**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**

- 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)**

1. Fold \( e_2 \) subsumption into rule

\[
\begin{align*}
G |-- e_1 : q_1 \rightarrow Q' & \\
& q_1 \leq q_1' \quad Q' \leq Q' \quad q_1' \leq q_2 & \rightarrow \\
& G |-- e_1 : q_1 \rightarrow Q_2 & \rightarrow \\
& G |-- e_2 : q_1 \quad q \leq q_1 & \\
& G |-- e_1 e_2 : q_2
\end{align*}
\]

2. Fold \( e_1 \) subsumption into rule

\[
\begin{align*}
q_1 \leq q_1' \quad Q' \leq Q' \quad q_1' \leq q_2 & \\
& q \leq q_1 & \\
& G |-- e_1 : q_1 \rightarrow Q' & \rightarrow \\
& G |-- e_2 : q_1 \quad q \leq q_1 & \\
& G |-- e_1 e_2 : q_2
\end{align*}
\]
Getting Rid of Subsumption (cont'd)

3. We don’t use Q, so remove that constraint

\[ q_1 \leq q \quad q \leq q_2 \]

\[ G |-- e_1 : q' \rightarrow Q' \quad G |-- e_2 : q \leq q_1 \]

\[ G |-- e_1 e_2 : q_2 \]

---

Getting Rid of Subsumption (cont'd)

4. Apply transitivity of \( \leq \)

- Remove intermediate \( q_1 \)

\[ q' = q_2 \]

\[ G |-- e_1 : q' \rightarrow Q' \quad G |-- e_2 : q \leq q' \]

\[ G |-- e_1 e_2 : q_2 \]

---

Getting Rid of Subsumption (cont'd)

5. We’re going to apply subsumption afterward, so no need to weaken \( q'' \)

\[ G |-- e_1 : q' \rightarrow Q' \quad G |-- e_2 : q \leq q' \]

\[ G |-- e_1 e_2 : q'' \]

---

Second Problem: Assumptions

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

\[ G, f : q_1 \rightarrow^2 q_2, x : q_1 |-- e : q_2 \quad q_2' \leq q_2 \]

\[ G |-- \text{fun f}^Q (x : q_1) : q_2 = e : q_1 \rightarrow^2 q_2 \]

- There’s a problem with applying this rule
  - We’re assuming that we’re given the argument type \( q_1 \) and the result type \( q_2 \)
  - 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…)

\[ G, f : ? \rightarrow^2 ?, x : ? |-- e : q_2 \quad q_2' \leq q_2 \]

\[ G |-- \text{fun f}^Q (x : q_1) : q_2 = e : q_1 \rightarrow^2 q_2 \]

- 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 $a, b, c, \ldots$ to our set of qualifiers
  - (Letters closer to $p, q, r$ will stand for constants)
- Define $\text{fresh} : t \rightarrow qt$ as
  
  - $\text{fresh}(\text{int}) = \text{int}$
  - $\text{fresh}($bool$) = \text{bool}$
  - $\text{fresh}($ref$Q) = \text{ref}$
  - $\text{fresh}(t_1 \rightarrow t_2) = \text{fresh}(t_1) \rightarrow \text{fresh}(t_2)$
  - Where $a$ is fresh

Rule for Functions

$$qt_1 = \text{fresh}(t_1) \quad qt_2 = \text{fresh}(t_2)$$

$$G : qt_1 \rightarrow \text{Q} \quad G' : qt_2 \rightarrow \text{Q}'$$

$$e \quad : \quad e' \quad : \quad qt_2 \leq qt_2'$$

$$G \vdash \text{fun} f (x : t_1) : t_2 = e : qt_1 \rightarrow Qqt_2$$

A Picture of Fresh Qualifiers

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 $Q \leq Q'$
  - $qt ::= \text{int} | \text{bool} | qt \rightarrow qt | \text{ref} qt$
- Solve by simplifying
  - Can read solution off of simplified constraints
- We'll present algorithm as a rewrite system
  - $S \Rightarrow S'$ means constraints $S$ rewrite to (simpler) constraints $S'$
### Solving Constraints: Step 1

- $S + \{ \text{int} \leq \text{int}' \} \Rightarrow S + \{ Q \leq Q' \}$
- $S + \{ \text{bool} \leq \text{bool}' \} \Rightarrow S + \{ Q \leq Q' \}$
- $S + \{ q_1 \rightarrow q_2 \leq q_1' \rightarrow q_2' \} \Rightarrow$
  - $S + \{ q_1' \leq q_1 \} + \{ q_2 \leq q_2' \} + \{ Q \leq Q' \}$
- $S + \{ \text{ref} q_1 \leq \text{ref} q_2 \} \Rightarrow$
  - $S + \{ q_1 \leq q_2 \} + \{ q_2 \leq q_1 \} + \{ Q \leq Q' \}$
- $S + \{ \text{mismatched constructors} \} \Rightarrow \text{error}$
  - 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 $q_1 = q_1'$, then $q_1$ and $q_1'$ 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

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

---

### Constraints as Graphs

- $a_0 \leq a_1$
- $a_2 \leq a_3$
- $a_4 \leq a_5$
- $a_6 \leq a_7$
- $a_8 \leq a_9$

---

### 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
  - \( p \cup q \leq r \) 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, it turns out we can use a really simple algorithm to check satisfiability of constraints over \( Q \)

Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?

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, \( q_t \) represents \( q_t \) and all of \( q_t \)'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
  - 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 \( \rightarrow \) tainted
    - Fine but untainted data passed in becomes tainted
  - untainted \( \rightarrow \) untainted
    - Fine but can’t pass in tainted data
  - untainted \( \rightarrow \) tainted
    - Not too useful
  - tainted \( \rightarrow \) untainted
    - Impossible

Function Calls and Context-Sensitivity

- 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 \( \forall a. a \rightarrow a \)

The Observation of Parametric Polymorphism

- Type inference on id yields a proof like this:

\[
\begin{equation}
\text{id : } a \rightarrow a
\end{equation}
\]

- 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

![Diagram]

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

![Diagram]

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
  - $x : \forall a. qt$ 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
  - \( a \leq^+ b \)
  - 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

\[
\begin{align*}
qt1 & = \text{fresh}(t1) & qt2 & = \text{fresh}(t2) \\
G, f : qt1 \rightarrow^{Q} qt2, x : qt1 \rightarrow^{e} qt2' & \quad qt2' : qt2 \\
G \vdash f : qt1 \rightarrow^{Q} qt2
\end{align*}
\]

• This is not quite the right rule, yet...

Resolving Instantiation Constraints

• Just like subtyping, reduce to only qualifiers
  - \( S \times \{ \text{int}^{Q} \rightarrow \text{int}^{Q'} \} \Rightarrow S \times \{ Q \rightarrow Q' \} \)
  - \( p \) stands for either + or -

\[
\begin{align*}
S \times \{ qt1 \rightarrow^{Q} qt2 \rightarrow^{Q'} qt2' \} & \Rightarrow \\
S \times \{ qt1 \rightarrow^{(-p)} qt1 \} \times (qt2 \rightarrow qt2') & \Rightarrow \\
S & \times \{ Q \rightarrow Q' \} \\
\text{Here } (-) \text{ is } - & \text{ and } (.) \text{ is } +
\end{align*}
\]
Instantiation Constraints as Graphs

- Three kinds of edges
  - $Q \leq Q'$ becomes $Q \rightarrow Q'$
  - $Q \leq +i Q'$ becomes $Q \rightarrow (i \rightarrow Q'$
  - $Q \leq -i Q'$ becomes $Q \rightarrow (i \leftarrow Q'$

An Example (Stolen from RF01)

```haskell
fun idpair (x:int*:int*:int*) = x in
fun f y = idpair (31, 4p) in
let z = snd (f3, O)
```

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

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 \textit{id} to \textit{q} inside of a function

\[
\begin{array}{c}
d \rightarrow \overset{2}{z} \rightarrow \overset{1}{b} \\
\overset{1}{a} \rightarrow \overset{1}{q}
\end{array}
\]

- And then another function returns \textit{z}
- Uh oh! (1)\textit{2} is not a valid flow path
- But \textit{q} may certainly pop out at \textit{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 \textit{z} at each instantiation

- Solution: Don’t do that!

**Our Example Again**

\[
\begin{array}{c}
d \rightarrow \overset{2}{z} \rightarrow \overset{1}{b} \\
\overset{1}{a} \rightarrow \overset{1}{q}
\end{array}
\]

- We want anything flowing into \textit{z}, on any path, to flow out in any way
- Add a self-loop to \textit{z} that consumes any mismatched parens

**Typing Rules, Fixed**

- Track unquantifiable vars at generalization

\[
\begin{array}{l}
\text{qt}_1 = \text{fresh}(t_1) \\
\text{qt}_2 = \text{fresh}(t_2)
\end{array}
\]

\[
G, f : (\text{qt}_1 \rightarrow \text{qt}_2, v), x : \text{qt}_1 \vdash e : \text{qt}_2 \\
\quad v = \text{free vars of } G
\]

\[
G \vdash f : \text{qt}_1
\]

\[
G \vdash e : (\text{qt}_1 \rightarrow \text{qt}_2, v)
\]

**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