \rightarrow t2) \Rightarrow (t2 \rightarrow t1)$
Propagating Instantiation Constraints (cont’d)

- \( S + \{ t_1 \rightarrow t_2 \rightarrow t'_2 \} \implies\)
  \( S + \{ t_2 \rightarrow t'_2 \} + \{ t'_1 \rightarrow t_1 \} \)

- \( S + \{ t_1 \rightarrow t_2 \rightarrow t'_1 \rightarrow t'_2 \} \implies\)
  \( S + \{ t_2 \rightarrow t'_2 \} + \{ t'_1 \rightarrow t_1 \} \)

Type Rule for Instantiation

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

\[
A(f) = t \quad t' = \text{fresh}(t) \quad t \rightarrow t'
\]

\[
A \vdash f : t'
\]

A Simple Example

Let id = \( \lambda x . x \) in
Let y = id, (ref y, O)
Let z = id, (ref y, O)

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

\[
M ::= (i \ M) i \quad \text{for any } i
\]

| \( M M \) | regular subtyping edge |
| \( d \) | empty |

- Also can include other paths, depending on application
Global Variables

- Consider the following identity function
  \[
  \text{let id} = \lambda x. (z := x; \text{id})
  \]
  - Here \( z \) is a global variable
- Typing of id, roughly speaking:

\[
\begin{align*}
\text{id} & : a \rightarrow b \\
\end{align*}
\]

Global Variables

\[
\text{let foo} = \lambda y. ((\text{id}^2 \ y); \text{id}) \text{ in} \\
\text{foo}^2 \ (\text{ref}^k \ 0)
\]

(Apply \text{id} to \( y \), then return the value \( y \) via \( z \))

\[
\begin{align*}
\text{id} & : a \rightarrow b \\
\text{id} & \text{ is a global variable}
\end{align*}
\]

- 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
  \[
  \begin{align*}
  A \vdash e_1 : t_1 & \quad A, x : (t_1, b) \vdash e_2 : t_2 & \quad b = \text{fv}(A) \\
  A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2
  \end{align*}
  \]

- Add self-loops at instantiation
  \[
  A(f) = (t, b) \quad t' = \text{fresh}(t) \quad t \rightarrow t'
  \]
  \[
  b \cdot b \quad b \rightarrow b
  \]
  \[
  A \vdash f_1 : t'
  \]

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**

```c
pthread_mutex_t k1 = ..., k2 = ...;
int x, y;
void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
    munge(&k1, &x);
    munge(&k2, &y);
}
```

---

**Example: Using Context-Sensitivity**

```c
pthread_mutex_t k1 = ..., k2 = ...;
int x, y;
void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
    munge(&k1, &x);
    munge(&k2, &y);
}
```

---

**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:
  ```c
  struct foo *p = malloc(...);
  // initialize *p
  fork(something with p); // p becomes shared
  // parent no longer uses p
  ```
- If we compute
  ```
  \{\text{effects of parent}\} \cap \{\text{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
  - \( f := O \mid \{ r \} \mid \text{eff} \mid f + f \)
  - Empty, locations, variables, union

Judgments

- Direction of flow
- Effect of rest of program, including evaluation of \( e \)
- Effect of rest of program after evaluating \( e \)

Type Rules

- No change from before to after
  ```
  A; f |- x : t; A(x); f
  ```
- Left-to-right order of evaluation
  ```
  A; f |- e1 : ref^r; f1
  A; f1 |- e2 : t; f2
  ```
- Memory write happens after \( e1 \) and \( e2 \) evaluated

Rule for Fork

- Child’s effect included in parent
- Include everything after the fork in the parent
- Label each fork
Computing Sharing

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

Including Child’s Effect in Parent

- Consider:
  - let $x = \text{ref}^R A$ in
    fork $(i)$;
    fork $(x := 2)$;
- 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
&kwritten(aget_comb.c:943)
References:
derereference at aget_comb.c:1079
locks acquired at dereference:
dkwritten_mutex(aget_comb.c:996)
in: FORK at aget_comb.c:468 ->
http_get_agnet_comb.c:468
derereference at aget_comb.c:984
locks acquired at dereference:
(two)
in: FORK at aget_comb.c:193 ->
signal_waiter(aget_comb.c:193) ->
signal_handler(aget_comb.c:957)

Experimental Results
Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (kloc)</th>
<th>Time</th>
<th>Warm</th>
<th>Ungraded</th>
<th>Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>plp</td>
<td>19.1</td>
<td>24.9s</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>eql</td>
<td>16.5</td>
<td>3.2s</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3c501</td>
<td>17.4</td>
<td>240.1s</td>
<td>24</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sundance</td>
<td>19.9</td>
<td>98.2s</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sis900</td>
<td>20.4</td>
<td>61.0s*</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>slip</td>
<td>22.7</td>
<td>16.5s*</td>
<td>19</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hp100</td>
<td>20.3</td>
<td>31.8s*</td>
<td>23</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

*- 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