Improving Software Quality and Security with Type Qualifiers

Jeff Foster
University of Maryland

Joint work with Alex Aiken, Rob Johnson, John Kodumal, Tachio Terauchi, and David Wagner

Introduction

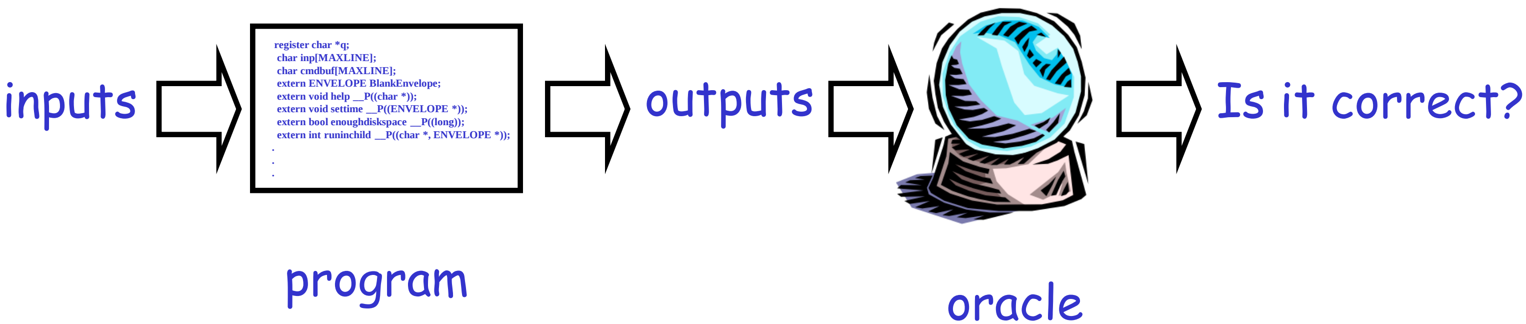
- Ensuring that software is reliable is hard
  - And doing so is important

[T]he national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion.

-- US National Institute of Standards and Technology Planning Report 02-3, May 2002

Current Practice

- Testing
  - Make sure program runs correctly on set of inputs



  - Drawbacks: Expensive, difficult, hard to cover all code paths, no guarantees

Current Practice (cont'd)

- Code Auditing
  - Convince someone else your source code is correct
  - Drawbacks: Expensive, hard, no guarantees


```
register char *q;
 char inp[MAXLINE];
 char cmdbuf[MAXLINE];
 extern ENVELOPE BlankEnvelope;
 extern void help __P((char *));
 extern void settime __P((ENVELOPE *));
 extern bool enoughdiskspace __P((long));
 extern int runinchild __P((char *, ENVELOPE *));
 extern void checksmtpattack __P((volatile int *, int, char *, ENVELOPE *));

 if (fileno(OutChannel) != fileno(stdout))
 {
  /* arrange for debugging output to go to remote host */
  (void) dup2(fileno(OutChannel), fileno(stdout));
 }
 settime(e);
 peerhostname = RealHostName;
 if (peerhostname == NULL)
  peerhostname = "localhost";
 CurHostName = peerhostname;
 CurSmtpClient = macvalue('_', e);
 if (CurSmtpClient == NULL)
  CurSmtpClient = CurHostName;

 setproctitle("server %s startup", CurSmtpClient);
#if DAEMON
 if (LogLevel > 11)
 {
  /* log connection information */
  sm_syslog(LOG_INFO, NOQID,
   "SMTP connect from %.100s (%.100s)",
   CurSmtpClient, anynet_ntoa(&RealHostAddr));
 }
#endif

 /* output the first line, inserting "ESMTP" as second word */
 expand(SmtpGreeting, inp, sizeof inp, e);
 p = strchr(inp, '\n');
 if (p != NULL)
  *p++ = '\0';
 id = strchr(inp, ' ');
 if (id == NULL)
  id = &inp[strlen(inp)];
 cmd = p == NULL ? "220 %.*s ESMTP%s" : "220-%.*s ESMTP%s";
 message(cmd, id - inp, inp, id);

 /* output remaining lines */
 while ((id = p) != NULL && (p = strchr(id, '\n')) != NULL)
 {
  *p++ = '\0';
  if (isascii(*id) && isspace(*id))
```

```
cmd <&cmdbuf[sizeof cmdbuf - 2])
   *cmd++ = *p++;
  *cmd = '\0';

  /* throw away leading whitespace */
  while (isascii(*p) && isspace(*p))
   p++;

  /* decode command */
  for (c = CmdTab; c->cmdname != NULL; c++)
  {
   if (!strcasecmp(c->cmdname, cmdbuf))
    break;
  }

  /* reset errors */
  errno = 0;

  /*
  **  Process command.
  **
  **  If we are running as a null server, return 550
  **  to everything.
  */

  if (nullserver)
  {
   switch (c->cmdcode)
   {
    case CMDQUIT:
    case CMDHELO:
    case CMDEHLO:
    case CMDNOOP:
    /* process normally */
    break;

    default:
    if (++badcommands > MAXBADCOMMANDS)
     sleep(1);
    usrerr("550 Access denied");
    continue;
   }
  }

  /* non-null server */
  switch (c->cmdcode)
  {
   case CMDMAIL:
   case CMDEXPN:
   case CMDVRFY:
```

```
while (isascii(*p) && isspace(*p))
     p++;
    if (*p == '\0')
     break;
    kp = p;

    /* skip to the value portion */
    while ((isascii(*p) && isalnum(*p)) || *p == '-')
     p++;
    if (*p == '=')
    {
     *p++ = '\0';
     vp = p;

     /* skip to the end of the value */
     while (*p != '\0' && *p != ' ' &&
         !(isascii(*p) && iscntrl(*p)) &&
         *p != '=')
      p++;
    }

    if (*p != '\0')
     *p++ = '\0';

    if (tTd(19, 1))
     printf("RCPT: got arg %s=\"%s\"\n", kp,
      vp == NULL ? "<null>" : vp);

    rcpt_esmtp_args(a, kp, vp, e);
    if (Errors > 0)
     break;
   }
   if (Errors > 0)
    break;

   /* save in recipient list after ESMTP mods */
   a = recipient(a, &e->e_sendqueue, 0, e);
   if (Errors > 0)
    break;

   /* no errors during parsing, but might be a duplicate */
   e->e_to = a->q_paddr;
   if (!bitset(QBADADDR, a->q_flags))
   {
    message("250 Recipient ok%s",
     bitset(QQUEUEUP, a->q_flags) ?
      " (will queue)" : "");
    nrcpts++;
   }
   else
   {
    /* punt -- should keep message in ADDRESS.... */
```

And If You're Worried about Security…

A malicious adversary is trying to exploit anything you miss!

What more can we do?

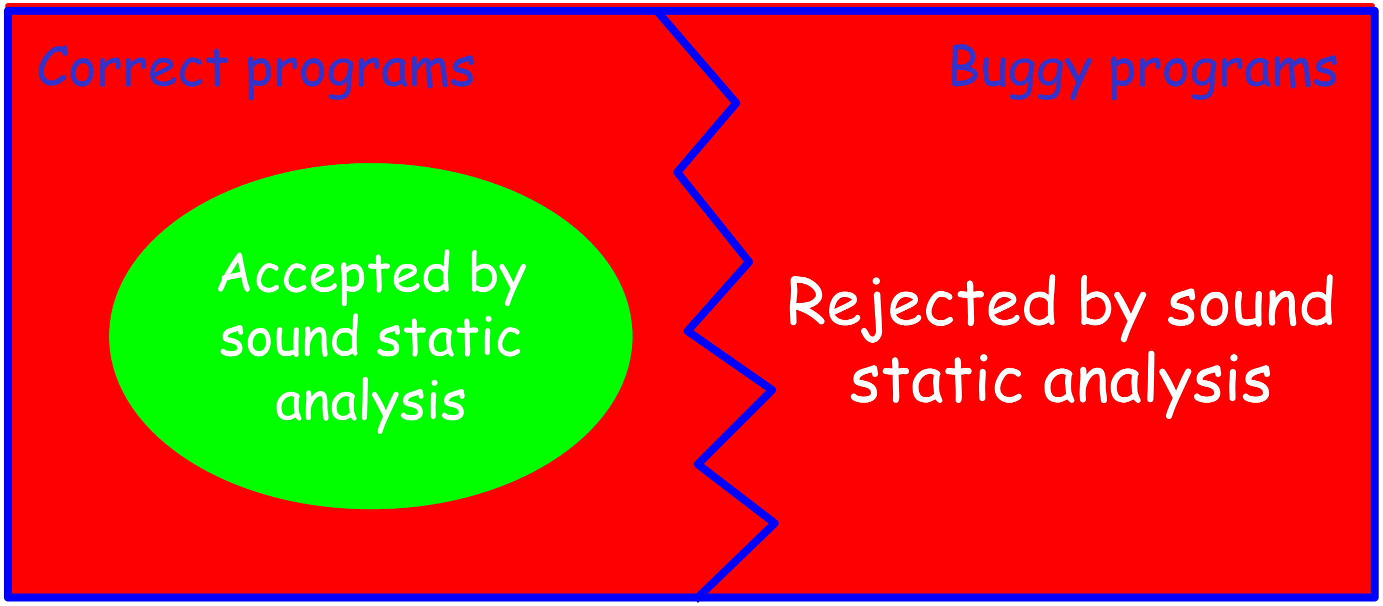
Tools for Software Quality

- Build tools that analyze source code (static analysis)
  - Reason about all possible runs of the program
- Check limited but very useful properties
  - Eliminate categories of errors
  - Let people concentrate on the deep reasoning
- Develop programming models
  - Avoid mistakes in the first place
  - Encourage programmers to think about and make manifest their assumptions

Oops — We Can't Do This!

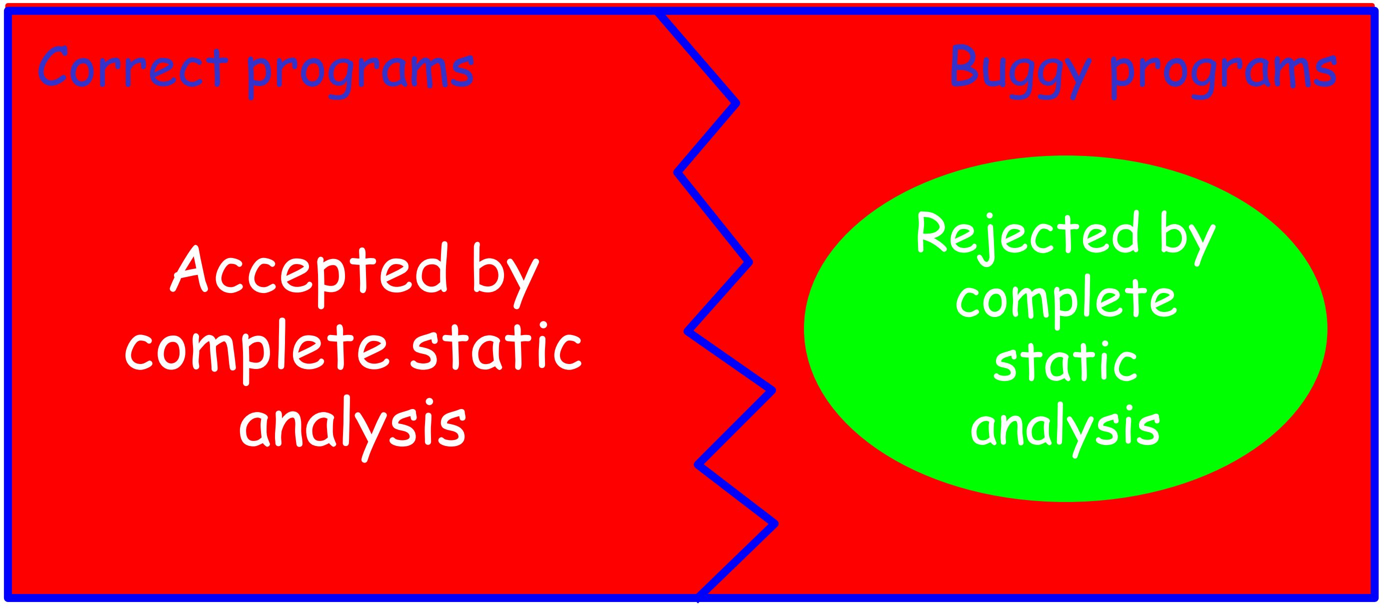
- Rice's Theorem: No computer program can precisely determine anything interesting about arbitrary source code
  - Does this program terminate?
  - Does this program produce value 42?
  - Does this program raise an exception?
  - Is this program correct?

Approximations in Static Analysis



- Sound: Only correct programs accepted

Approximations in Static Analysis



- Sound: Only correct programs accepted
- Complete: Only incorrect programs rejected

Approximations in Static Analysis (cont'd)

- Warning: Some people flip the senses of sound or complete direction

- Consider the type system of simply typed lambda calculus to be a static analysis
  - Is it sound or complete?

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 reject ugly code in the name of safety

Type Qualifiers

- Extend standard type systems (C, Java, ML)
  - Programmers already use types
  - Programmers understand types
  - Get programmers to write down a little more...

const int	ANSI C
ptr(tainted char)	Format-string vulnerabilities
kernel ptr(char) → char	User/kernel vulnerabilities

Application: Format String Vulnerabilities

- I/O functions in C use format strings

```
printf("Hello!");                 Hello!
printf("Hello, %s!", name);       Hello, name!
```

- Instead of

```
printf("%s", name);
```

Why not

```
printf(name);                     ?
```

Format String Attacks

- Adversary-controlled format specifier

```
name := <data-from-network>
printf(name);   /* Oops */
```

  - Attacker sets name = "%s%s%s" to crash program
  - Attacker sets name = "...%n..." to write to memory
    - Yields (often remote root) exploits

- Lots of these bugs in the wild
  - Still new ones appearing!
  - Too restrictive to forbid variable format strings

Using Tainted and Untainted

- Add qualifier annotations

```
int printf(untainted char *fmt, ...)
taintedchar *getenv(const char *)
```

tainted = may be controlled by adversary

untainted = must not be controlled by adversary

Subtyping

```
void f(tainted int);
untainted int a;
f(a);
```

OK

```
void g(untainted int);
tainted int b;
f(b);
```

Error

f accepts tainted or untainted data

g accepts only untainted data

untainted $\le$ tainted

tainted $\not\le$ untainted

untainted $<$ tainted

Demo of cqual

http://www.cs.umd.edu/~jfoster

The Plan

- The Nice Theory
- The Icky Stuff in C
- Data Race Detection?

A Simple Language

- We'll add type qualifiers to lambda calculus
  - …with a few extra constructs
  - Same approach works for other languages (like C)

e ::= x | n | true | false | if e then e else e
  | \x:t.e| ee

t ::= int | bool | t→t

Type Qualifiers

- Let Q be the set of type qualifiers
  - Assumed to be chosen in advance and fixed
  - E.g., Q = {tainted, untainted}

- Then the *qualified types* are just
  - $qt ::= int^{Q} \mid bool^{Q} \mid qt \rightarrow^{Q} qt$

- Hypothetical examples
  - int^{pos} positive integer
  - $bool^{tainted}$ tainted boolean
  - $qt \rightarrow^{frees} qt$ function that might free memory

Abstract Syntax with Qualifiers

e ::= x | n | true | false | if e then e else e |
\x:qt.e| ee| annot(Q, e) | check(Q, e)

- annot(Q, e) = "expression e has qualifier Q"
  - Will sometimes write as superscript
- check(Q, e) = "fail if e does not have qualifier Q"
  - Checks only the top-level qualifier

- Examples:
  - let fread = \x:qt. …annot(tainted, …)
  - let printf = \x:qt. check(untainted, x); …

Typing Rules: Qualifier Introduction

- Newly-constructed values have "bare" types

$$\frac{}{G \mid\text{--}\ n : int}$$

$$\frac{}{G \mid\text{--}\ true : bool} \qquad \frac{}{G \mid\text{--}\ false : bool}$$

- Annotation adds an outermost qualifier

$$\frac{G \mid\text{--}\ e1 : s}{G \mid\text{--}\ annot(Q, e) : Q\ s}$$

Typing Rules: Qualifier Elimination

- By default, discard qualifier at destructors

$$\frac{G \mid\!\text{--}\ e1 : bool^{Q} \quad G \mid\!\text{--}\ e2 : qt \quad G \mid\!\text{--}\ e3 : qt}{G \mid\!\text{--}\ \text{if } e1 \text{ then } e2 \text{ else } e3 : qt}$$

- Use check() if you want to do a test

$$\frac{G \mid\!\text{--}\ e : Q\ s}{G \mid\!\text{--}\ check(Q, e) : Q\ s}$$

- Ex: if (check(untainted, b) then … else …

Subtyping

- Our example used *subtyping*
  - If anyone expecting a T can be given an S instead, then S is a *subtype* of T.
  - Allows untainted to be passed to tainted positions
  - I.e., check(tainted, annot(untainted, 42)) should typecheck

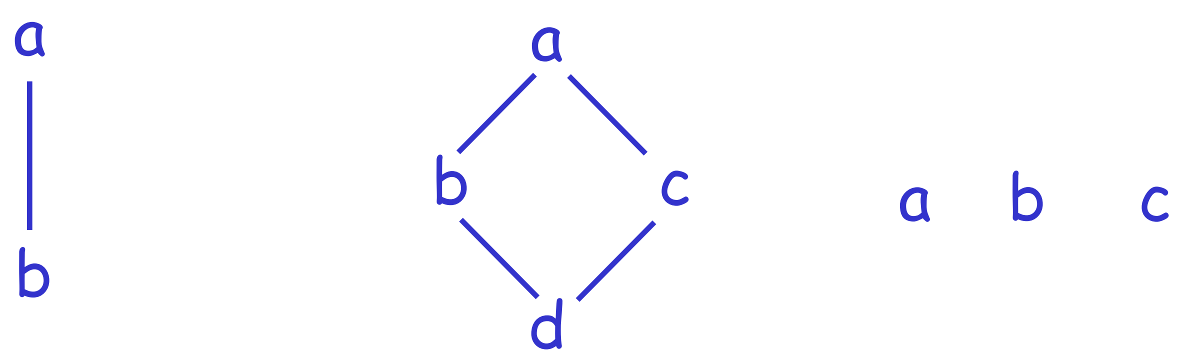
- How do we add that to our system?

Partial Orders

- Qualifiers Q come with a partial order $\leq$:
  - $q \leq q$ (reflexive)
  - $q \leq p, p \leq q \Rightarrow q = p$ (anti-symmetric)
  - $q \leq p, p \leq r \Rightarrow q \leq r$ (transitive)
- Qualifiers introduce subtyping

- In our example:
  - untainted < tainted

Example Partial Orders



2-point lattice

Discrete partial order

- Lower in picture = lower in partial order
- Edges show ≤relations

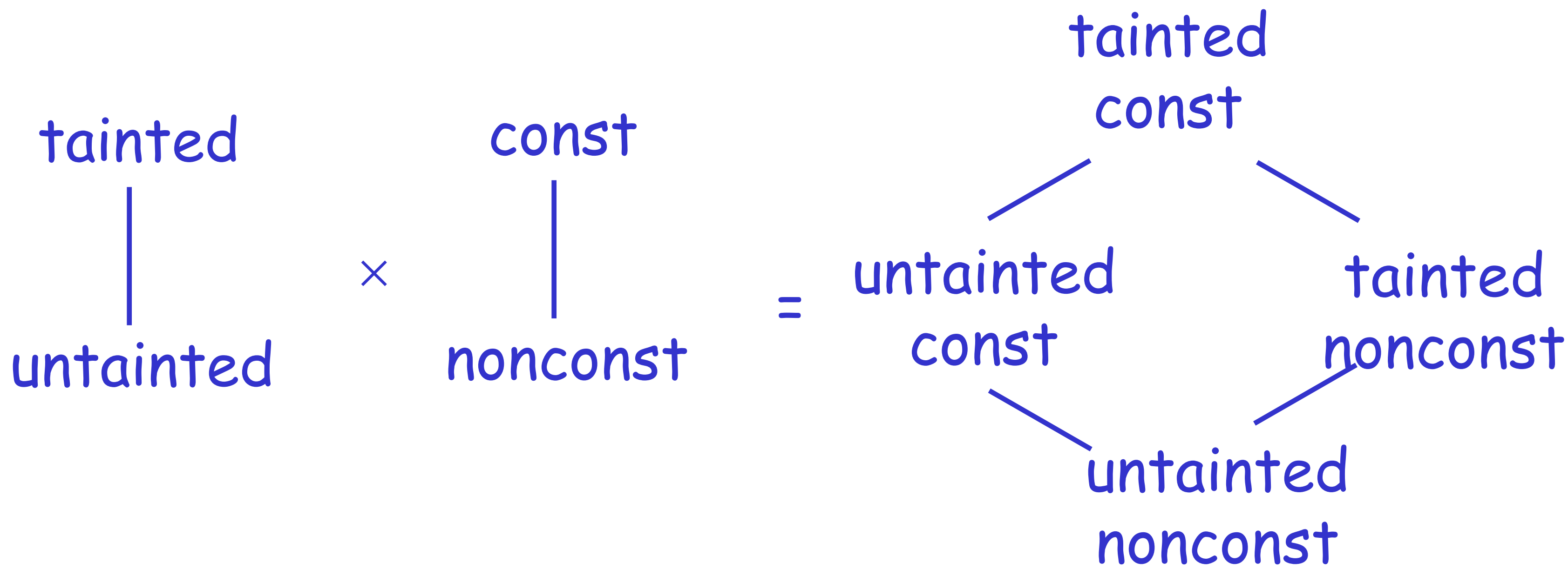
Combining Partial Orders

- Let $(Q_1, \leq_1)$ and $(Q_2, \leq_2)$ be partial orders
- We can form a new partial order, their cross-product:

$$(Q_1, \leq_1) \times (Q_2, \leq_2) = (Q, \leq)$$

where
  - $Q = Q_1 \times Q_2$
  - $(a, b) \leq (c, d)$ if $a \leq_1 c$ and $b \leq_2 d$

Example



- Makes sense with orthogonal sets of qualifiers
  - Allows us to write type rules assuming only one set of qualifiers

Extending the Qualifier Order to Types

$$\frac{Q \le Q'}{\text{bool}^{Q} \le \text{bool}^{Q'}} \qquad \frac{Q \le Q'}{\text{int}^{Q} \le \text{int}^{Q'}}$$

- Add one new rule *subsumption* to type system

$$\frac{G \mid\text{--}\ e : qt \quad qt \le qt'}{G \mid\text{--}\ e : qt'}$$

- Means: If any position requires an expression of type qt', it is safe to provide it a subtype qt

Use of Subsumption

$$\cfrac{\cfrac{\cfrac{\vdash 42 : int}{\vdash annot(untainted, 42) : untainted\ int} \quad untainted \leq tainted}{\vdash annot(untainted, 42) : tainted\ int}}{\vdash check(tainted, annot(untainted, 42)) : tainted\ int}$$

Subtyping on Function Types

- What about function types?

$$\frac{?}{qt1 \to^{Q} qt2 \leq qt1' \to^{Q'} qt2'}$$

- 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 $\underbrace{qt1' \to^{Q'} qt2'}_{g} \leq \underbrace{qt1 \to^{Q} qt2}_{f}$?
- Return type:
  - We are expecting qt2 (f's return type)
  - So we can only return *at most* qt2
  - $qt2' \leq qt2$
- Example: A function that returns tainted can be replaced with one that returns untainted

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

- When is $qt1' \rightarrow^{Q'} qt2' \leq qt1 \rightarrow^{Q} qt2$?

  $\underbrace{qt1' \rightarrow^{Q'} qt2'}_{g} \leq \underbrace{qt1 \rightarrow^{Q} qt2}_{f}$

- Argument type:
  - We are supposed to accept qt1 (f's argument type)
  - So we must accept *at least* qt1
  - $qt1 \leq qt1'$
- Example: A function that accepts untainted can be replaced with one that accepts tainted

Subtyping on Function Types

$$\frac{qt1' \leq qt1 \quad qt2 \leq qt2' \quad Q \leq Q'}{qt1 \rightarrow^{Q} qt2 \leq qt1' \rightarrow^{Q'} qt2'}$$

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

Dynamic Semantics with Qualifiers

- Operational semantics tags values with qualifiers
  - $v ::= n^Q \mid true^Q \mid false^Q \mid \backslash^Q x{:}qt.e$

- Evaluation rules same as usual, carrying the qualifiers along, e.g.,

$$\frac{}{\text{if } true^Q \text{ then } e1 \text{ else } e2 \rightarrow e1}$$

Dynamic Semantics with Qualifiers (cont'd)

- One new rule checks a qualifier:

$$\frac{Q' \leq Q}{\text{check}(Q, v^{Q'}) \rightarrow v}$$

  - Evaluation at a check can continue only if the qualifier matches what is expected
    - Otherwise the program gets *stuck*
  - (Also need rule to evaluate under a check)

Soundness

- We want to prove
  - Preservation: Evaluation preserves types
  - Progress: Well-typed programs don't get stuck
- Proof: Exercise
  - See if you can adapt standard proofs to this system
  - (Not too much work; really just need to show that check doesn't get stuck)

Updateable References

- Our language is missing *side-effects*
  - There's no way to write to memory
  - Recall that this doesn't limit expressiveness
    - But side-effects sure are handy

Language Extension

- We'll add ML-style references
  - $e ::= \ldots \mid ref^Q e \mid !e \mid e := e$
    - $ref^Q e$ -- Allocate memory and set its contents to e
      - Returns memory location
      - Q is qualifier on pointer (not on contents)
      - (Wrote annotas superscript, for convenience)
    - $!e$ -- Return the contents of memory location e
    - $e1 := e2$ -- Update $e1$'s contents to contain $e2$

  - Things to notice
    - No null pointers (memory always initialized)
    - No mutable local variables (only pointers to heap allowed)

Static Semantics

- Extend type language with references:
  - qt ::= … | ref^Q qt
    - Note: In ML the ref appears on the right

$$\frac{G \mid\!-- \ e : qt}{G \mid\!-- \ ref^Q\ e : ref^Q\ qt}$$

$$\frac{G \mid\!-- \ e : ref^Q\ qt}{G \mid\!-- \ !e : qt}$$

$$\frac{G \mid\!-- \ e1 : ref^Q\ qt \qquad G \mid\!-- \ e2 : qt}{G \mid\!-- \ e1 := e2 : qt}$$

Subtyping References

- The *wrong* rule for subtyping references is

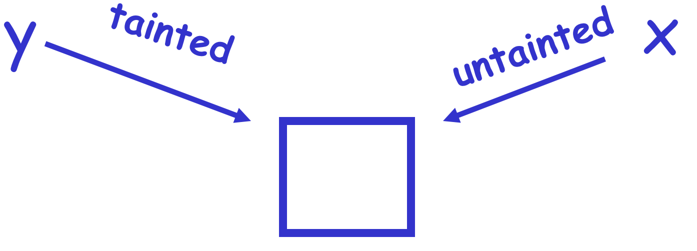
$$\frac{Q \leq Q' \quad qt \leq qt'}{ref^{Q}\ qt \leq ref^{Q'}\ qt'}$$

- Counterexample

```
let x = ref 0^untainted in   /*x: ref (int^untainted)  */
  let y = x in         /* y : ref (int^tainted)  */
y := 3^tainted;
check(untainted, !x)                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(); // forbidden at runtime
  - a[0].foo(); // …so this can't happen

Solution #2: Purely Static Approach

- Reason from rules for functions
  - A reference is like an object with two methods:
    - get : unit $\rightarrow$ qt
    - set : qt $\rightarrow$ unit
  - Notice that qt occurs both co- and contravariantly
- The right rule:

$$\frac{Q \leq Q' \quad qt \leq qt' \quad qt' \leq qt}{ref^{Q}\ qt \leq ref^{Q'}\ qt'} \quad \text{or} \quad \frac{Q \leq Q' \quad qt = qt'}{ref^{Q}\ qt \leq ref^{Q'}\ qt'}$$

Soundness

- We want to prove
  - Preservation: Evaluation preserves types
  - Progress: Well-typed programs don't get stuck
- Can you prove it with updateable references?
  - Hint: You'll need a stronger induction hypothesis
    - You'll need to reason about types in the store
    - E.g., so that if you retrieve a value out of the store, you know what type it has

Type Qualifier Inference

- Recall our motivating example
  - We gave a legacy C program that had *no information* about qualifiers
  - We added signatures *only* for the standard library functions
  - Then we checked whether there were any contradictions

- This requires *type qualifier inference*

Type Qualifier Inference Statement

- Given a program with
  - Qualifier annotations
  - Some qualifier checks
  - And no other information about qualifiers
- Does there exist a valid typing of the program?

- We want an algorithm to solve this problem

First Problem: Subsumption Rule

$$\frac{G \mid\!-\!- e : qt \quad qt \leq qt'}{G \mid\!-\!- e : qt'}$$

- 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

$$\frac{G \mid\text{--}\ e1 : qt' \rightarrow qt'' \qquad G \mid\text{--}\ e2 : qt \qquad qt \leq qt'}{G \mid\text{--}\ e1\ e2 : qt''}$$

$$\frac{G \mid\text{--}\ e : Q'\ s \qquad Q' \leq Q}{G \mid\text{--}\ check(Q, e) : Q\ s}$$

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

Second Problem: Assumptions

- Let's take a look at the rule for functions:

$$\frac{G, x{:}qt1 \mid\!-\!- e : qt2' \qquad qt2' \leq qt2}{G \mid\!-\!- \backslash x{:}qt1.e : qt1 \rightarrow qt2}$$

- There's a problem with applying this rule
  - We're assuming that we're given the argument type qt1 and the result type qt2
  - But in the problem statement, we said we only have annotations and checks

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

Why Do We Want Qualifier Inference?

- Because our programs weren't written with qualifiers in mind
  - They don't have qualifiers in their type annotations
  - In particular, functions don't list qualifiers for their arguments
- Because it's less work for the programmer
  - …but it's harder to understand when a program doesn't type check

Adding Fresh Qualifiers

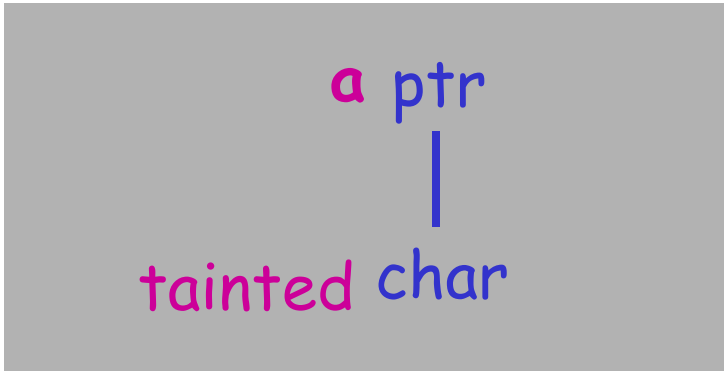
- We'll add qualifier variables a, b, c, … to our set of qualifiers
  - Letters closer to p, q, r will stand for constants
  - Inference = map variables to constants so program type checks
- Define fresh : t→ qt as
  - fresh(int) = int^a
  - fresh(bool) = $bool^a$
  - fresh(ref t) = ref^a fresh(t)
  - fresh(t1→t2) = fresh(t1) $\to^a$ fresh(t2)
    - Where a is fresh

Rule for Functions

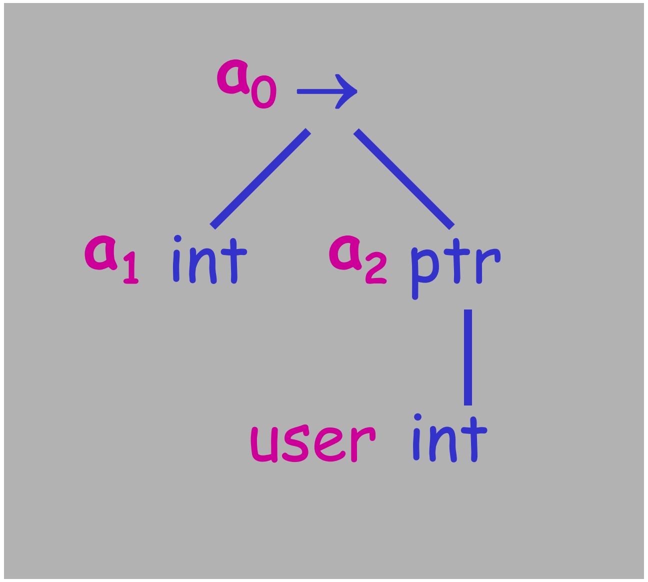
$$\frac{G, x{:}qt1 \mid\!\!-\!- e : qt2' \quad qt2' \leq qt2 \quad qt1 = fresh(t1)}{G \mid\!\!-\!- \backslash x{:}t1.e : qt1 \rightarrow qt2}$$

A Picture of Fresh Qualifiers

ptr(tainted char)



int → user ptr(int)



Where Are We?

- A syntax-directed system
  - For each expression, clear which rule to apply
- Constant qualifiers
- Variable qualifiers
  - Want to find a valid assignment to constant qualifiers
- Constraints $qt \leq qt'$ and $Q \leq Q'$
  - These restrict our use of qualifiers
  - These will limit solutions for qualifier variables

Qualifier Inference Algorithm

1. Apply syntax-directed type inference rules
   - This generates fresh unknowns and constraints among the unknowns
2. Solve the constraints
   - Either compute a *solution*
   - Or fail, if there is no solution
     - Implies the program has a type error
     - Implies the program *may* have a bug

Solving Constraints: Step 1

- Constraints of the form $qt \leq qt'$ and $Q \leq Q'$
  - $qt ::= int^Q \mid bool^Q \mid qt \to^Q qt \mid ref^Q\ qt$
- Solve by simplifying
  - Can read solution off of simplified constraints
- We'll present algorithm as a rewrite system
  - $S \Longrightarrow S'$ means constraints S rewrite to (simpler) constraints S'

Solving Constraints: Step 1

- $S + \{ int^{Q} \le int^{Q'} \} \Rightarrow S + \{ Q \le Q' \}$
- $S + \{ bool^{Q} \le bool^{Q'} \} \Rightarrow S + \{ Q \le Q' \}$
- $S + \{ qt1 \to^{Q} qt2 \le qt1' \to^{Q'} qt2' \} \Rightarrow$

  $S + \{ qt1' \le qt1 \} + \{ qt2 \le qt2' \} + \{ Q \le Q' \}$
- $S + \{ ref^{Q}\ qt1 \le ref^{Q'}\ qt2 \} \Rightarrow$

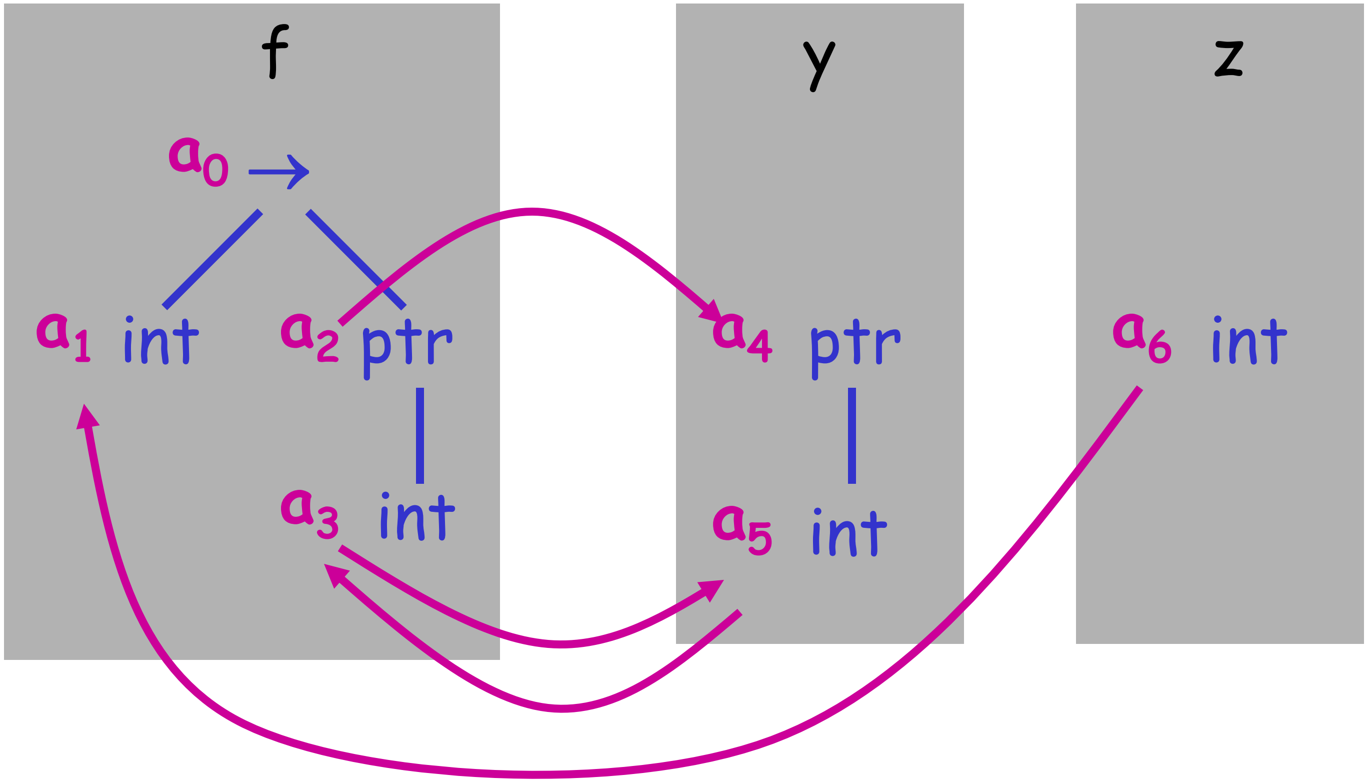
  $S + \{ qt1 \le qt2 \} + \{ qt2 \le qt1 \} + \{ Q \le Q' \}$
- S + { mismatched constructors } ==> error
  - Can't happen if program correct w.r.t. std types

Solving Constriants: Step 2

- Our type system is called a *structural subtyping system*
  - If $qt \leq qt'$, then qt and qt' have the same shape
- When we're done with step 1, we're left with constraints of the form $Q \leq Q'$
  - Where either of Q, Q' may be an unknown
  - This is called an *atomic subtyping system*
  - That's because qualifiers don't have any "structure"

Constraint Generation

ptr(int) f(x : int) = { ... }　　　　y := f(z)

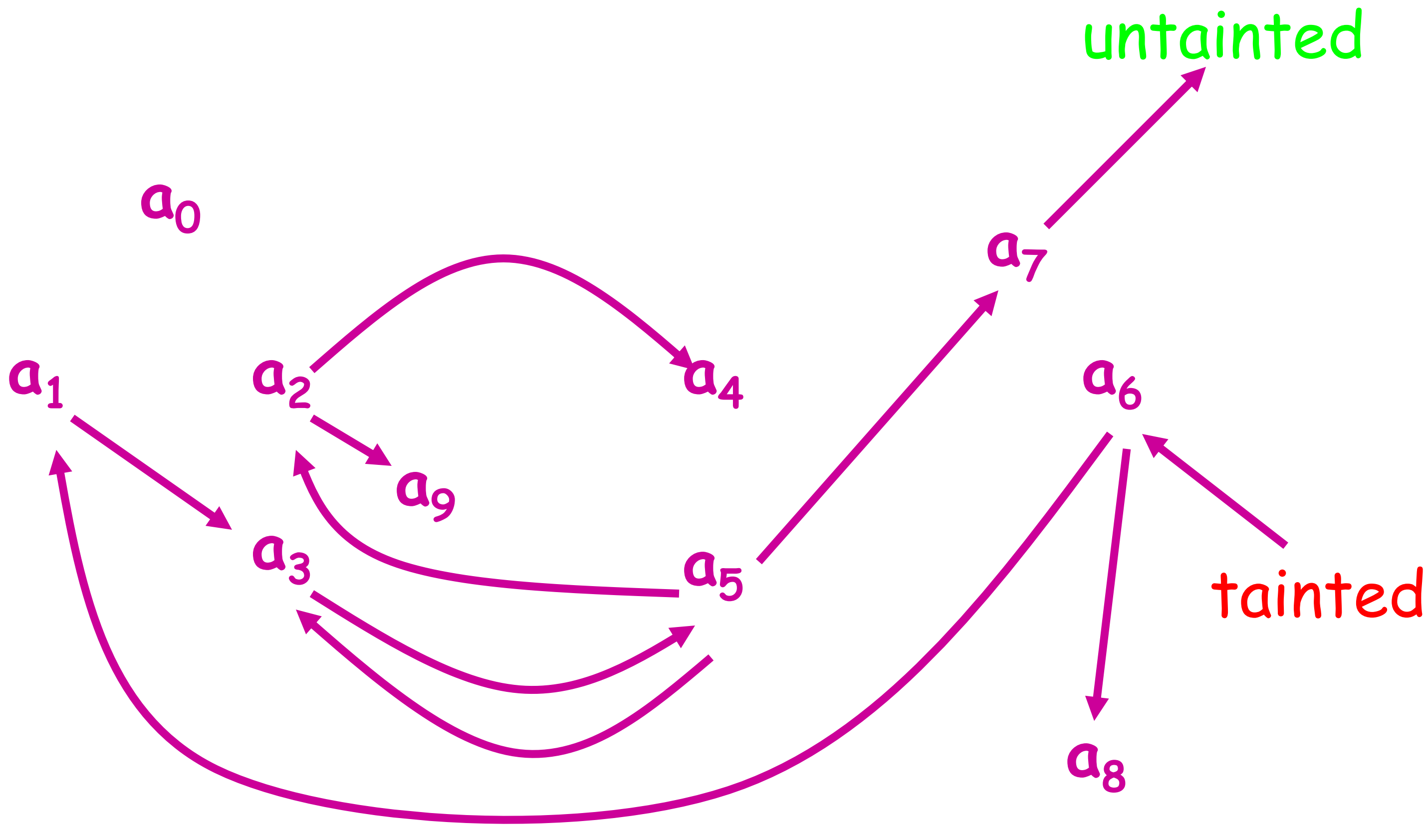


$a_6 \leq a_1$

$a_2 \leq a_4$

$a_3 = a_5$

Constraints as Graphs



$a_6 \leq a_1$

$a_2 \leq a_4$

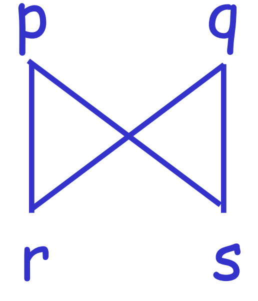
$a_3 = a_5$

⋮

Some Bad News

- Solving atomic subtyping constraints is NP-hard in the general case

- The problem comes up with some really weird partial orders



But That's OK

- These partial orders don't seem to come up in practice
  - Not very natural

- Most qualifier partial orders have one of two desirable properties:
  - They either always have *least upper bounds* or *greatest lower bounds* for any pair of qualifiers

Lubs and Glbs

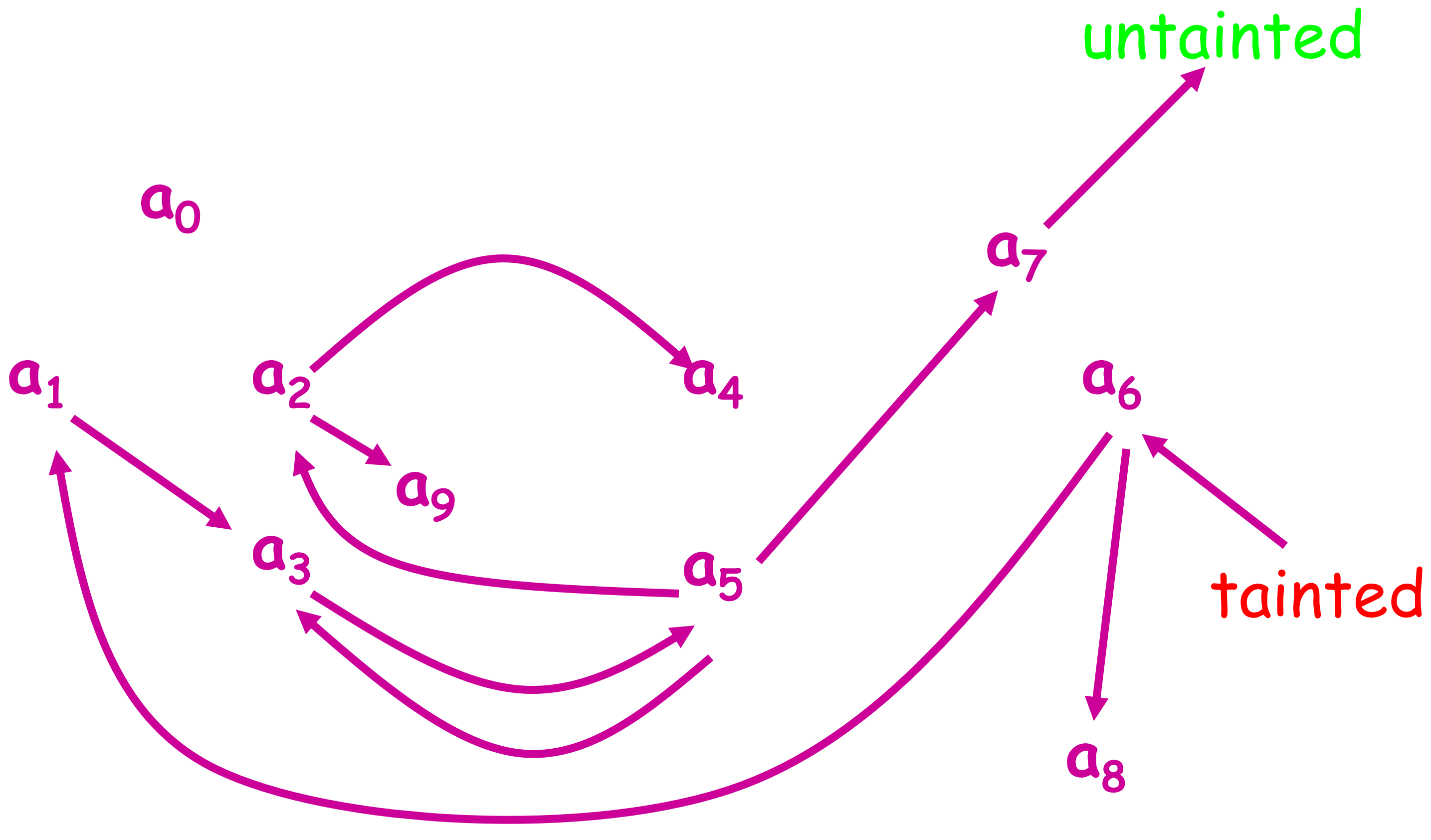
- lub = Least upper bound
  - p lub q = r such that
    - $p \leq r$ and $q \leq r$
    - If $p \leq s$ and $q \leq s$, then $r \leq s$
- glb = Greatest lower bound, defined dually
- lub and glb may not exist

Lattices

- A *lattice* is a partial order such that lubs and glbs always exist
- If Q is a lattice, we can check satisfiability via transitive closure
  - Apply the following rule exhaustively:
    - S + { Q ≤ Q′, Q′≤ Q″ } +==> S + { Q ≤ Q″ }
  - Error if we ever derived tainted ≤ untainted
    - Or a similar unsatisfiable constraint

Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



$a_6 \leq a_1$

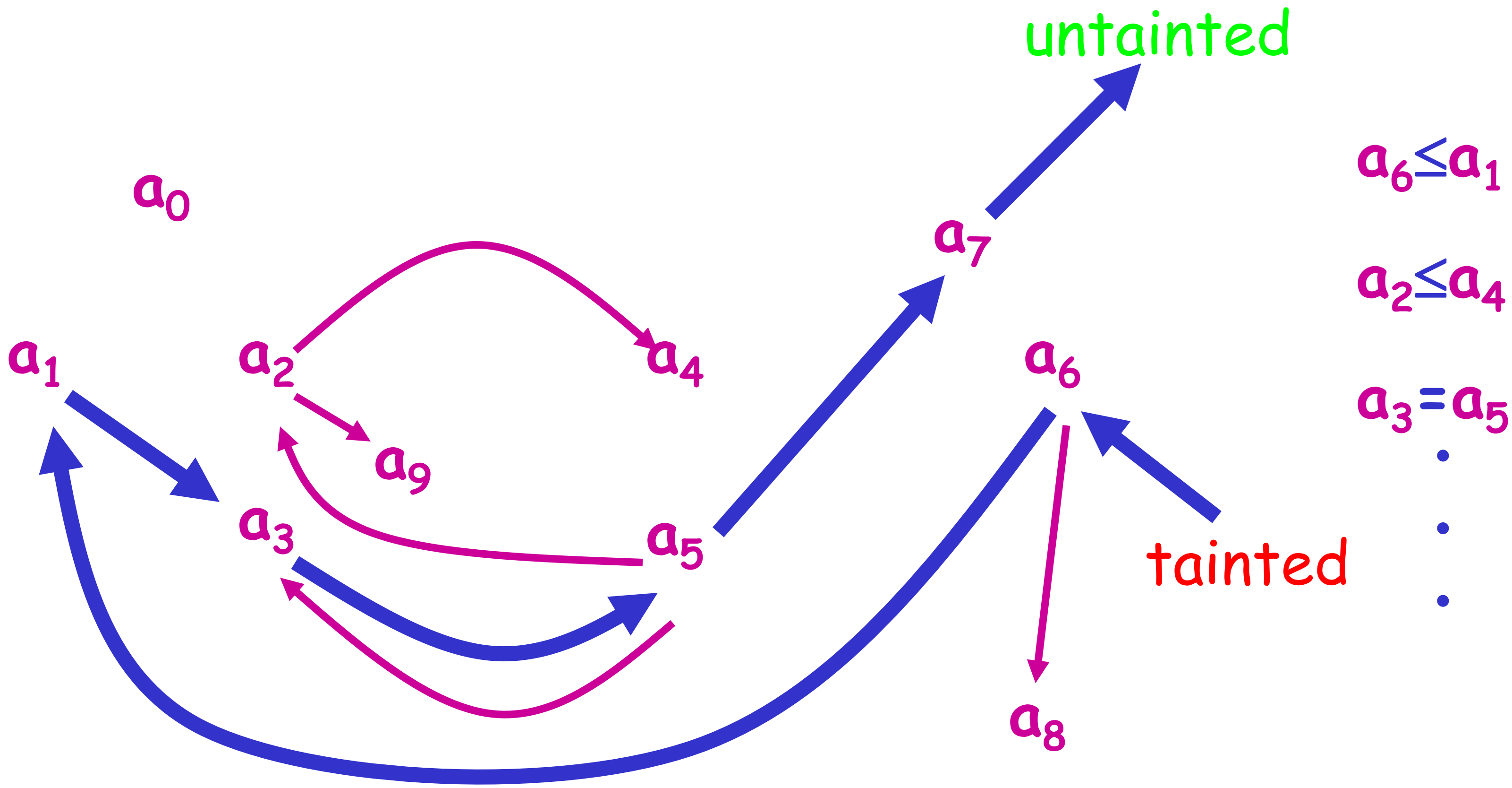
$a_2 \leq a_4$

$a_3 = a_5$

⋮

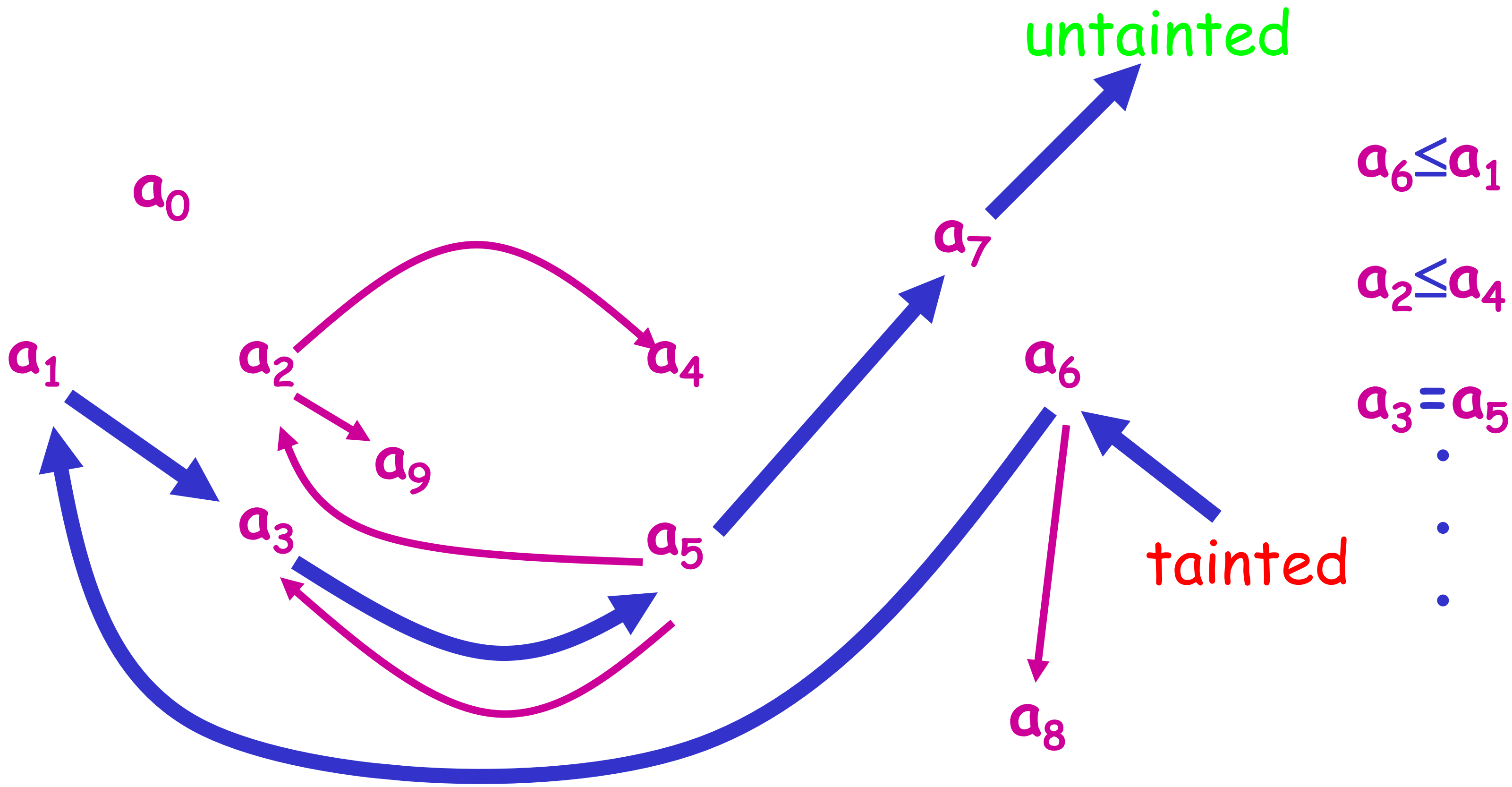
Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



Satisfiability via Graph Reachability

tainted $\leq a_6 \leq a_1 \leq a_3 \leq a_5 \leq a_7 \leq$ untainted



Satisfiability in Linear Time

- Initial program of size n
  - Fixed set of qualifiers tainted, untainted, ...
- Constraint generation yields O(n) constraints
  - Recursive abstract syntax tree walk
- Graph reachability takes O(n) time
  - Works for semi-lattices, discrete p.o., products

Limitations of Subtyping

- Subtyping gives us a kind of *polymorphism*
  - A *polymorphic* type represents multiple types
  - In a subtyping system, qt represents qt and all of qt's subtypes

- As we saw, this flexibility helps make the analysis more precise
  - But it isn't always enough…

Limitations of Subtype Polymorphism

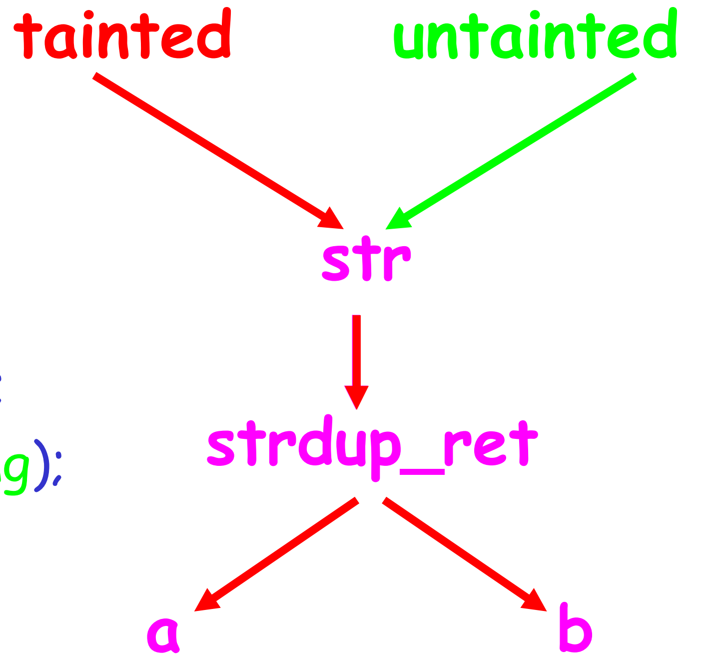
- Consider tainted and untainted again
  - untainted $\leq$ tainted
- Let's look at the identity function
  - let id = \x:int.x
- What qualified types can we infer for id?

Types for id

- let id = \x:int.x
  - tainted int→ tainted int
    - Fine but untainted data passed in becomes tainted
  - untainted int→untainted int
    - Fine but can't pass in tainted data
  - untainted int→ tainted int
    - Not too useful
  - tainted int→ untainted int
    - Impossible

Function Calls and Context-Sensitivity

```
char *strdup(char *str) {
  // return a copy of str
}
char *a = strdup(tainted_string);
char *b = strdup(untainted_string);
```



- All calls to strdup conflated
  - *Monomorphic* or *context-insensitive*

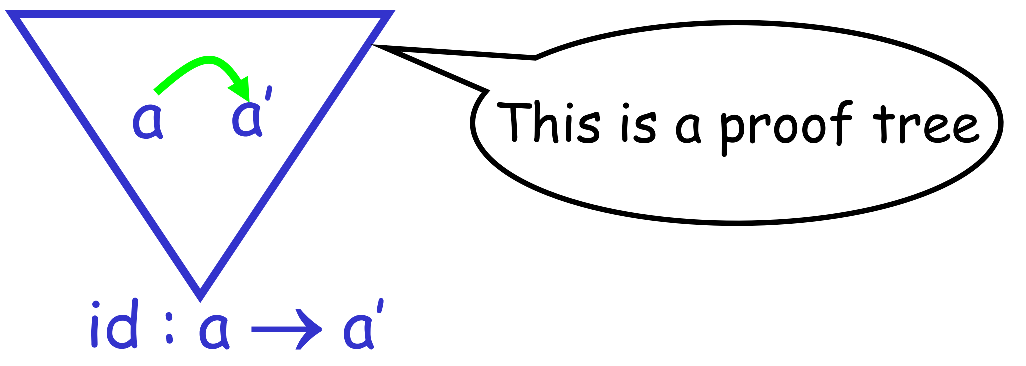
What's Happening Here?

- The qualifier on x appears both covariantly and contravariantly in the type
  - We're stuck

- We need *parametric polymorphism*
  - Considerlet id = \x:int.x

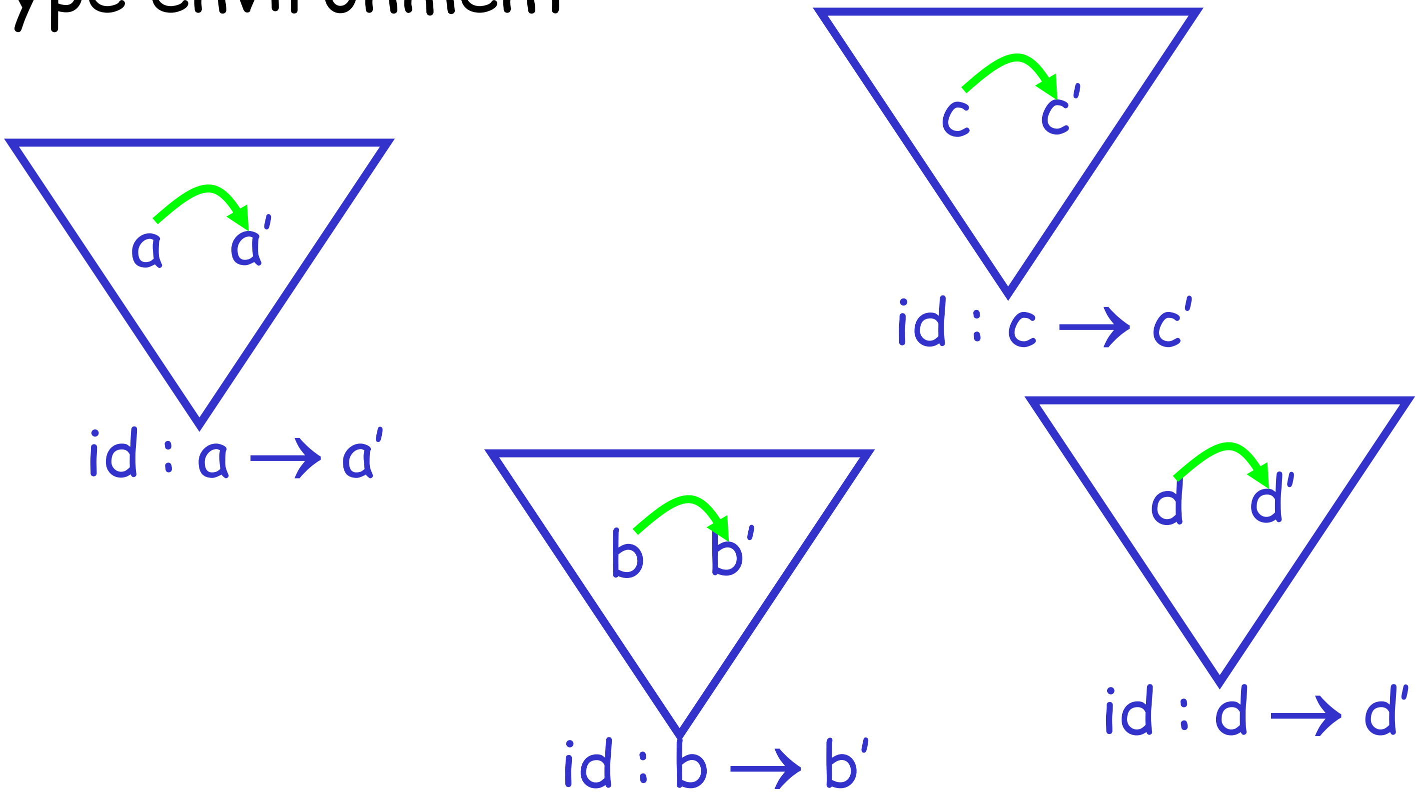
The Observation of Parametric Polymorphism

- Type inference on id yields a proof like this:



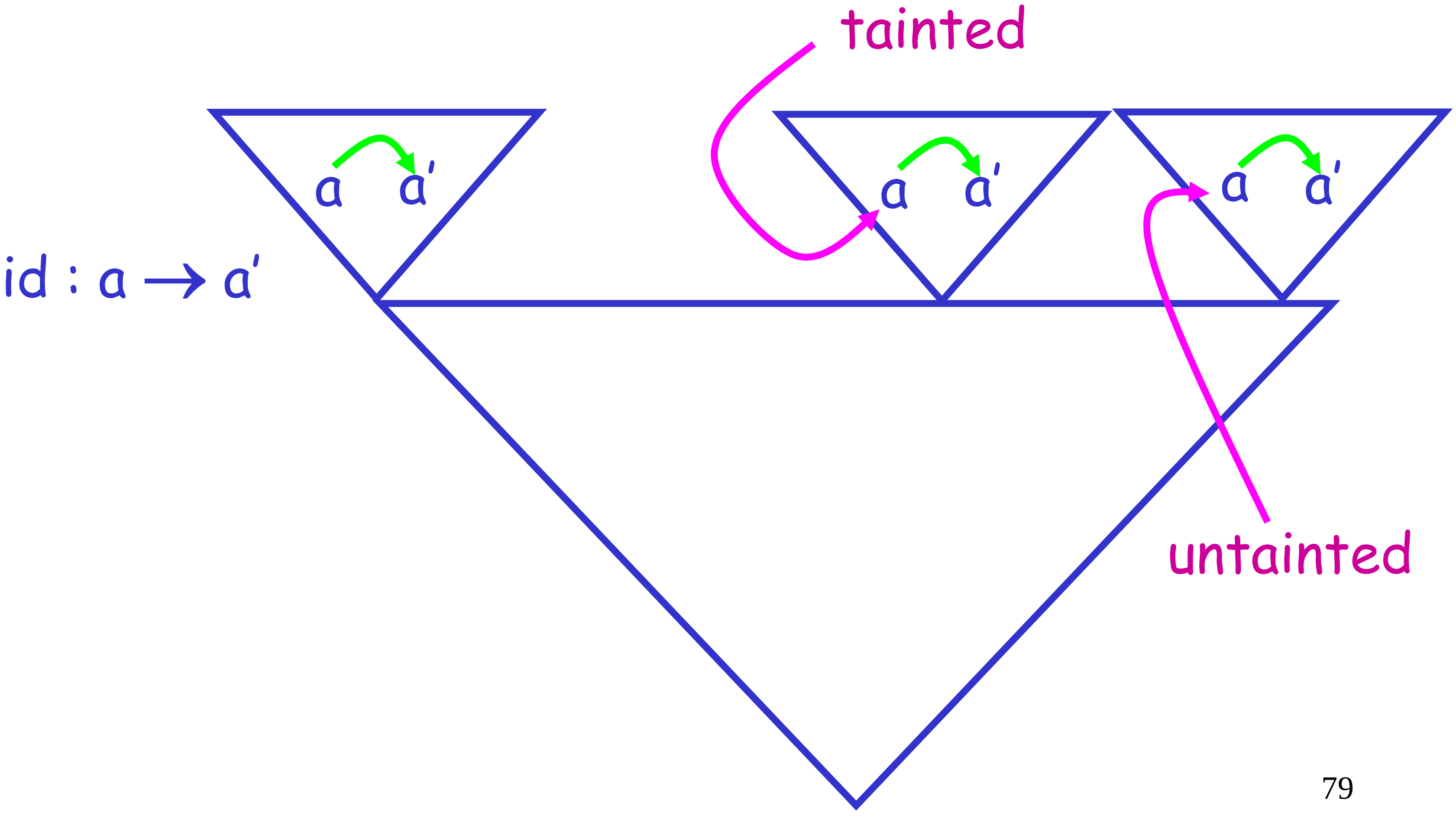
The Observation of Parametric Polymorphism

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



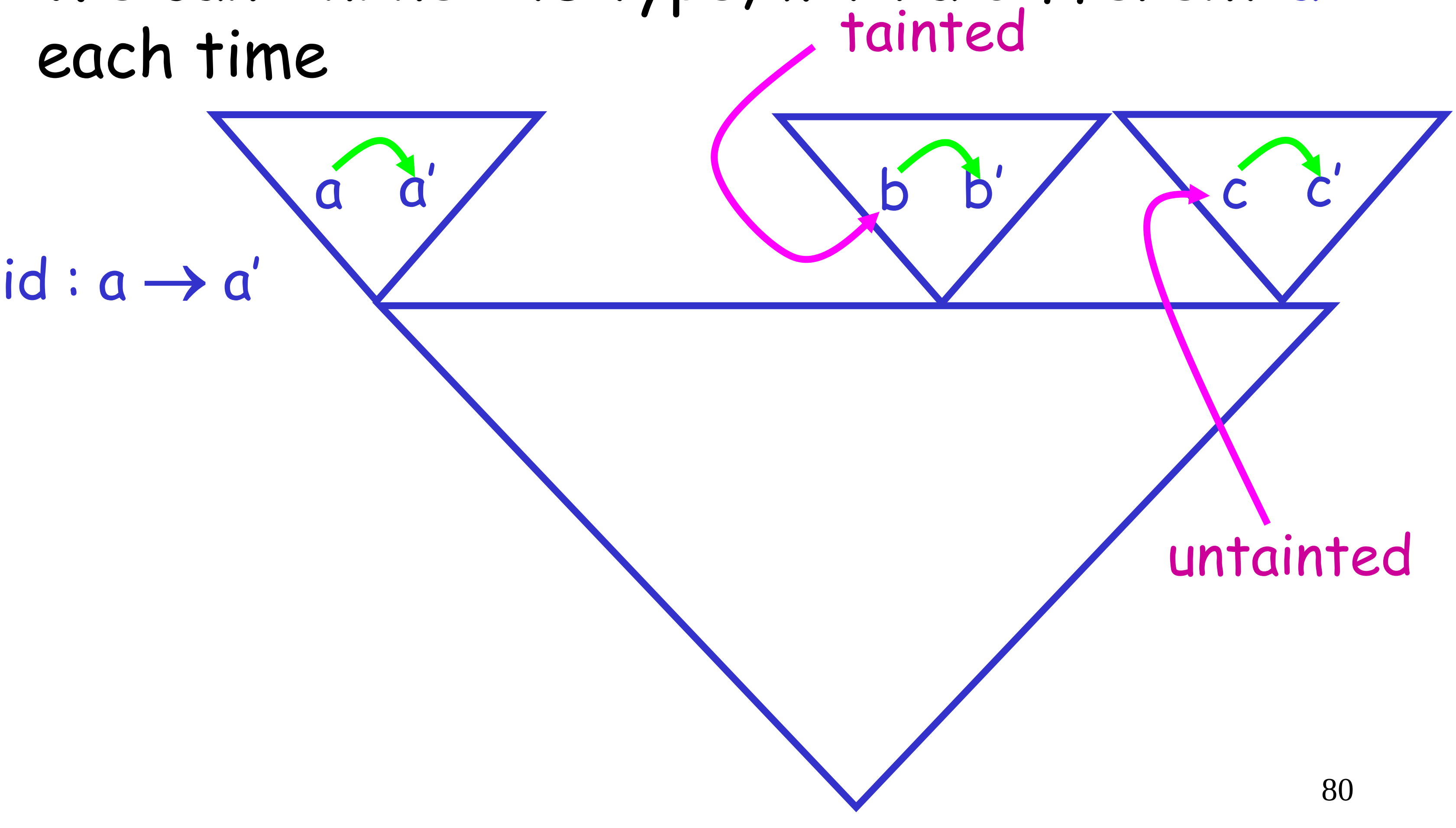
The Observation of Parametric Polymorphism

- 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

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

  - Instantiate at uses

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

Take the original type

Substitute bound vars (arbitrarily)

Polymorphically Constrained Types

- Notice that we inlined not only the *type*, but also the *constraints*

- We need polymorphically constrained types

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

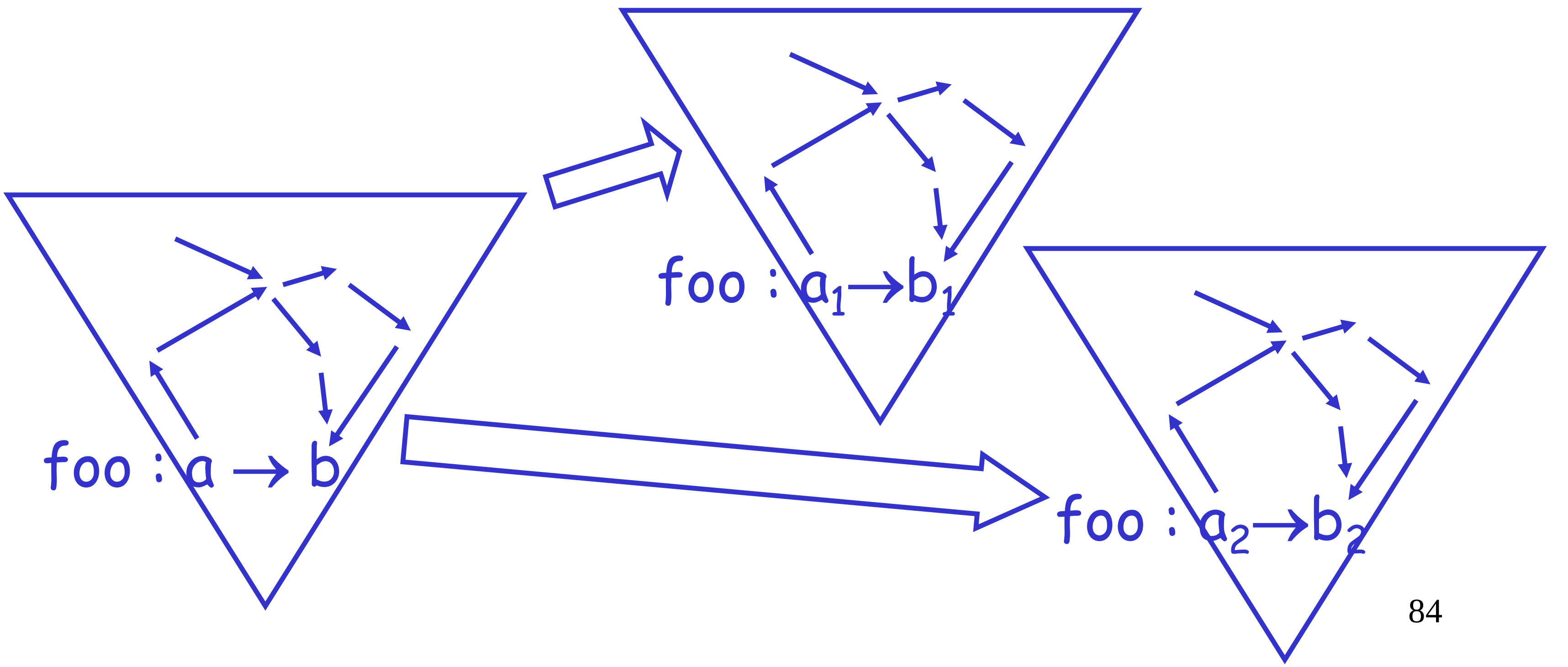
  - For any qualifiers **a** where constraints C hold, x has type qt

Examples of Polymorphically Constrained Types

- int id(int x) { return x; }
  - id : ∀a,b. a int → b int where a ≤ b
- char *strcat(char *s, char *append);
  - strcat : ∀a,b,c. (a char * x b char *) → c char * where b ≤ c, a = c
- void *malloc(size_t size);
  - malloc : ∀a. () → a void * where []

Polymorphically Constrainted Types

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



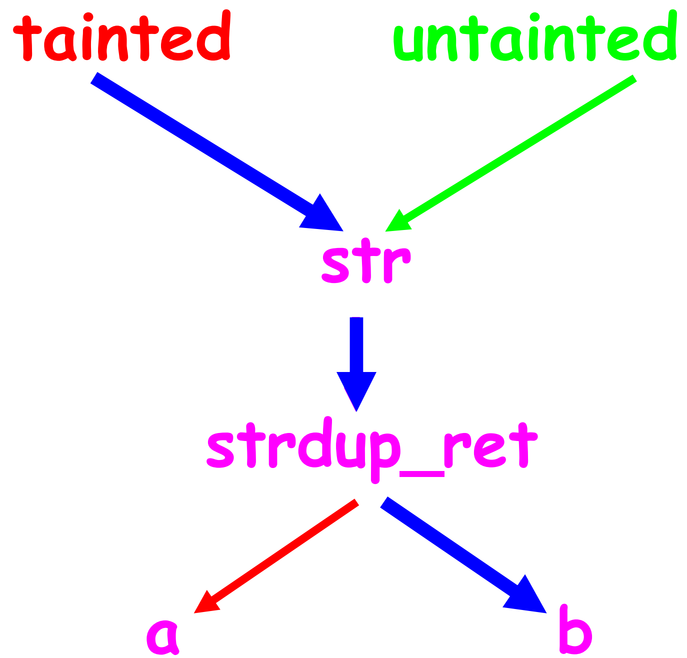
Comparison to Type Polymorphism

- 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 qualifiers
    - That's because we'll apply this to *C*

A Better Solution: CFL Reachability

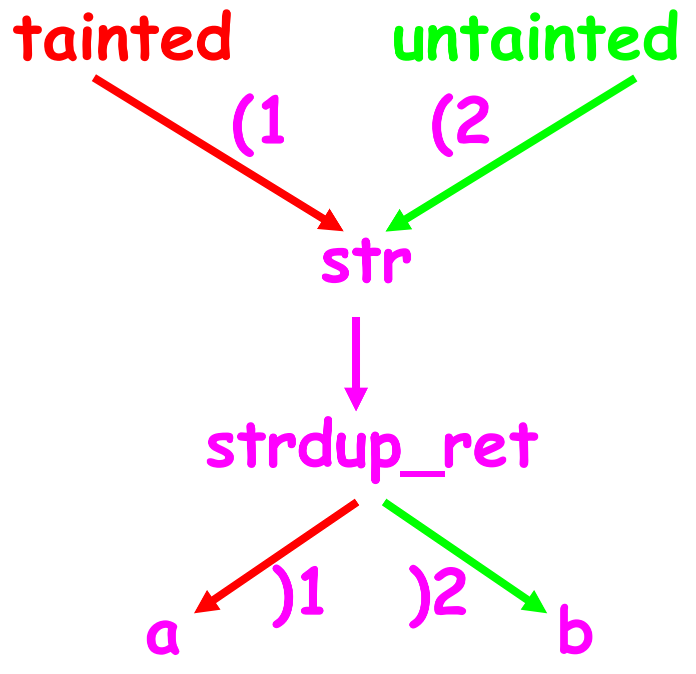
- Can reduce this to another problem
  - Equivalent to the constraint-copying formulation
  - Supports polymorphic recursion in qualifiers
  - It's easy to implement
  - It's efficient: $O(n^3)$
    - Previous best algorithm $O(n^8)$ [Mossin, PhD thesis]
- Idea due to Horwitz, Reps, and Sagiv [POPL'95],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

The Problem Restated: Unrealizable Paths



- No execution can exhibit that particular call/return sequence

Only Propagate Along Realizable Paths

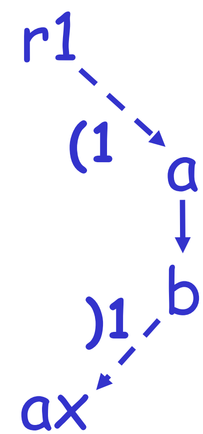


- Add edge labels for calls and returns
  - Only propagate along *valid* paths whose returns balance calls

Parenthesis Edges

- Paren edges represent substitutions
  - id : $\forall$a, b . a $\rightarrow$b where a $\leq$b
  - let x = id^{1} (ref^{r1} 0)
- 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 + flow
  - b $\dashrightarrow^{)1}$ axb instantiated to ax, and b flows to ax
  - r1 $\dashrightarrow^{(1}$ aainstantiated to r1, and r1 flows to a

Type Rule for Instantiation

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

$$\frac{qt = G(f) \qquad qt' = fresh(qt) \qquad qt \xrightarrow{)_i} qt'}{G \mid\!\text{--}\ f_i : qt'}$$

Rules for Propagating Parenthesis Edges

- $S + \{ int^{Q} \; int \xrightarrow{)_i} Q' \} \Longrightarrow S + \{ Q \quad Q' \xrightarrow{)_i} \}$

- $S + \{ int^{Q} \; int \xrightarrow{(_i} Q' \} \Longrightarrow S + \{ Q \quad Q' \xrightarrow{(_i} \}$

Rules for Propagating Parenthesis Edges

- $S + \{\, qt1 \to^{Q} qt2 \quad qt1 \xrightarrow{)_i} \to^{Q'} qt2' \,\} \Longrightarrow$

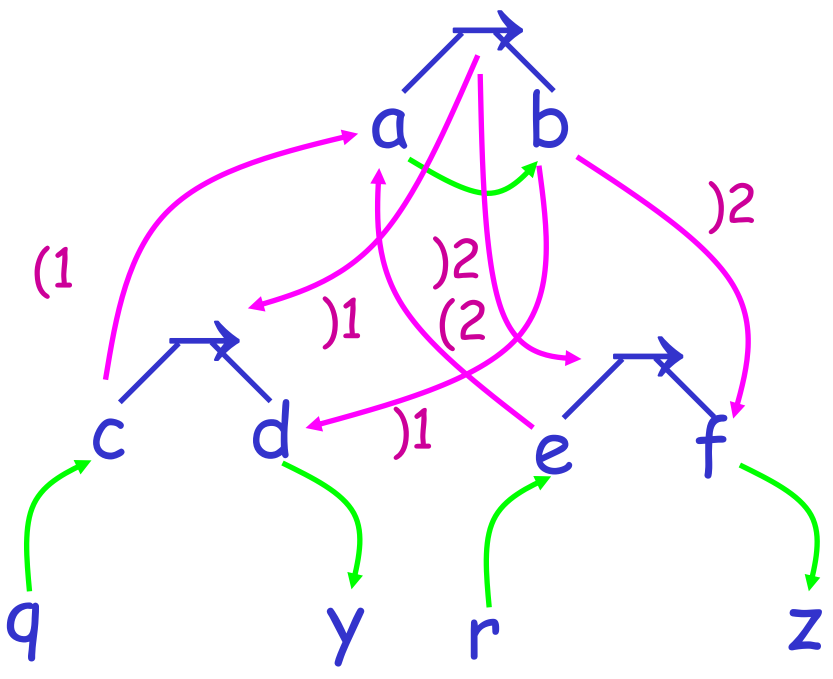
  $S + \{\, qt1' \xrightarrow{(_i} qt1 \,\} + \{\, qt2 \quad qt2' \xrightarrow{)_i} \,\} + \ldots$

- $S + \{\, qt1 \to^{Q} qt2 \quad qt1 \xrightarrow{(_i} \to^{Q'} qt2' \,\} \Longrightarrow$

  $S + \{\, qt1' \xrightarrow{)_i} qt1 \,\} + \{\, qt2 \quad qt2' \xrightarrow{(_i} \,\} + \ldots$

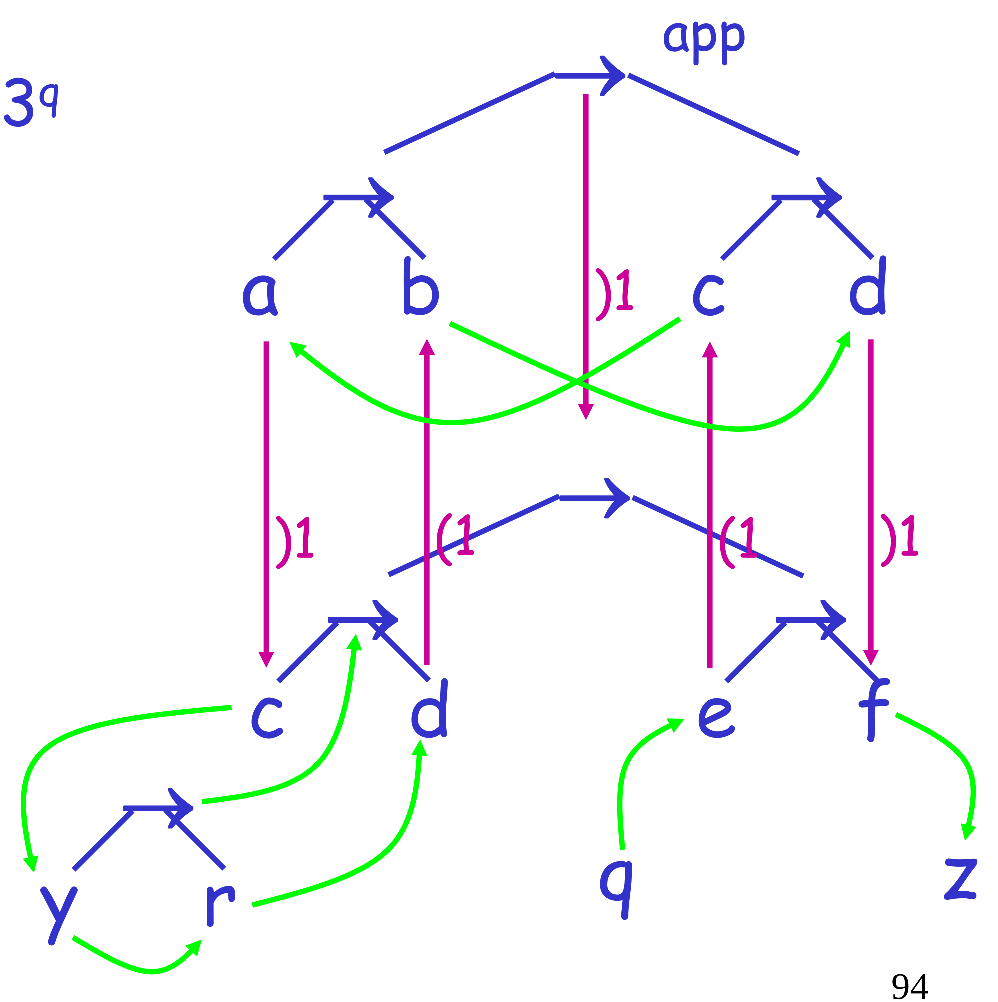
A Simple Example

fun id x = x in
  let y = id_1 3^q
  let z = id_2 4^r



A Higher-Order Example

fun app f x = f x in
 let z = app_1 (\y.y) 3^q



Two Observations

- We *are* doing constraint copying
  - Notice the edge from c to a got "copied" to q 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

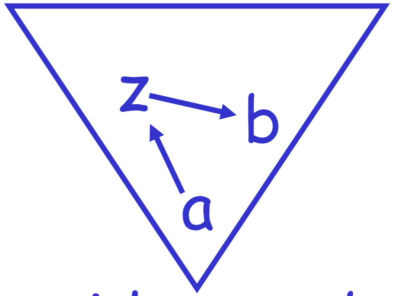
- Also can include other paths, depending on application

Global Variables

- Consider the following identity function

  fun id(x:int):int = z := x; !z

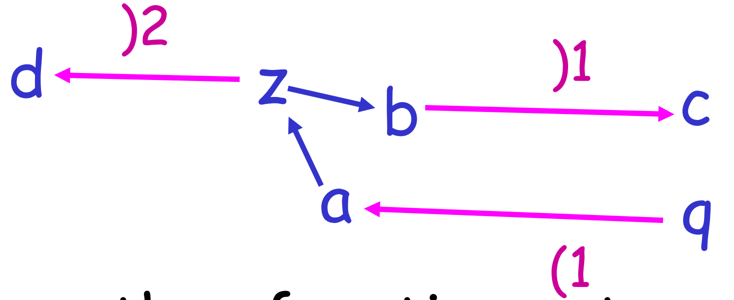
  - Here z is a global variable
- Typing of id, roughly speaking:



$id : a \rightarrow b$

Global Variables

- Suppose we instantiate and apply id to q inside of a function



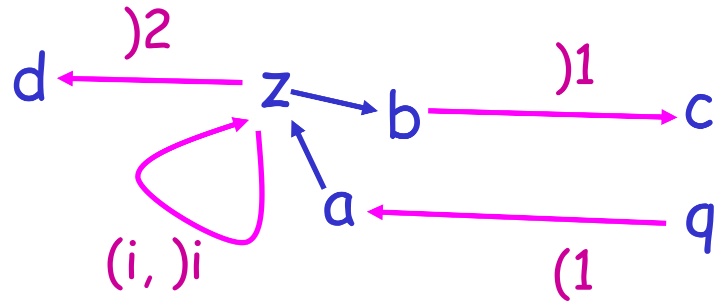
  - And then another function returns z
  - Uh oh! (1)2 is not a valid flow path
    - But q may certainly reach d

Don't Quantify Global Type (Qualifier) Vars

- 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 parens

Typing Rules, Fixed

- Track unquantifiable vars at generalization

$$\frac{A \mid\!-- e1 : qt1 \qquad A, x : (qt1, \mathbf{b}) \mid\!-- e2 : qt2 \qquad \mathbf{b} = fv(A)}{A \mid\!-- \text{let } x = e1 \text{ in } e2 : qt2}$$

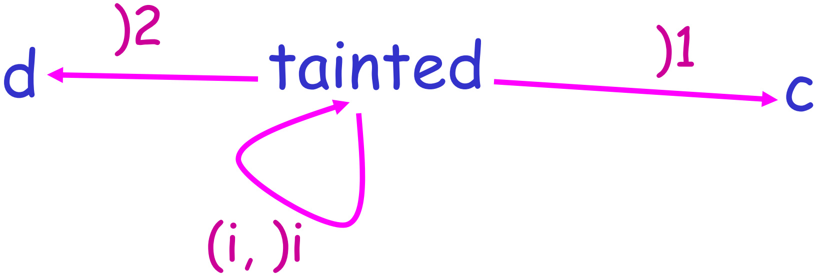
- Add self-loops at instantiation

$$\frac{A(f) = (qt, \mathbf{b}) \qquad qt' = fresh(qt) \quad \xrightarrow{)_i} tt' \qquad \xrightarrow{)_i} \mathbf{bbbb} \xrightarrow{(_i}}{A \mid\!-- f_i : qt'}$$

Qualifier Constants

- Also use self-loops for qualifier constants

```
let taint () =annot(tainted, 42) in
  let c = taint()
  let d = taint()
```



Efficiency

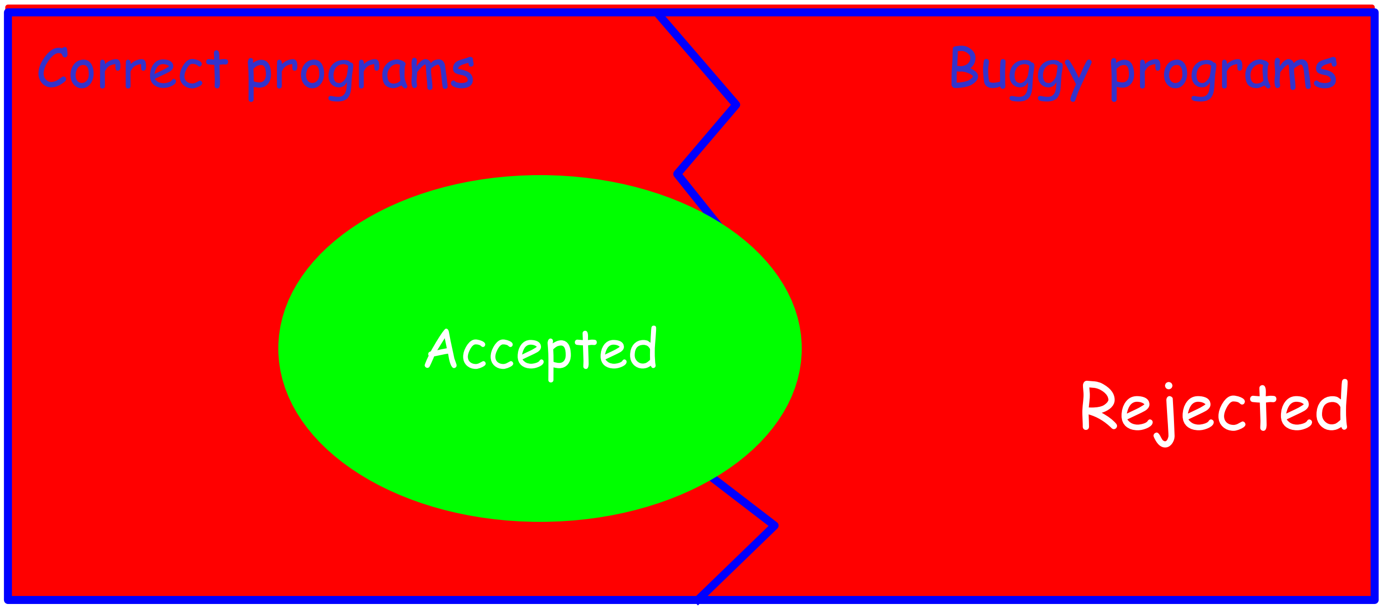
- Constraint generation yields O(n) constraints
  - Same as before
  - Important for scalability
- Context-free language reachability is $O(n^3)$
  - But a few tricks make it practical (not much slowdown in analysis times)
- For more details, see
  - Rehof + Fahndrich, POPL'01

Type Qualifiers:
The Icky Stuff in C

Introduction

- That's all the theory behind this system
  - More complicated system: flow-sensitive qualifiers
  - Not going to cover that here
- Suppose we want to apply this to a language like C
  - It's a little more complicated!

The Reality for "Sound" C Static Analysis



- C has too many undefined behaviors that programmers rely on
- Forbidding all suspicious code yields a useless analysis

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

Multiple Files

- Most applications have multiple source code files
- If we do inference on one file without the others, won't get complete information:

```
extern int t;

x = t;
```

```
$tainted int t = 0;
```

- Problem: In left file, we're assuming t may have any qualifier (we make a fresh variable)

Multiple Files: Solution #1

- Don't analyze programs with multiple files!
- Can use CIL merger from Necula to turn a multi-file app into a single-file app
  - E.g., I have a merged version of the linux kernel, 470432 lines
- Problem: Want to present results to user
  - Hard to map information back to original source

Multiple Files: Solution #2

- Make conservative assumptions about missing files
  - E.g., anything globally exposed may be tainted

- Problem: Very conservative
  - Going to be hard to infer useful types

Multiple Files: Solution #3

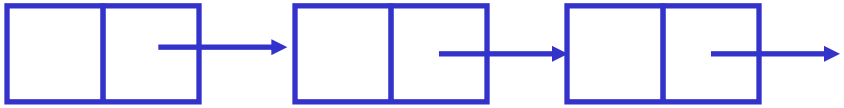
- Give tool all files at same time
  - Whole-program analysis
- Include files that give types to library functions
  - In CQual, we have prelude.cq
- Unify (or just equate) types of globals

- Problem: Analysis really needs to scale

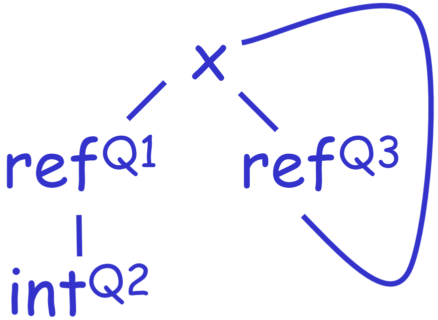
Structures (or Records): Scalability Issues

- One problem: Recursion
  - Do we allow qualifiers on different levels to differ?

```
struct list {
    int elt;
    struct list *next;
}
```



  - Our choice: no (we don't want to do shape analysis)



Structures: Scalability Issues

- Natural design point: All instances of the same struct share the same qualifiers
- This is what we used to do
  - Worked pretty well, especially for format-string vulnerabilities
  - Scales well to large programs (linear in program size)
- Fell down for user/kernel pointers
  - Not precise enough

Structures: Scalability Issues

- Second problem: Multiple Instances
  - Naïvely, each time we see

    `struct inode x;`

    we'd like to make a copy of the type `struct inode` with fresh qualifiers
  - Structure types in C programs are often long
    - `struct inode` in the Linux kernel has 41 fields!
    - Often contain lots of nested structs
  - This won't scale!

Multiple Structure Instances

- Instantiate struct types lazily
  - When we see

    ```
    struct inode x;
    ```

    we make an empty record type for x with a pointer to type struct inode
  - Each time we access a field f of x, we add fresh qualifiers for f to x's type (if not already there)
  - When two instances of the same struct meet, we unify their records
    - This is a heuristic we've found is acceptable

Lazy Field Expansion

a: <empty>

C type

```
struct foo {
  int x;
  char * y;
  ...
}
```

Lazy Field Expansion

a: <empty>

...a.x...

C type

```
struct foo {
  int x;
  char * y;
  ...
}
```

Lazy Field Expansion

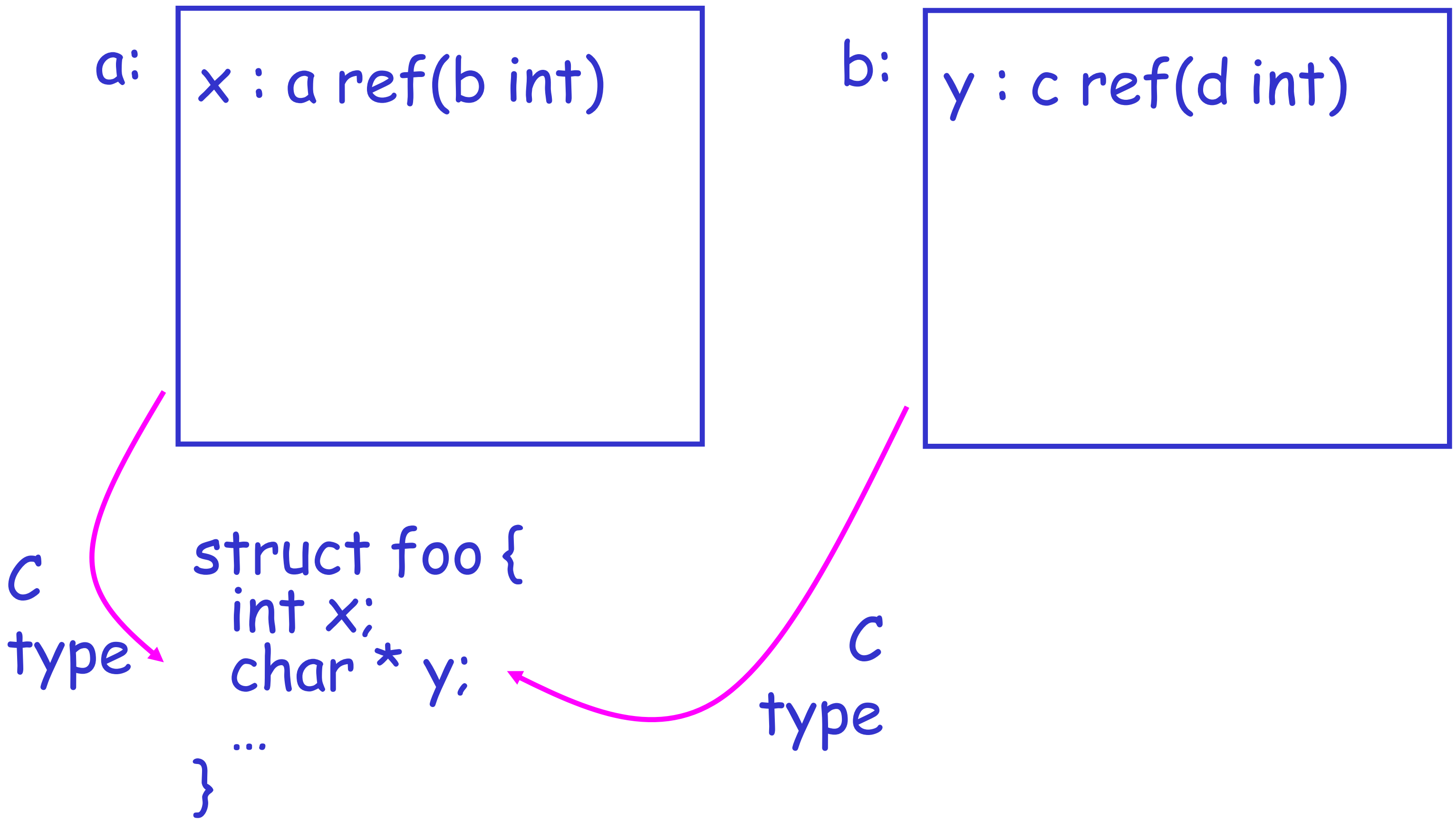
a: x : a ref(b int)

...a.x...

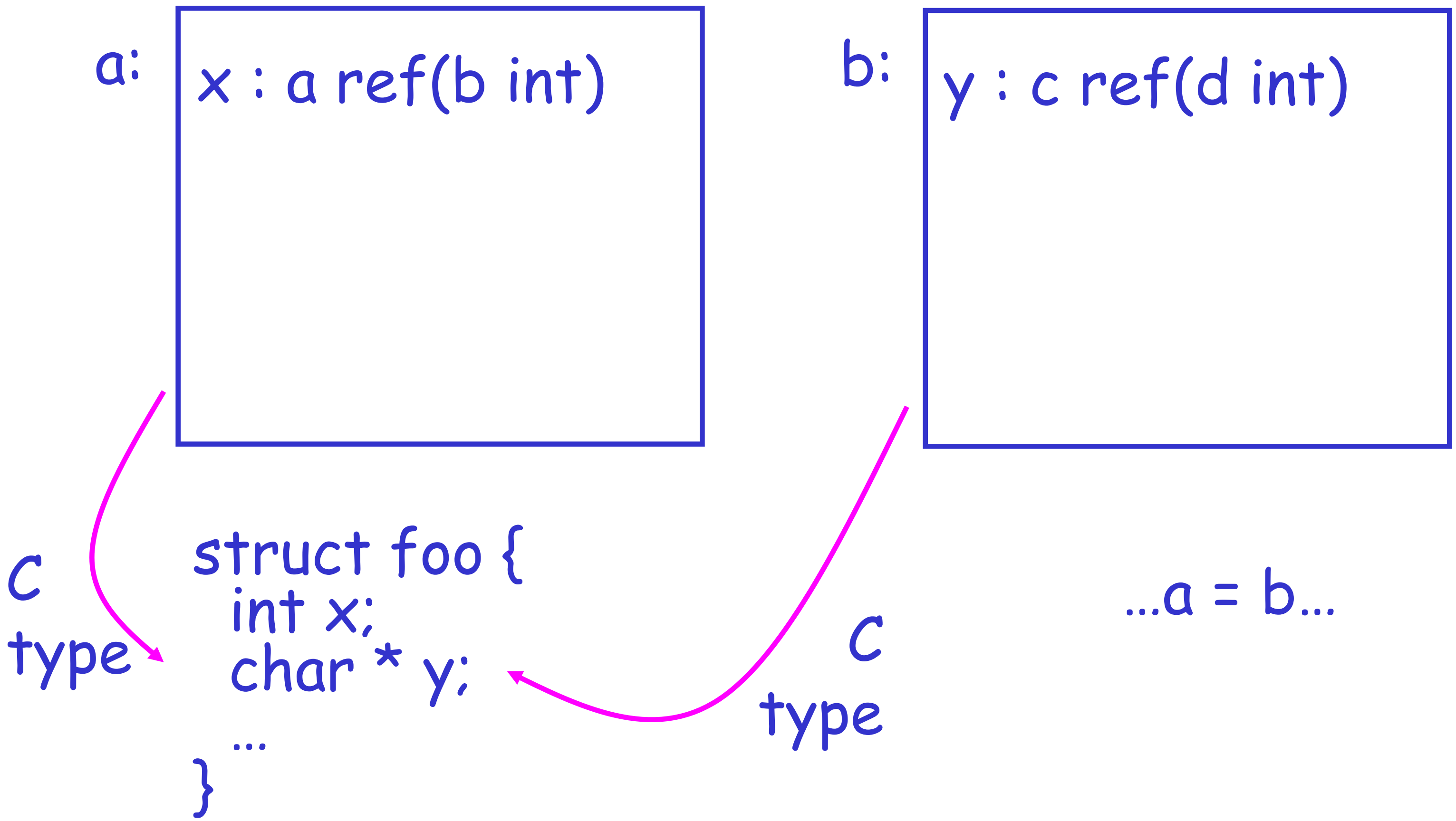
C type

```
struct foo {
  int x;
  char * y;
  ...
}
```

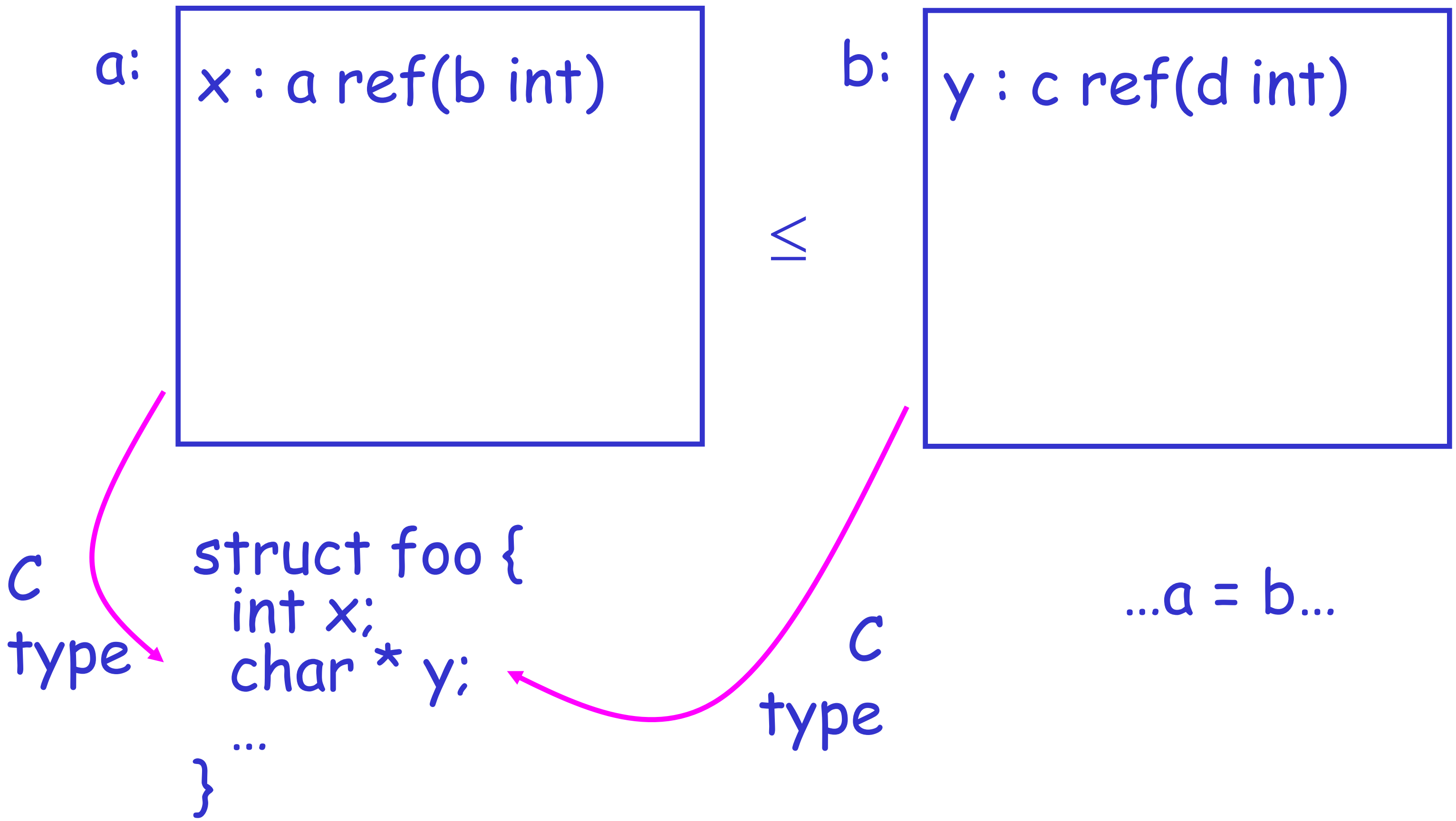
Lazy Field Expansion



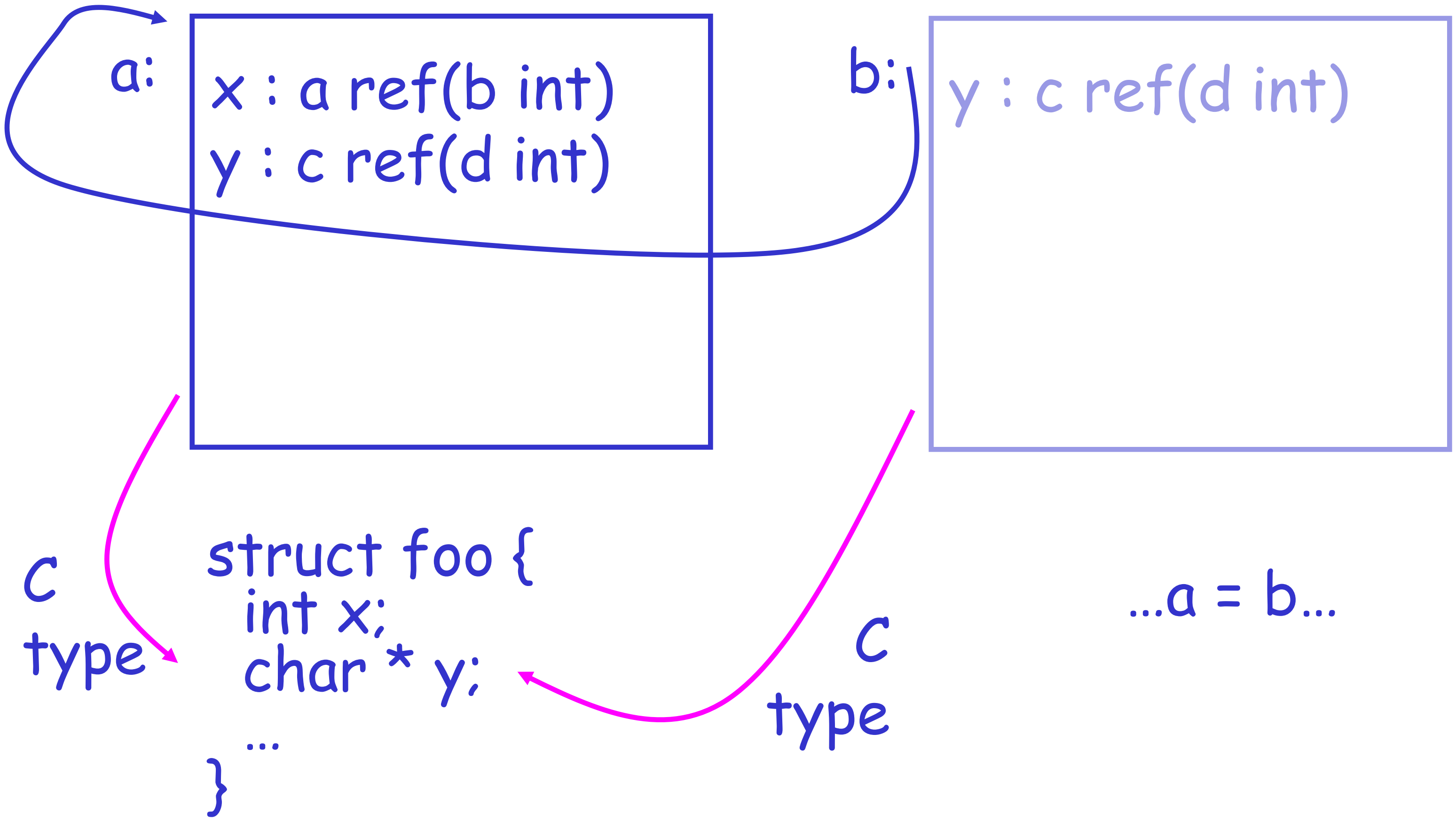
Lazy Field Expansion



Lazy Field Expansion



Lazy Field Expansion



Subtyping Under Pointer Types

- Recall we argued that an updateable reference behaves like an object with *get* and *set* operations

- Results in this rule:

$$\frac{Q \leq Q' \quad qt \leq qt' \quad qt' \leq qt}{ref^{Q}\, qt \leq ref^{Q'}\, qt'}$$

- What if we can't write through reference?

Subtyping Under Pointer Types

- C has a type qualifier const
  - If you declare const int *x, then *x = … not allowed
- So const pointers don't have "get" method
  - Can treat ref as covariant

$$\frac{Q \leq Q' \quad qt \leq qt' \quad \text{const} \leq Q'}{\text{ref}^{Q}\, qt \leq \text{ref}^{Q'}\, qt'}$$

Subtyping Under Pointer Types

- Turns out this is very useful
  - We're tracking taintedness of strings
  - Many functions read strings without changing their contents
  - Lots of use of const + opportunity to add it

Presenting Inference Results

- Type error = unsatisfiable constraints
  - E.g., path from tainted to untainted
- Heuristics for presenting "good" errors
  - Suppress derivative errors
    - $L \leq l1 \leq \ldots \leq ln \leq x \leq u1 \leq \ldots \leq um \leq u$ where $li = uj$
  - Suppress redundant errors
    - Only report one error for the above path
  - Suppress purely anonymous paths
    - Those that correspond to intermediate qualifier variables

Type Casts

Experiment: Format String Vulnerabilities

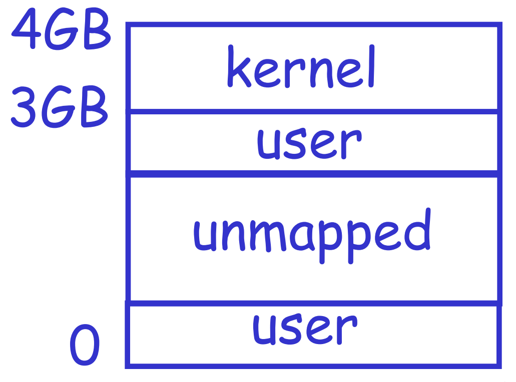
- Analyzed 10 popular unix daemon programs
  - Annotations shared across applications
    - One annotated header file for standard libraries
    - Includes annotations for polymorphism
      - Critical to practical usability
- Found several known vulnerabilities
  - Including ones we didn't know about
- User interface critical

Results: Format String Vulnerabilities

Name	Warn	Bugs
identd-1.0.0	0	0
mingetty-0.9.4	0	0
bftpd-1.0.11	1	1
muh-2.05d	2	~2
cfengine-1.5.4	5	3
imapd-4.7c	0	0
ipopd-4.7c	0	0
mars_nwe-0.99	0	0
apache-1.3.12	0	0
openssh-2.3.0p1	0	0

Experiment: User/kernel Vulnerabilities (Johnson + Wagner 04)

- In the Linux kernel, the kernel and user/mode programs share address space



  - The top 1GB is reserved for the kernel
  - When the kernel runs, it doesn't need to change VM mappings
    - Just enable access to top 1GB
    - When kernel returns, prevent access to top 1GB

Tradeoffs of This Memory Model

- Pros:
  - Not a lot of overhead
  - Kernel has direct access to user space

- Cons:
  - Leaves the door open to attacks from untrusted users
  - A pain for programmers to put in checks

An Attack

- Suppose we add two new system calls

```
int x;
void sys_setint(int *p) { memcpy(&x, p, sizeof(x)); }
void sys_getint(int *p) { memcpy(p, &x, sizeof(x)); }
```

- Suppose a user calls getint(buf)
  - Well-behaved program: buf points to user space
  - Malicious program: buf points to unmapped memory
  - Malicious program: buf points to kernel memory
    - We've just written to kernel space! Oops!

Another Attack

- Can we compromise security with setint(buf)?
  - What if buf points to private kernel data?
    - E.g., file buffers
  - Result can be read with getint

The Solution: copy_from_user, copy_to_user

- Our example should be written

```
int x;
void sys_setint(int *p) { copy_from_user(&x, p, sizeof(x)); }
void sys_getint(int *p) { copy_to_user(p, &x, sizeof(x)); }
```

- These perform the required safety checks
  - Return number of bytes that couldn't be copied
  - from_user pads destination with 0's if couldn't copy

It's Easy to Forget These

- Pointers to kernel and user space look the same
  - That's part of the point of the design
- Linux 2.4.20 has 129 syscalls with pointers to user space
  - All 129 of those need to use copy_from/to
  - The ioctl implementation passes user pointers to device drivers (without sanitizing them first)
- The result: Hundreds of copy_from/_to
  - One (small) kernel version: 389 from, 428 to
  - And there's no checking

User/Kernel Type Qualifiers

- We can use type qualifiers to distinguish the two kinds of pointers
  - kernel -- This pointer is under kernel control
  - user -- This pointer is under user control

- Subtyping kernel < user
  - It turns out copy_from/copy_to can accept pointers to kernel space where they expect pointers to user space

Type Signatures

- We add signatures for the appropriate fns:

```
intcopy_from_user(void *kernel to,
                  void *user from, intlen)
intmemcpy(void *kernel to,
          void *kernel from, intlen)
intx;
void sys_setint(int *userp) {
copy_from_user(&x, p, sizeof(x)); }
void sys_getint(int *userp) {
memcpy(p, &x, sizeof(x)); }
```

Lives in kernel

OK

OK

Error

Qualifiers and Type Structure

- Consider the following example:

```
void ioctl(void *user arg) {
  struct cmd { char *datap; } c;
  copy_from_user(&c, arg, sizeof(c));
  c.datap[0] = 0;    // not a good idea
}
```

- The pointer arg comes from the user
  - So datap in c also comes from the user
  - We shouldn't deference it without a check

Well-Formedness Constraints

- Simpler example

  char **user p;

  - Pointer p is under user control
  - Therefore so is *p
- We want a rule like:
  - In type ref^{user} (Q s), it must be that $Q \leq user$
  - This is a *well-formedness* condition on types

Well-Formedness Constraints

- Use conditional constraints

$$\frac{\vdash_{wf} (Q'\ s) \qquad Q \leq user \Longrightarrow Q' \leq user}{\vdash_{wfref^{Q}} (Q'\ s)}$$

  - "If Q must be user, then Q' must be also"
- Specify on a per-qualifier level whether to generate this constraint
  - Not hard to add to constraint resolution

Well-Formedness Constraints

- Similar constraints for struct types

$$\frac{\text{For all } i,\quad |\text{--wf } (Qi\ si) \qquad Q \leq user \implies Qi \leq user}{|\text{--wf } struct^{Q}\ (Q1\ s1, \ldots, Qn\ sn)}$$

  - Again, can specify this per-qualifier

A Tricky Example

int copy_from_user(<kernel>, <user>, <size>);

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                 unsigned long arg) {
  …case I2C_RDWR:
      if (copy_from_user(&rdwr_arg,
                         (struct i2c_rdwr_iotcl_data *) arg,
                         sizeof(rdwr_arg)))
        return -EFAULT;
      for (i = 0; i < rdwr_arg.nmsgs; i++) {
        if (copy_from_user(rdwr_pa[i].buf,
                           rdwr_arg.msgs[i].buf,
                           rdwr_pa[i].len)) {
              res = -EFAULT; break;
  } }
```

A Tricky Example

int copy_from_user(<kernel>, <user>, <size>);

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
unsigned long arg) {
   ...case I2C_RDWR:
        if (copy_from_user(&rdwr_arg,
                           (struct i2c_rdwr_iotcl_data *) arg,
                           sizeof(rdwr_arg)))
          return -EFAULT;
        for (i = 0; i < rdwr_arg.nmsgs; i++) {
          if (copy_from_user(rdwr_pa[i].buf,
                             rdwr_arg.msgs[i].buf,
                             rdwr_pa[i].len)) {
                res = -EFAULT; break;
    } }
```

user

A Tricky Example

```
int copy_from_user(<kernel>, <user>, <size>);
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
unsigned long arg) {
   …case I2C_RDWR:
        if (copy_from_user(&rdwr_arg,
(struct i2c_rdwr_iotcl_data *) arg,
                                      sizeof(rdwr_arg)))
          return -EFAULT;
        for (i = 0; i < rdwr_arg.nmsgs; i++) {
           if (copy_from_user(rdwr_pa[i].buf,
                                         rdwr_arg.msgs[i].buf,
                                         rdwr_pa[i].len)) {
                   res = -EFAULT; break;
        } }
```

user

OK

A Tricky Example

```
int copy_from_user(<kernel>, <user>, <size>);
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
unsigned long arg) {
   …case I2C_RDWR:
        if (copy_from_user(&rdwr_arg,
(struct i2c_rdwr_iotcl_data *) arg,
                                        sizeof(rdwr_arg)))
          return -EFAULT;
        for (i = 0; i < rdwr_arg.nmsgs; i++) {
          if (copy_from_user(rdwr_pa[i].buf,
rdwr_arg.msgs[i].buf,
                                          rdwr_pa[i].len)) {
                    res = -EFAULT; break;
     } }
```

user

OK

Bad

Experimental Results

- Ran on two Linux kernels
  - 2.4.20 -- 11 bugs found
  - 2.4.23 -- 10 bugs found
- Needed to add 245 annotations
  - Copy_from/to, kmalloc, kfree, …
  - All Linux syscalls take user args (221 calls)
    - Could have be done automagically (All begin with sys_)
- Ran both single file (unsound) and whole-kernel
  - Disabled subtyping for single file analysis

More Detailed Results

- 2.4.20, full config, single file
  - 512 raw warnings, 275 unique, 7 exploitable bugs
    - Unique = combine msgs for user qual from same line
- 2.4.23, full config, single file
  - 571 raw warnings, 264 unique, 6exploitable bugs
- 2.4.23, default config, single file
  - 171 raw warnings, 76 unique, 1 exploitable bug
- 2.4.23, default config, whole kernel
  - 227 raw warnings, 53 unique, 4 exploitable bugs

Observations

- Quite a few false positives
  - Large code base magnifies false positive rate

- Several bugs persisted through a few kernels
  - 8 bugs found in 2.4.23 that persisted to 2.5.63
  - An unsound tool, MECA, found 2 of 8 bugs
  - ==> Soundness matters!

Observations

- Of 11 bugs in 2.4.23…
  - 9 are in device drivers
  - Good place to look for bugs!
  - Note: errors found in “core” device drivers
    - (4 bugs in PCMCIA subsystem)

- Lots of churn between kernel versions
  - Between 2.4.20 and 2.4.23
    - 7 bugs fixed
    - 5 more introduced

Conclusion

- Type qualifiers are specifications that…
  - Programmers will accept
    - Lightweight
  - Scale to large programs
  - Solve many different problems

- In the works: ccqual, jqual, Eclipse interface