

# Security via Type Qualifiers

COMP 150-AVS

Fall 2018

# Introduction

---

- Ensuring that software is secure is hard
- Standard practice for software quality:
  - Testing
    - Make sure program runs correctly on set of inputs
  - Code auditing
    - Convince yourself and others that your code is correct

# Drawbacks to Standard Approaches

---

- Difficult
- Expensive
- Incomplete
  
- A **malicious adversary** is trying to exploit anything you miss!

# Tools for Security

---

- What more can we do?
  - Build tools that analyze source code
    - 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 security

# Tools Need Specifications

---

```
spin_lock_irqsave(&tty->read_lock, flags);  
put_tty_queue_nolock(c, tty);  
spin_unlock_irqrestore(&tty->read_lock, flags);
```

- **Goal: Add specifications to programs**  
In a way that...
  - Programmers will accept
    - Lightweight
  - Scales to large programs
  - Solves many different problems

# 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
  - Too restrictive to forbid variable format strings

# Using Tainted and Untainted

---

- Add qualifier annotations

```
int printf(untainted char *fmt, ...)
```

```
tainted char *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

f accepts tainted or  
untainted data

untainted  $\leq$  tainted

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

Error

g accepts only untainted  
data

tainted  $\not\leq$  untainted

untainted  $<$  tainted

# The Plan

---

- The Nice Theory
- Polymorphism
- The Icky Stuff in C

# Type Qualifiers for MinML

---

- We'll add type qualifiers to MinML
  - Same approach works for other languages (like C)
- Standard type systems define types as
  - $t ::= c_0(t, \dots, t) \mid \dots \mid c_n(t, \dots, t)$ 
    - Where  $\Sigma = c_0 \dots c_n$  is a set of *type constructors*
- Here are the **types** of MinML
  - $t ::= \text{int} \mid \text{bool} \mid t \rightarrow t$ 
    - Here  $\Sigma = \text{int}, \text{bool}, \rightarrow$  (written infix)

# Type Qualifiers for MinML (cont'd)

---

- Let  $Q$  be the set of type qualifiers
  - Assumed to be chosen in advance and fixed
  - E.g.,  $Q = \{\text{tainted}, \text{untainted}\}$
- Then the *qualified types* are just
  - $qt ::= Q s$
  - $s ::= c_0(qt, \dots, qt) \mid \dots \mid c_n(qt, \dots, qt)$ 
    - Allow a type qualifier to appear on each type constructor
- For MinML
  - $qt ::= \text{int}^Q \mid \text{bool}^Q \mid qt \rightarrow^Q qt$

# Abstract Syntax of MinML with Qualifiers

---

$e ::= x \mid n \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid$   
 $\text{fun } f^Q(x:qt):qt = e \mid e \mid \text{annot}(Q, e) \mid \text{check}(Q, e)$

- $\text{annot}(Q, e)$  = "expression  $e$  has qualifier  $Q$ "
- $\text{check}(Q, e)$  = "fail if  $e$  does not have qualifier  $Q$ "
  - Checks only the top-level qualifier
- Examples:
  - $\text{fun } \text{fread}(x:qt):\text{int}^{\text{tainted}} = \dots \text{annot}(\text{tainted}, 42)$
  - $\text{fun } \text{printf}(x:qt):qt' = \text{check}(\text{untainted}, x), \dots$

# Typing Rules: Qualifier Introduction

---

- Newly-constructed values have “bare” types

$$\frac{}{G \vdash n : \text{int}}$$

$$\frac{}{G \vdash \text{true} : \text{bool}}$$

$$\frac{}{G \vdash \text{false} : \text{bool}}$$

- Annotation adds an outermost qualifier

$$\frac{G \vdash e_1 : s}{G \vdash \text{annot}(Q, e) : Q s}$$

# Typing Rules: Qualifier Elimination

---

- By default, discard qualifier at destructors

$$\frac{G \dashv\vdash e1 : \text{bool}^Q \quad G \dashv\vdash e2 : qt \quad G \dashv\vdash e3 : qt}{G \dashv\vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : qt}$$

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

$$\frac{G \dashv\vdash e1 : Qs}{G \dashv\vdash \text{check}(Q, e) : Qs}$$

# 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?

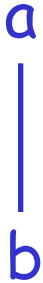
# Partial Orders

---

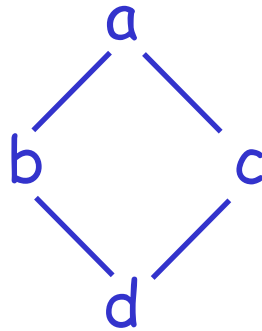
- Qualifiers  $Q$  come with a partial order  $\leq$ :
  - $q \leq q$  (reflexive)
  - $q \leq p, p \leq q \Rightarrow q = p$  (anti-symmetric)
  - $q \leq p, p \leq r \Rightarrow q \leq r$  (transitive)
- Qualifiers introduce subtyping
- In our example:
  - $\text{untainted} < \text{tainted}$

# Example Partial Orders

---



2-point lattice



Discrete partial order

- Lower in picture = lower in partial order
- Edges show  $\leq$  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:

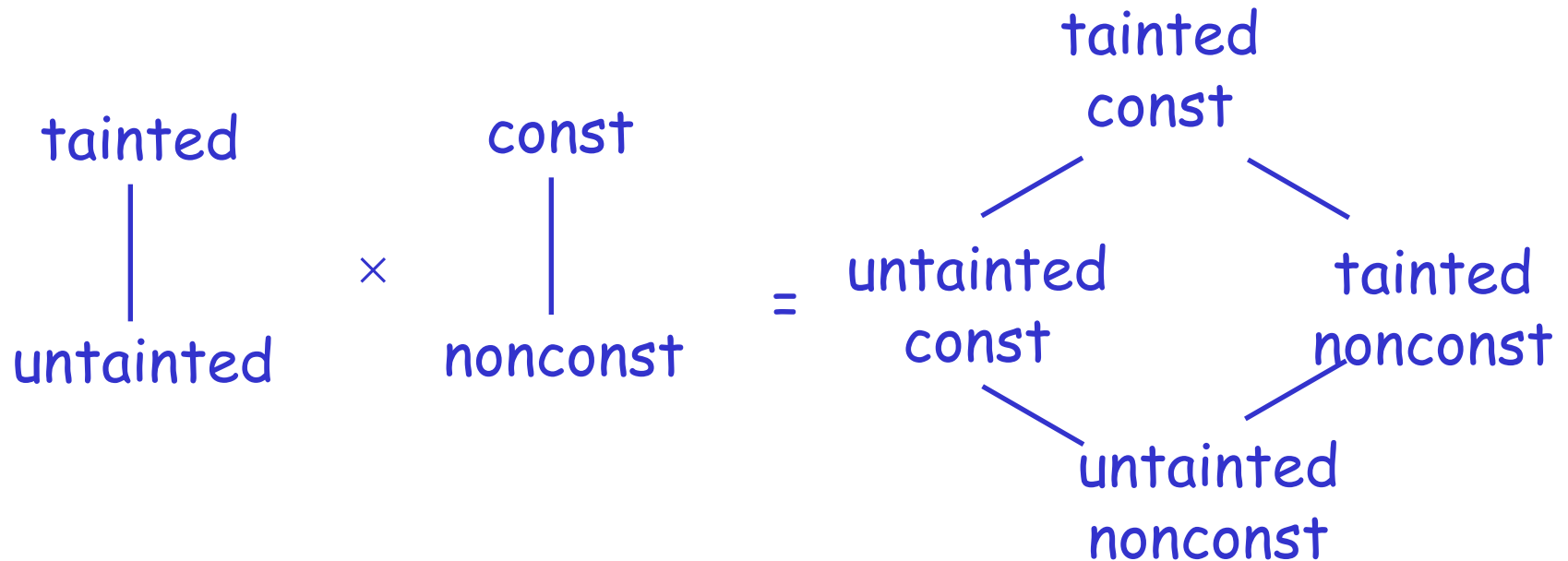
$$(Q_1, \leq_1) \times (Q_2, \leq_2) = (Q, \leq)$$

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

---

$$\frac{Q \leq Q'}{\text{bool}^Q \leq \text{bool}^{Q'}}$$

$$\frac{Q \leq Q'}{\text{int}^Q \leq \text{int}^{Q'}}$$

- Add one new rule *subsumption* to type system

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

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

# Use of Subsumption

---

$\vdash 42 : \text{int}$

---

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

---

$\vdash \text{annot}(\text{untainted}, 42) : \text{tainted int}$

---

$\vdash \text{check}(\text{tainted}, \text{annot}(\text{untainted}, 42)) : \text{tainted int}$

# Subtyping on Function Types

---

- What about function types?

$$\frac{?}{q_{t1} \rightarrow^Q q_{t2} \leq q_{t1'} \rightarrow^{Q'} q_{t2'}}$$

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

# Replacing "f x" by "g x"

---

- When is  $qt1' \rightarrow^Q qt2' \leq qt1 \rightarrow^Q qt2$  ?
- 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  $qt1' \rightarrow^Q qt2' \leq qt1 \rightarrow^Q qt2$  ?
- 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
  - $v ::= x \mid n^Q \mid \text{true}^Q \mid \text{false}^Q$   
 $\mid \text{fun } f^Q (x : qt1) : qt2 = e$
- Evaluation rules same as before, carrying the qualifiers along, e.g.,

---

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

# 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 proofs to this system
  - (Not too much work; really just need to show that `check` doesn't get stuck)

# Updateable References

---

- Our MinML 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
  - $e ::= \dots \mid \text{ref}^Q e \mid !e \mid e := e$ 
    - $\text{ref}^Q e$  -- Allocate memory and set its contents to  $e$ 
      - Returns memory location
      - $Q$  is qualifier on pointer (not on contents)
    - $!e$  -- 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:
  - $qt ::= \dots \mid \text{ref}^Q qt$ 
    - Note: In ML the ref appears on the right

$$\frac{G \dashv\vdash e : qt}{G \dashv\vdash \text{ref}^Q e : \text{ref}^Q qt}$$

$$\frac{G \dashv\vdash e : \text{ref}^Q qt}{G \dashv\vdash !e : qt}$$

$$\frac{G \dashv\vdash e1 : \text{ref}^Q qt \quad G \dashv\vdash e2 : qt}{G \dashv\vdash e1 := e2 : qt}$$

# Subtyping References

---

- The *wrong* rule for subtyping references is

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

- Counterexample

let  $x = \text{ref } 0^{\text{untainted}}$  in

let  $y = x$  in

$y := 3^{\text{tainted}};$

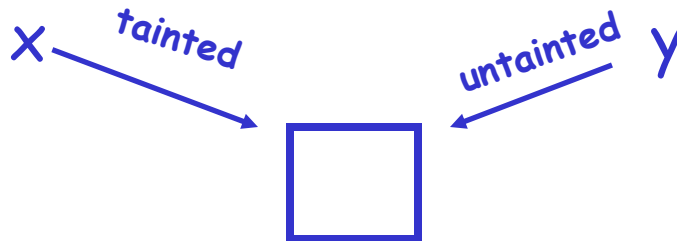
$\text{check}(\text{untainted}, !x)$

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:
    - $\text{get} : \text{unit} \rightarrow \text{qt}$
    - $\text{set} : \text{qt} \rightarrow \text{unit}$
  - Notice that  $\text{qt}$  occurs both co- and contravariantly
- The right rule:

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

# Challenge Problem: 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

# 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

# First Problem: Subsumption Rule

---

$$\frac{G \vdash e : qt \quad qt \leq qt'}{G \vdash e : qt'}$$

- 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
  - Incorporate it directly into the other rules

$$\frac{\frac{G \dashv\vdash e1 : qt' \rightarrow^{Q'} qt'' \quad qt1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt2}{G \dashv\vdash e1 : qt1 \rightarrow^Q qt2} \quad \frac{G \dashv\vdash e2 : qt \quad qt \leq qt1}{G \dashv\vdash e2 : qt1}}{G \dashv\vdash e1 e2 : qt2}$$

# Getting Rid of Subsumption (cont'd)

---

- 1. Fold  $e_2$  subsumption into rule

$$\frac{\begin{array}{c} G \dashv\vdash e_1 : qt' \rightarrow^{Q'} qt'' \\ qt_1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt_2 \end{array}}{\frac{G \dashv\vdash e_1 : qt_1 \rightarrow^Q qt_2 \quad G \dashv\vdash e_2 : qt \quad qt \leq qt_1}{G \dashv\vdash e_1 e_2 : qt_2}}$$

## Getting Rid of Subsumption (cont'd)

---

- 2. Fold  $e1$  subsumption into rule

$$qt1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt2$$

$$\frac{G \vdash\!\!\!-\ e1 : qt' \rightarrow^{Q'} qt'' \quad G \vdash\!\!\!-\ e2 : qt \quad qt \leq qt1}{G \vdash\!\!\!-\ e1 \ e2 : qt2}$$

## Getting Rid of Subsumption (cont'd)

---

- 3. We don't use  $Q$ , so remove that constraint

$$\begin{array}{c} qt1 \leq qt' \qquad qt'' \leq qt2 \\ G \dashv\vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad G \dashv\vdash e2 : qt \quad qt \leq qt1 \\ \hline G \dashv\vdash e1 e2 : qt2 \end{array}$$

# Getting Rid of Subsumption (cont'd)

---

- 4. Apply transitivity of  $\leq$ 
  - Remove intermediate  $qt_1$

$$\frac{qt'' \leq qt_2 \quad G \vdash\!\!-\ e_1 : qt' \rightarrow^{Q'} qt'' \quad G \vdash\!\!-\ e_2 : qt \quad qt \leq qt'}{G \vdash\!\!-\ e_1 e_2 : qt_2}$$

## Getting Rid of Subsumption (cont'd)

---

- 5. We're going to apply subsumption afterward, so no need to weaken  $qt''$

$$\frac{G \vdash\!\!\!-\ e1 : qt' \rightarrow^{Q'} qt'' \quad G \vdash\!\!\!-\ e2 : qt \quad qt \leq qt'}{G \vdash\!\!\!-\ e1 \ e2 : qt''}$$

## Getting Rid of Subsumption (cont'd)

---

- We apply the same reasoning to the other rules
  - We're left with a purely syntax-directed system
- Good! Now we're half-way to an algorithm

## Second Problem: Assumptions

---

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

$$\frac{G, f: qt1 \rightarrow^Q qt2, x:qt1 \dashv\vdash e : qt2' \quad qt2' \leq qt2}{G \dashv\vdash \text{fun } f^Q (x:qt1):qt2 = e : qt1 \rightarrow^Q qt2}$$

- 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

# Unknowns in Qualifier Inference

---

- We've got *regular* type annotations for functions
  - (We could even get away without these...)

$$\frac{G, f: ? \rightarrow^Q ?, x:? \dashv\vdash e : qt2' \quad qt2' \leq qt2}{G \dashv\vdash \text{fun } f^Q (x:t1):t2 = e : qt1 \rightarrow^Q qt2}$$

- How do we pick the qualifiers for  $f$ ?
  - We generate fresh, unknown *qualifier variables* and then solve for them

# Adding Fresh Qualifiers

---

- We'll add qualifier variables  $a, b, c, \dots$  to our set of qualifiers
  - (Letters closer to  $p, q, r$  will stand for constants)
- Define  $\text{fresh} : t \rightarrow qt$  as
  - $\text{fresh}(\text{int}) = \text{int}^a$
  - $\text{fresh}(\text{bool}) = \text{bool}^a$
  - $\text{fresh}(\text{ref}^Q t) = \text{ref}^a \text{fresh}(t)$
  - $\text{fresh}(t1 \rightarrow t2) = \text{fresh}(t1) \rightarrow^a \text{fresh}(t2)$ 
    - Where  $a$  is fresh

# Rule for Functions

---

$qt1 = \text{fresh}(t1) \quad qt2 = \text{fresh}(t2)$

$G, f: qt1 \rightarrow^Q qt2, x:qt1 \dashv\vdash e : qt2' \quad qt2' \leq qt2$

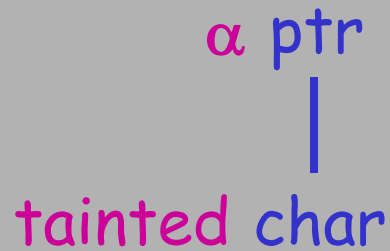
---

$G \dashv\vdash \text{fun } f^Q (x:t1):t2 = e : qt1 \rightarrow^Q qt2$

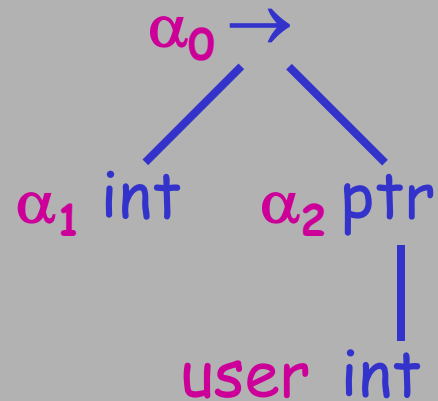
# A Picture of Fresh Qualifiers

---

ptr(tainted char)



int  $\rightarrow$  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 security vulnerability

# Solving Constraints: Step 1

---

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

# Solving Constraints: Step 1

---

- $S + \{ \text{int}^Q \leq \text{int}^{Q'} \} \implies S + \{ Q \leq Q' \}$
- $S + \{ \text{bool}^Q \leq \text{bool}^{Q'} \} \implies S + \{ Q \leq Q' \}$
- $S + \{ \text{qt1} \xrightarrow{Q} \text{qt2} \leq \text{qt1}' \xrightarrow{Q'} \text{qt2}' \} \implies$   
 $S + \{ \text{qt1}' \leq \text{qt1} \} + \{ \text{qt2} \leq \text{qt2}' \} + \{ Q \leq Q' \}$
- $S + \{ \text{ref}^Q \text{qt1} \leq \text{ref}^{Q'} \text{qt2} \} \implies$   
 $S + \{ \text{qt1} \leq \text{qt2} \} + \{ \text{qt2} \leq \text{qt1} \} + \{ Q \leq Q' \}$
- $S + \{ \text{mismatched constructors} \} \implies \text{error}$ 
  - Can't happen if program correct w.r.t. std types

## Solving Constraints: Step 2

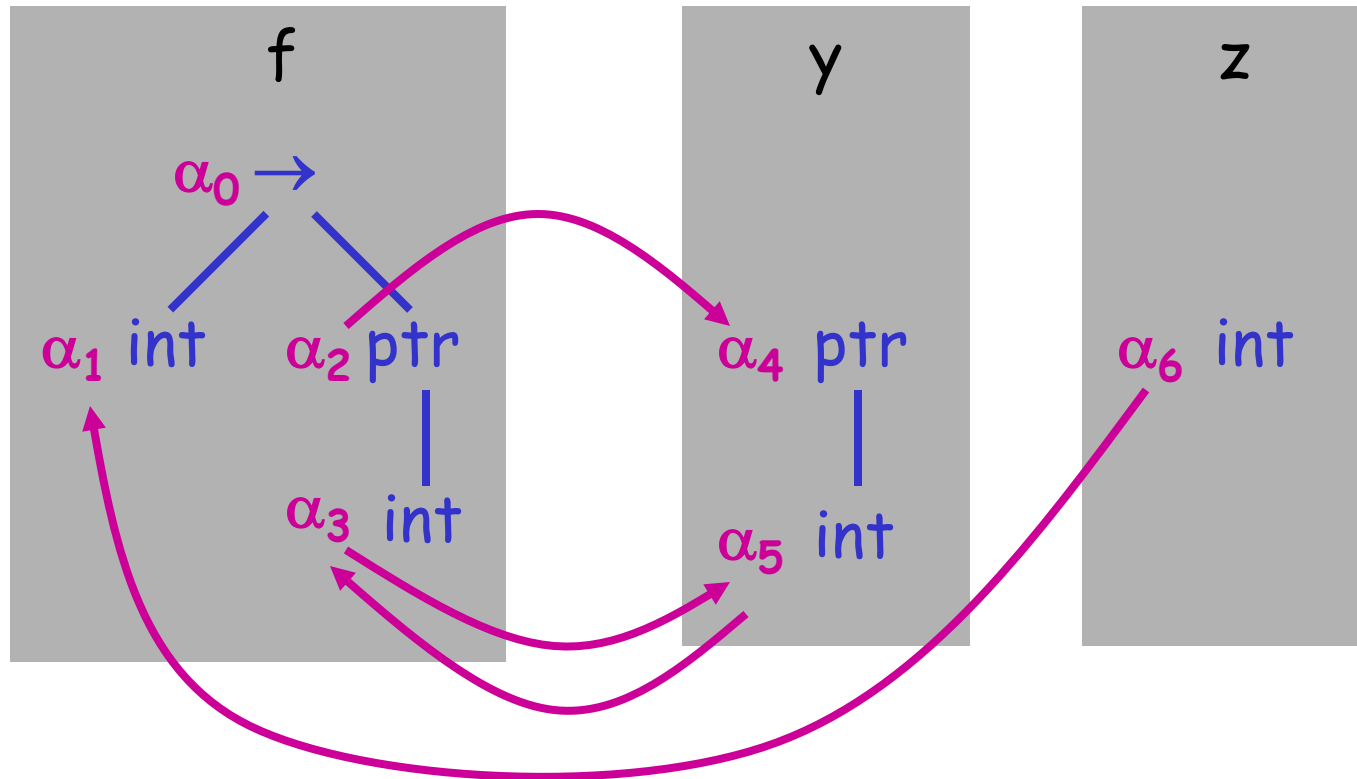
---

- 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)`



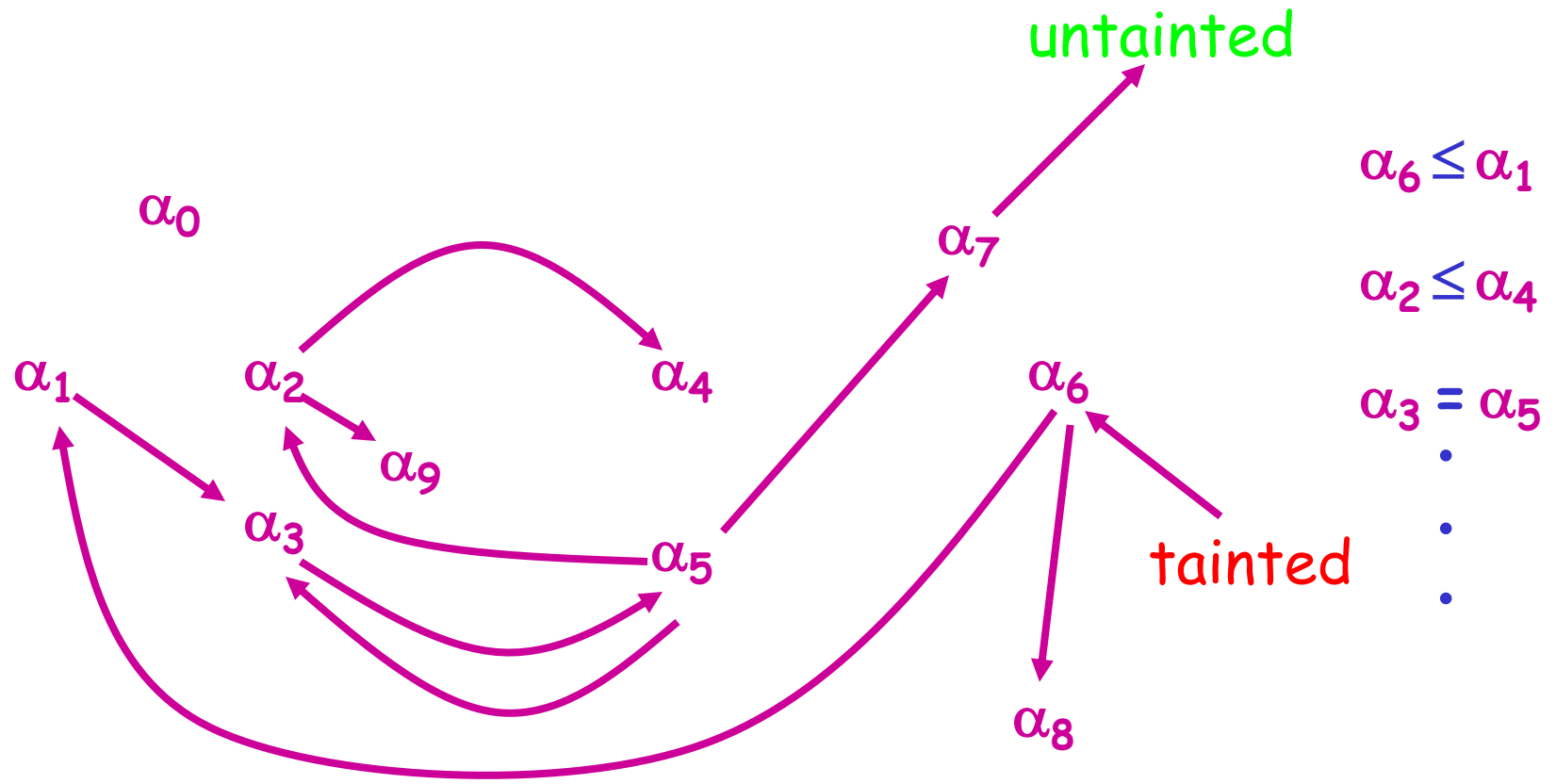
$$\alpha_6 \leq \alpha_1$$

$$\alpha_2 \leq \alpha_4$$

$$\alpha_3 = \alpha_5$$

# Constraints as Graphs

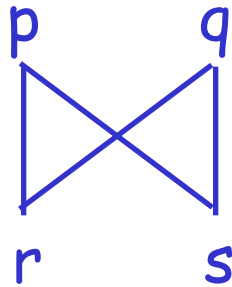
---



# 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
  - $p \text{ lub } q = r$  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

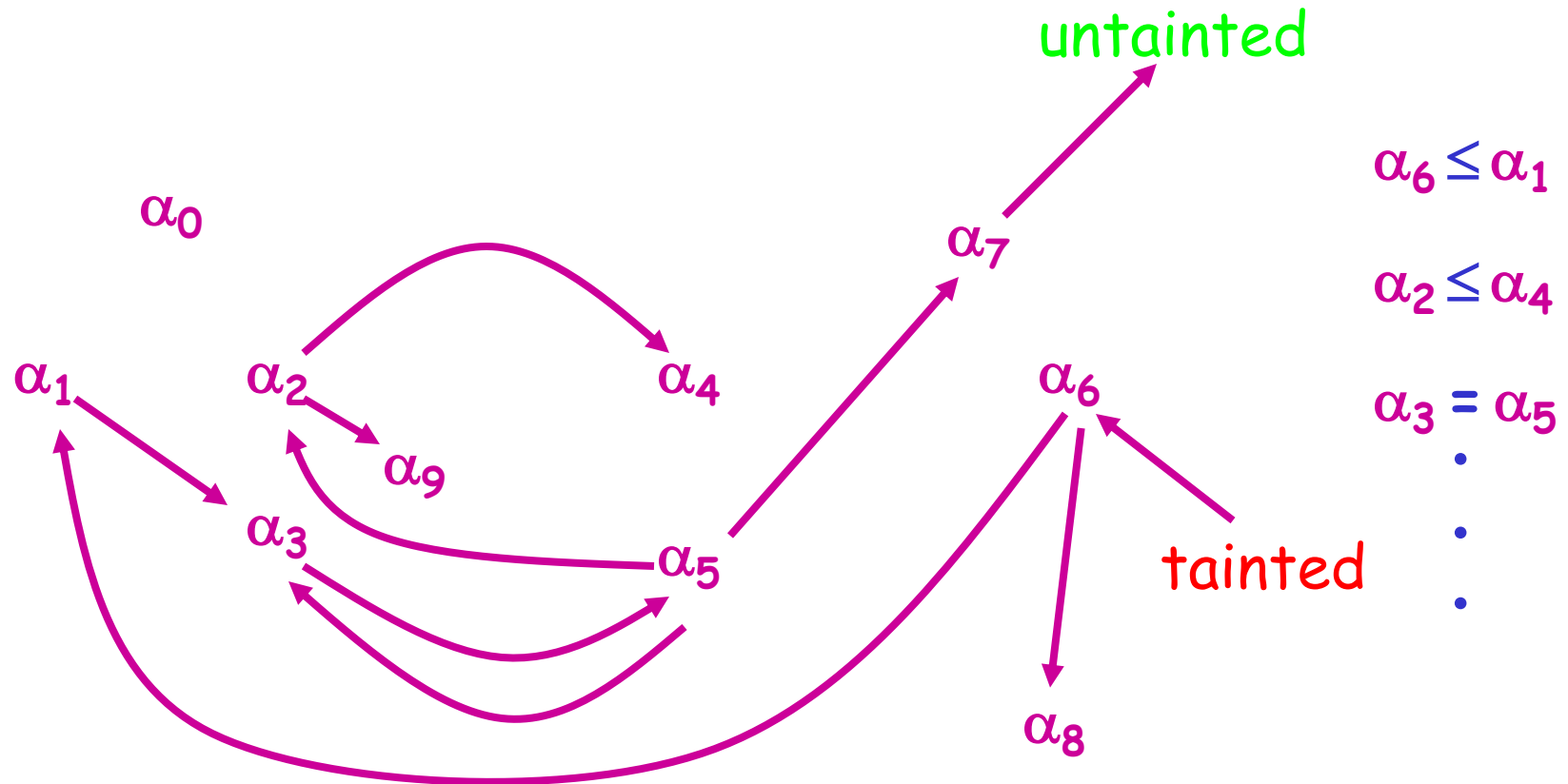
# Lattices

---

- A *lattice* is a partial order such that lubs and glbs always exist
- If  $\mathcal{Q}$  is a lattice, it turns out we can use a really simple algorithm to check satisfiability of constraints over  $\mathcal{Q}$

# Satisfiability via Graph Reachability

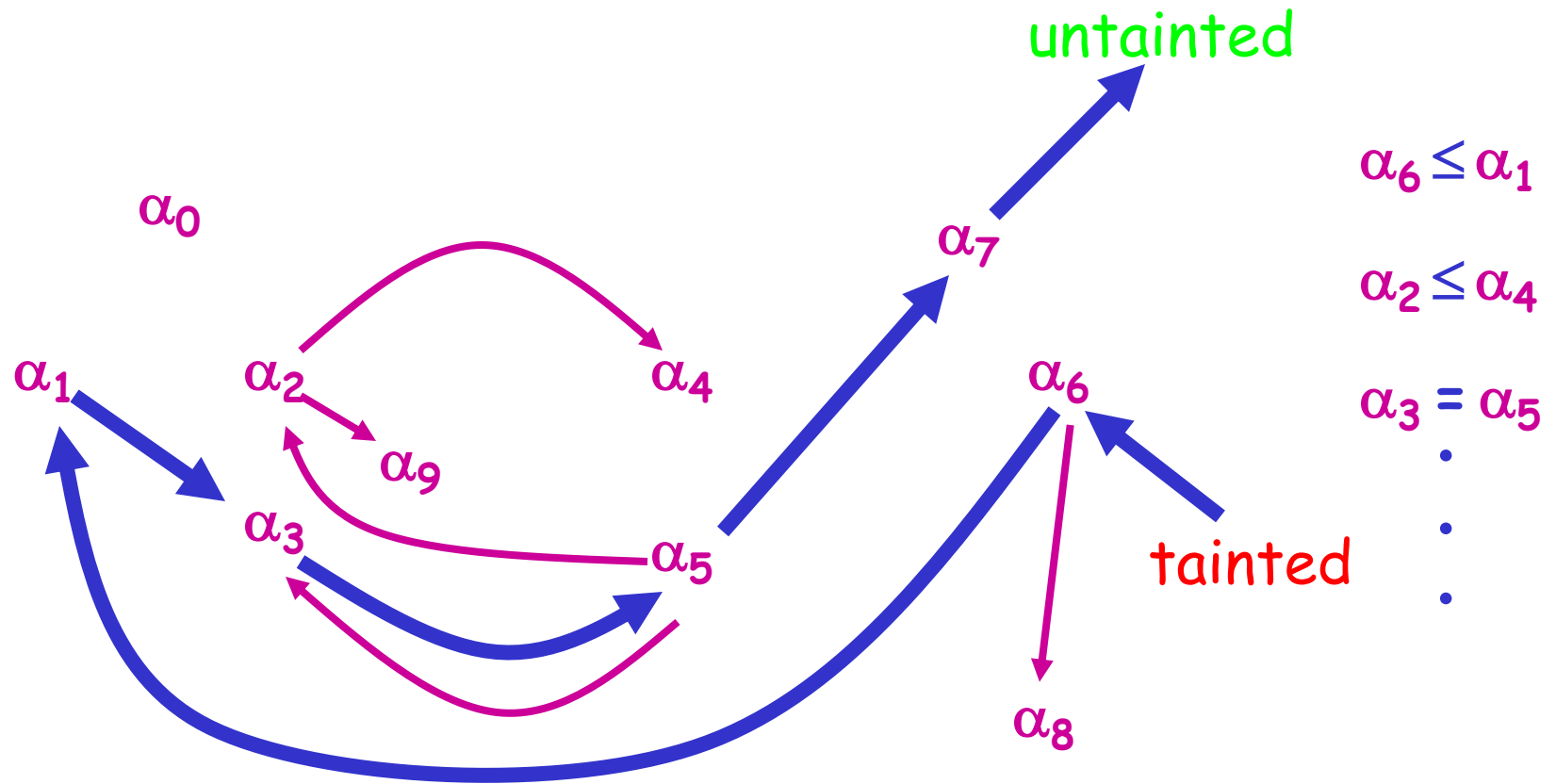
Is there an inconsistent path through the graph?





# Satisfiability via Graph Reachability

tainted  $\leq \alpha_6 \leq \alpha_1 \leq \alpha_3 \leq \alpha_5 \leq \alpha_7 \leq$  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 ≤ tainted`
- Let's look at the identity function
  - `fun id (x:int):int = x`
- What qualified types can we infer for `id`?

# Types for id

---

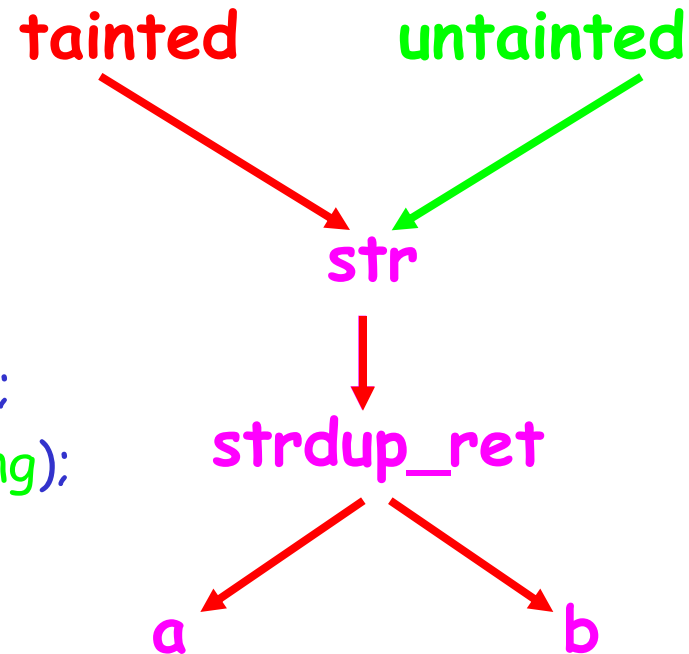
- `fun id (x:int):int = x` (ignoring int, qual on id)
  - `tainted → tainted`
    - Fine but untainted data passed in becomes tainted
  - `untainted → untainted`
    - Fine but can't pass in tainted data
  - `untainted → tainted`
    - Not too useful
  - `tainted → untainted`
    - Impossible

# 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?

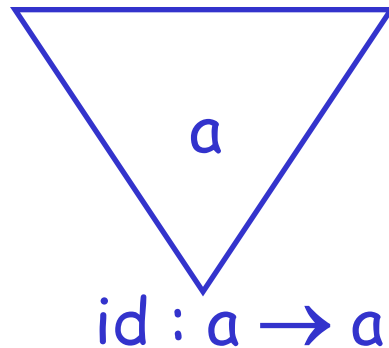
---

- The qualifier on  $x$  appears both covariantly and contravariantly in the type
  - We're stuck
- We need *parametric polymorphism*
  - We want to give `fun id (x:int):int = x` the type
$$\forall a.int^a \rightarrow int^a$$

# The Observation of Parametric Polymorphism

---

- Type inference on `id` yields a proof like this:

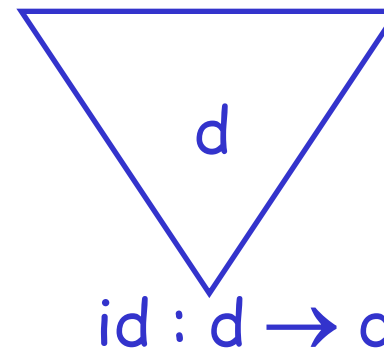
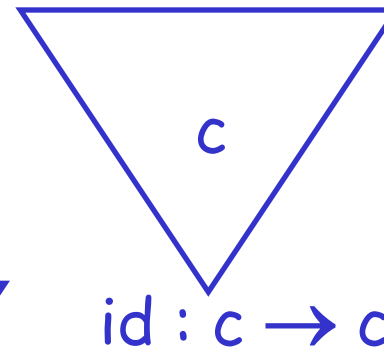
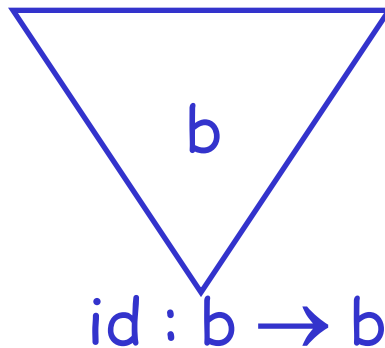
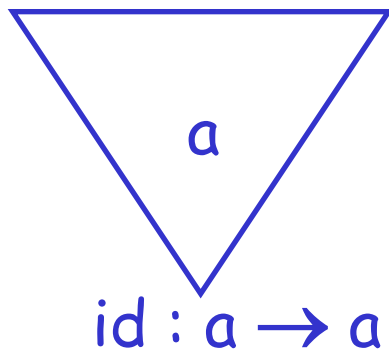


- If we just infer a type for `id`, no constraints will be placed on `a`

# The Observation of Parametric Polymorphism

---

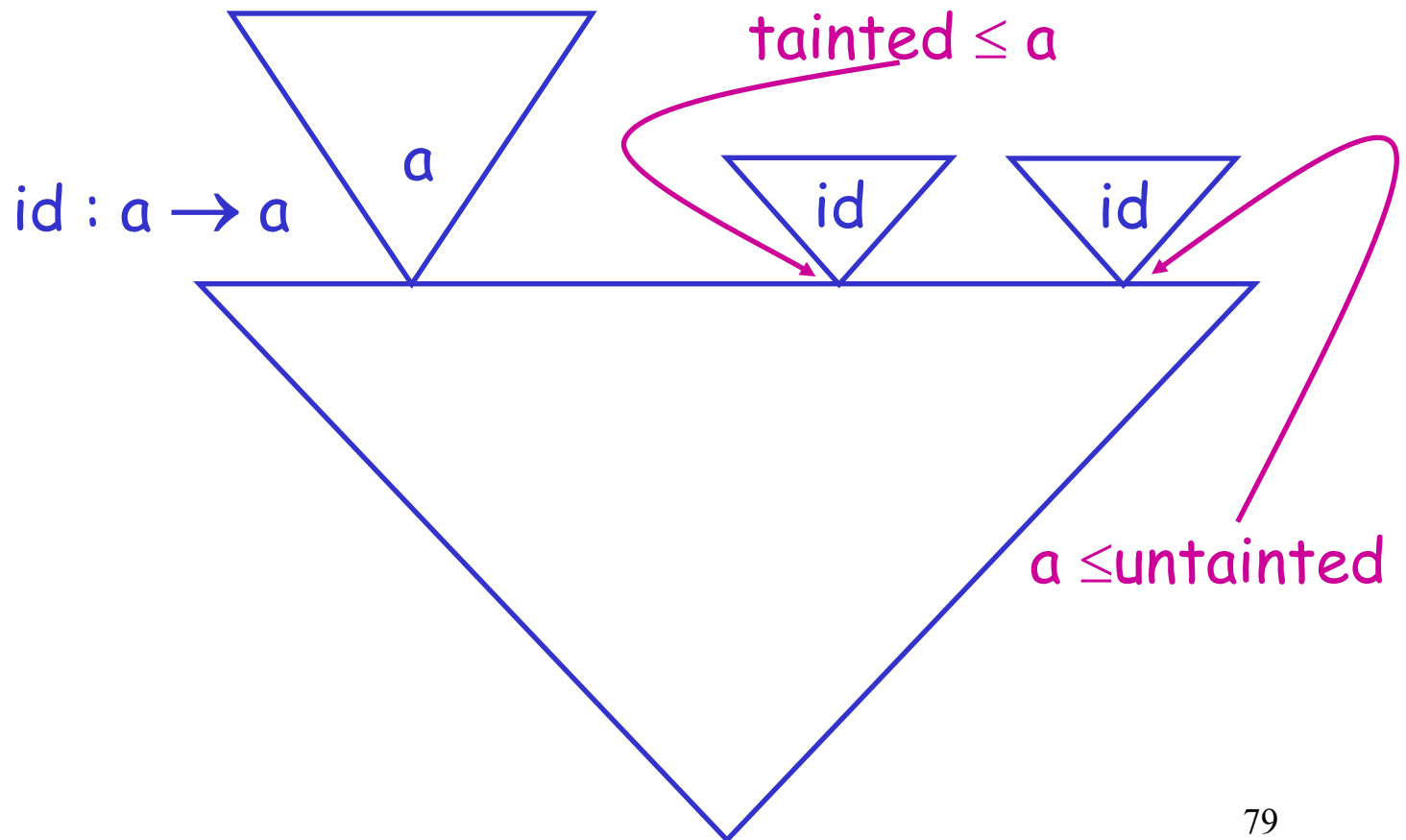
- We can duplicate this proof *for any*  $a$ , *in any* type environment



# The Observation of Parametric Polymorphism

---

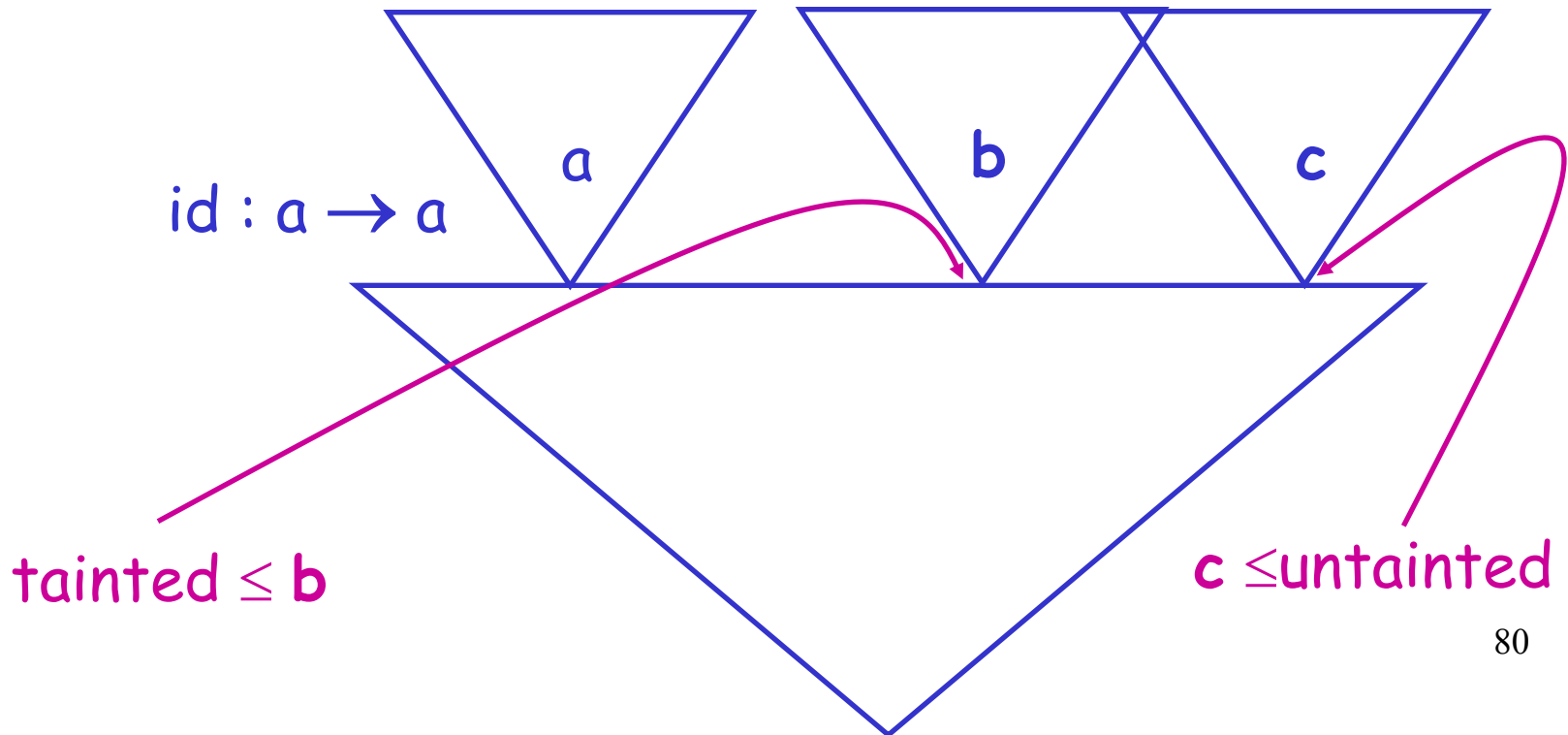
- The constraints on  $a$  only come from "outside"



# The Observation of Parametric Polymorphism

---

- But the two uses of `id` are different
  - We can inline `id`
  - And compute a type with a different `a` each time



# Implementing Polymorphism Efficiently

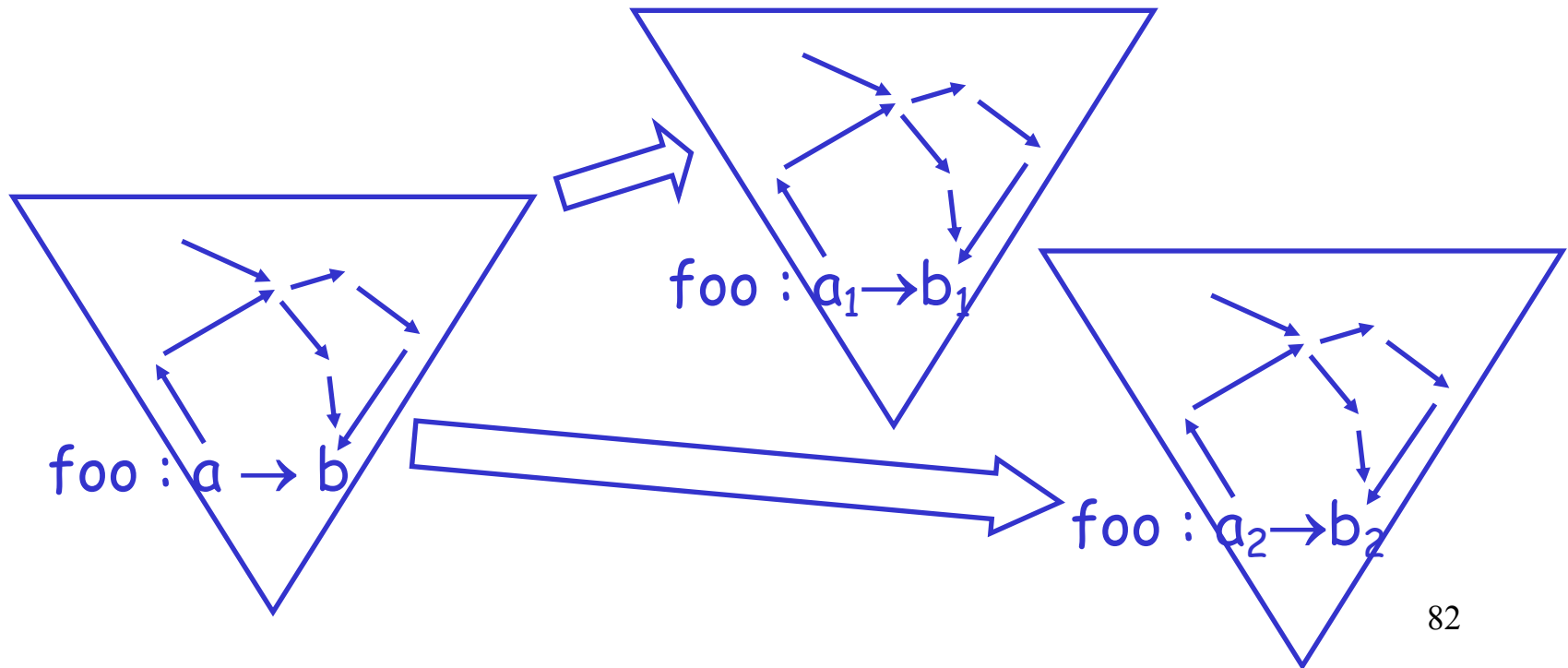
---

- 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  $C$
- We need polymorphically constrained types
$$x : \forall a. qt \text{ where } C$$
  - For any qualifiers  $a$  where constraints  $C$  hold,  $x$  has type  $qt$

# Polymorphically Constrained Types

---

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



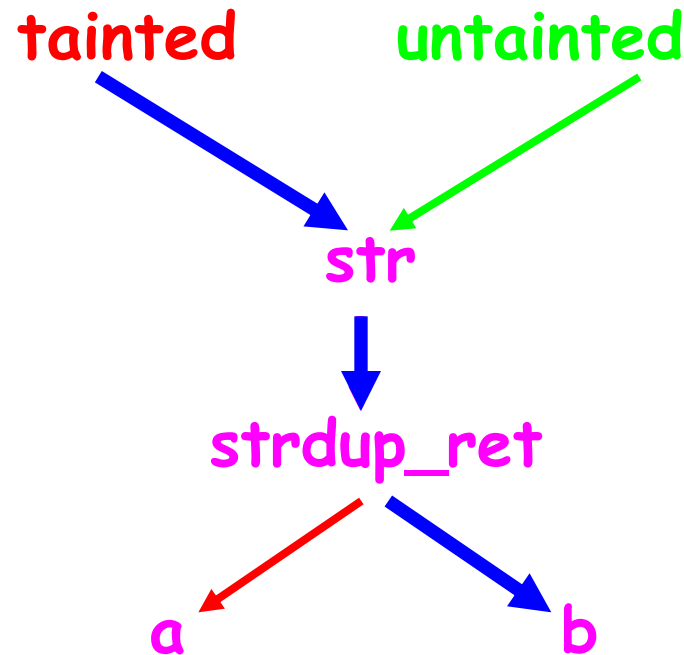
# A Better Solution: CFL Reachability

---

- 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  $O(n^8)$
- Idea due to Horwitz, Reps, and Sagiv, and Rehof, Fahndrich, and Das

# The Problem Restated: Unrealizable Paths

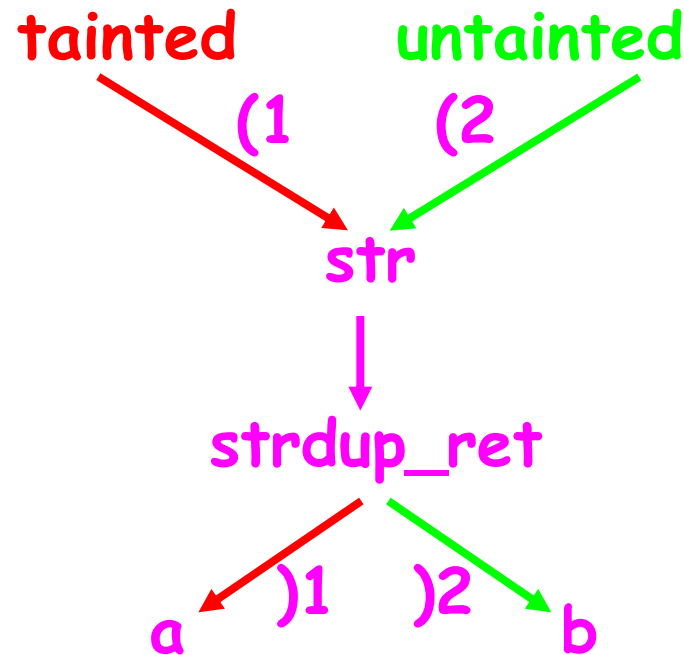
---



- 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

# Instantiation Constraints

---

- These edges represent a new kind of constraint

$$a \leq^{+/-}_i b$$

- At use  $i$  of a polymorphic type
- Qualifier variable  $a$
- Is instantiated to qualifier  $b$
- Either positively or negatively (or both)
- Formally, these are *semiunification* constraints
  - But we won't discuss that

# Type Rules

---

- We'll use Hindley-Milner style polymorphism
  - Quantifiers only appear at the outmost level
  - Quantified types only appear in the environment

$$qt1 = \text{fresh}(t1) \quad qt2 = \text{fresh}(t2)$$

$$G, f: qt1 \rightarrow^Q qt2, x:qt1 \dashv\vdash e : qt2' \quad qt2' \leq qt2$$

---

$$G \dashv\vdash \text{fun } f^Q (x:t1):t2 = e : qt1 \rightarrow^Q qt2$$

- \* This is not quite the right rule, yet...

# Type Rules

---

$$\frac{qt = G(f) \quad qt' = \text{fresh}(qt) \quad qt \leq_{+i} qt'}{G \vdash f_i : qt'}$$

- Implicit: Only apply to function names ( $f$ )
- Each has a label  $i$
- $\text{fresh}(qt)$  generates type like  $qt$  but with fresh quals
  - \*This is not quite the right rule yet...

# Resolving Instantiation Constraints

---

- Just like subtyping, reduce to only qualifiers
  - $S + \{ \text{int}^Q \leq_{pi} \text{int}^{Q'} \} \implies S + \{ Q \leq_{pi} Q' \}$ 
    - $p$  stands for either  $+$  or  $-$
  - ...
  - $S + \{ qt1 \rightarrow^Q qt2 \leq_{pi} qt1' \rightarrow^{Q'} qt2' \} \implies$   
 $S + \{ qt1 \leq_{(-p)i} qt1' \} + \{ qt2 \leq_{pi} qt2' \} + \{ Q \leq_{pi} Q' \}$ 
    - Here  $-(+)$  is  $-$  and  $-(-)$  is  $+$

# Instantiation Constraints as Graphs

---

- Three kinds of edges

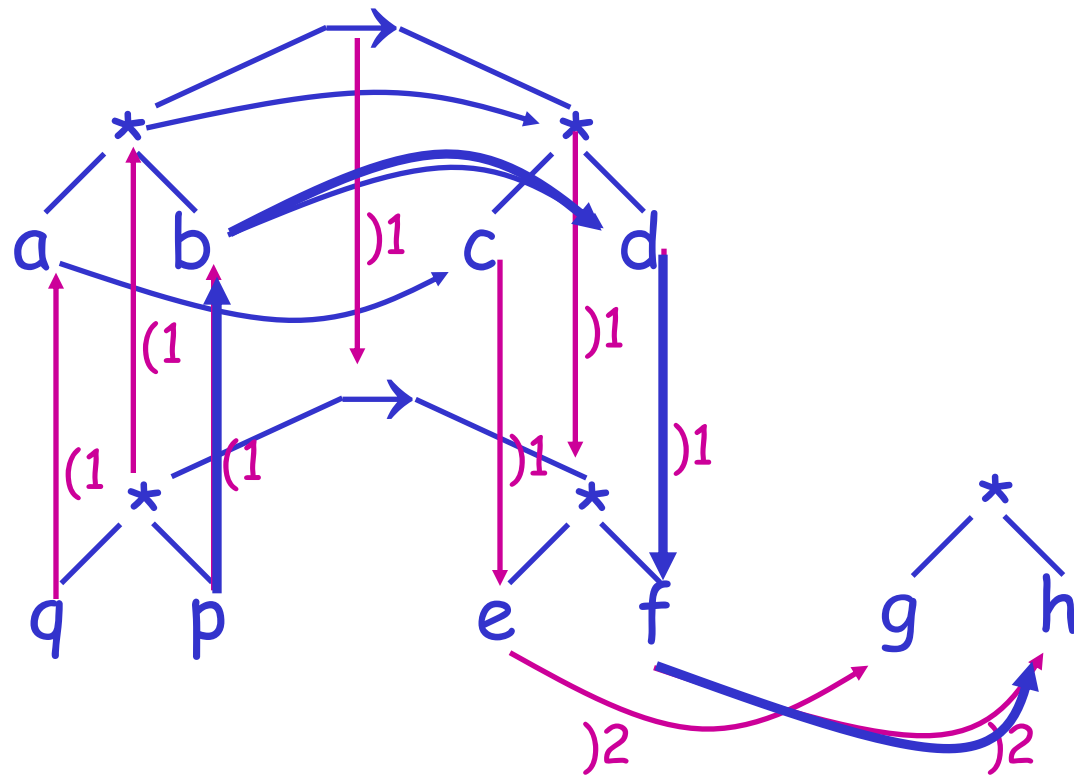
- $Q \leq Q'$  becomes  $Q \longrightarrow Q'$

- $Q \leq +i Q'$  becomes  $Q \xrightarrow{)i} Q'$

- $Q \leq -i Q'$  becomes  $Q \xleftarrow{(i} Q'$

# An Example (Stolen from RF01)

```
fun idpair (x:int*int):int*int = x in
  fun f y = idpair1 (3q, 4p) in
    let z = snd (f2 0)
```



# Two Observations

---

- *We are doing constraint copying*
  - Notice the edge from **b** to **d** got "copied" to **p** to **f**
    - 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

# CFL Reachability Grammar

---

$S ::= P N$

$P ::= M P$

|  $)i P$

|

for any  $i$   
empty

$N ::= M N$

|  $(i N$

|

for any  $i$   
empty

$M ::= (i M )i$

|  $M M$

|

|

$d$

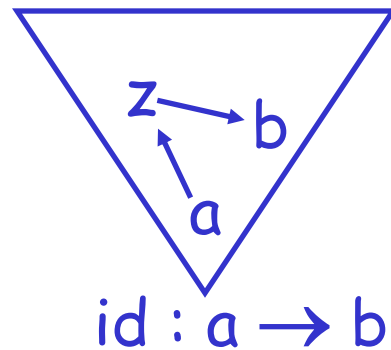
regular subtyping edge  
empty

- Paths may have **unmatched** but not **mismatched** parens

# Global Variables

---

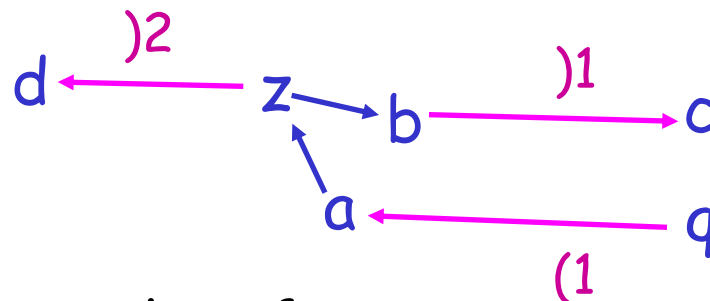
- Consider the following identity function  
`fun id(x:int):int = z := x; !z`
  - Here `z` is a global variable
- Typing of `id`, roughly speaking:



# 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 pop out at `d`

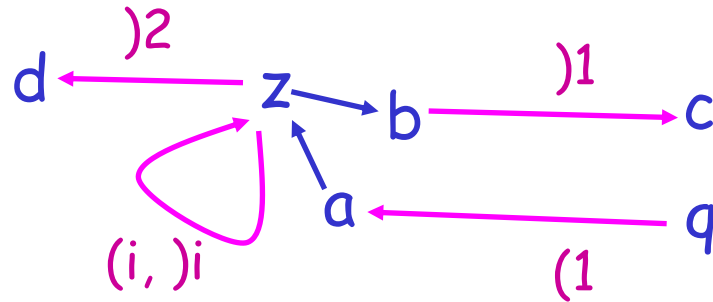
# Thou Shalt Not Quantify a Global Type (Qualifier) Variable

---

- 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 parens

# Typing Rules, Fixed

---

- Track unquantifiable vars at generalization

$$qt1 = \text{fresh}(t1) \quad qt2 = \text{fresh}(t2)$$

$$G, f: (qt1 \rightarrow^Q qt2, v), x:qt1 \dashv\vdash e : qt2' \quad qt2' \leq qt2$$

$$v = \text{free vars of } G$$

---

$$G \dashv\vdash \text{fun } f^Q (x:t1):t2 = e : (qt1 \rightarrow^Q qt2, v)$$

# Typing Rules, Fixed

---

- Add self-loops at instantiation

$$(qt, v) = G(f) \quad qt' = \text{fresh}(qt) \quad qt \leq +i qt'$$

$$v \leq +i v \quad v \leq -i v$$

---

$$G \dashv\vdash f_i : qt'$$

# 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

# Security via 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
    - (Haven't applied it to security)
- Suppose we want to apply this to a language like *C*
  - It doesn't quite look like MinML!

# 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 MinML

---

- Variables will have ref types:
  - $x : \text{ref}^Q \langle \text{contents type} \rangle$
  - Parameters as well, but r-types in fn sigs
- On rhs of assignment, add deref of variables

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

# 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 Nacula 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 {
```

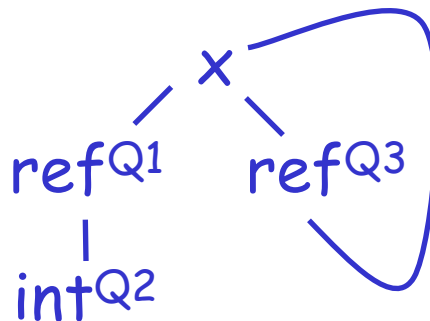
```
    int elt;
```

```
    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

# Subtyping Under Pointer Types

---

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

- Results in this rule:

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

- 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 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

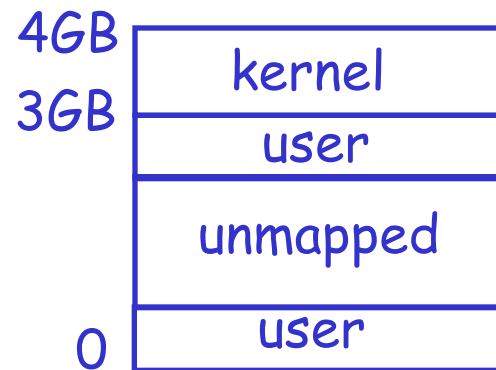
---

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:

```
int copy_from_user(void *kernel to,  
                  void *user from, int len)
```

```
int memcpy(void *kernel to,  
           void *kernel from, int len)
```

```
int x;
```

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

```
void sys_getint(int *user p) {  
    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.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 `refuser (Q s)`, it must be that  $Q \leq \text{user}$
    - This is a *well-formedness* condition on types

# Well-Formedness Constraints

---

- As a type rule

$$\frac{|--wf (Q' s) \quad Q' \leq Q}{|--wf \text{ref}^Q (Q' s)}$$

- We implicitly require all types to be well-formed
- But what about other qualifiers?
  - Not all qualifiers have these structural constraints
  - Or maybe other quals want  $Q \leq Q'$

# Well-Formedness Constraints

---

- Use conditional constraints

$$\frac{|--wf (Q' s) \quad Q \leq user \implies Q' \leq user}{|--wf ref^Q (Q' s)}$$

- “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

For all  $i$ ,  $\vdash\text{-wf } (Q_i s_i) \quad Q \leq \text{user} \implies Q_i \leq \text{user}$

$\vdash\text{-wf struct}^Q (Q_1 s_1, \dots, Q_n s_n)$

- 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_ioctl_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

---

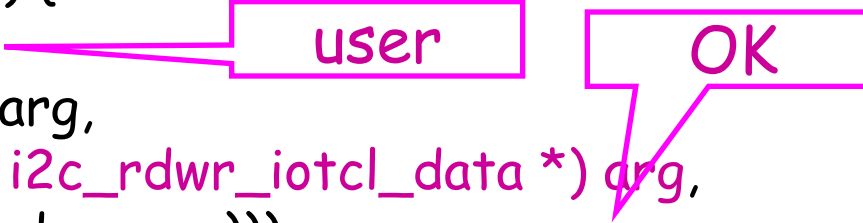
```
int copy_from_user(<kernel>, <user>, <size>):
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                unsigned long arg) {
...case I2C_RDWR:
    if (copy_from_user(&rdwr_arg,
                      (struct i2c_rdwr_ioctl_data *) arg,
                      sizeof(rdwr_arg)))
        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_ioctl_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_ioctl_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, 6 exploitable 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