A Theory of Type Qualifiers

Jeffrey S. Foster, Manuel Fähndrich, Alexander Aiken

PLDI 1999

Paper presentation by
Georg Schmid and Samuel Grütter

May 20, 2015

Overview

- Many languages support a range of *type qualifiers* – they enable a simple, yet useful form of subtyping.
  - E.g. const on references, nonzero on integers.
- Authors present a framework for adding type qualifiers to λ-calculi.
- In particular, they show how to
  - extend the typing rules, and
  - support type inference –
  - even in the polymorphic case.

Motivation

- Use-case: Analyze C sources and infer additional const qualifiers

Motivation

- Use-case: Analyze C sources and infer additional const qualifiers
- This rectifies one particular peculiarity of C's type system:

```
int *id1(int *x) { return x; }
const int *id2(const int *x) { return x; }
```

→ Inference of const qualifiers removes need for multiple versions of same procedure

Preliminaries
Qualifiers

Definition 1 A type qualifier q is *positive* (*negative*) if $\tau \preceq q\ \tau$ ($q\ \tau \preceq \tau$) for any type τ.

- E.g. const is a positive qualifier, while nonzero is negative.

Preliminaries

Qualifiers

Definition 1 A type qualifier q is *positive* (*negative*) if $\tau \preceq q\ \tau$ $(q\ \tau \preceq \tau)$ for any type τ.

- E.g. const is a positive qualifier, while nonzero is negative.

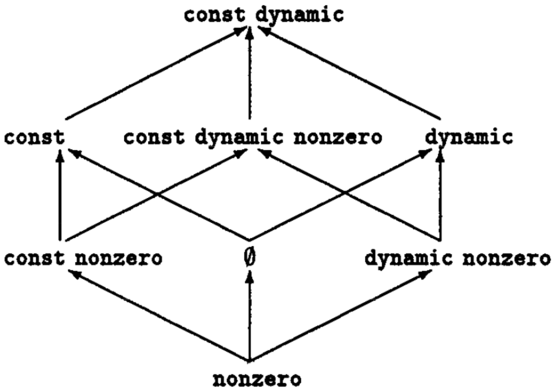
Definition 2 (Qualifier lattice) Each positive qualifier q defines a two-point lattice $L_q = \bot_q \sqsubseteq q$. Each negative qualifier q defines a two-point lattice $L_q = q \sqsubseteq \top_q$. The *qualifier lattice* L is defined by $L = L_{q_1} \times \cdots \times L_{q_n}$. We write $\bot$ and $\top$ for the bottom and top elements of L.

$\Rightarrow$ Question: What is a two-point lattice?

Preliminaries

Qualifiers (2)

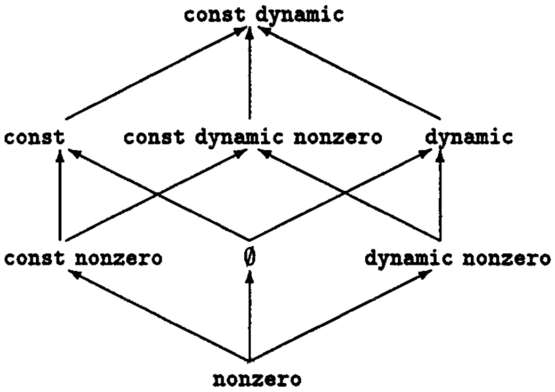
- Let's look at the lattice of positive qualifiers const, dynamic and (negative) nonzero:



Preliminaries

Qualifiers (2)

- Let's look at the lattice of positive qualifiers const, dynamic and (negative) nonzero:

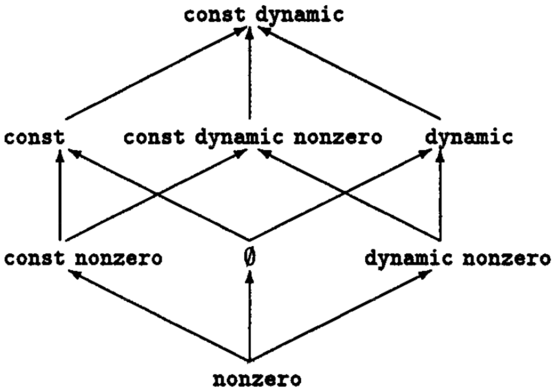


- Question: What is the $\perp$ of this lattice? What is $\top$?

Preliminaries

Qualifiers (2)

- Let's look at the lattice of positive qualifiers const, dynamic and (negative) nonzero:



- Question: What is the $\perp$ of this lattice? What is $\top$?
  Answer: $\perp =$ "nonzero", $\top =$ "const dynamic"

Preliminaries
Qualifiers (3)

- How should we interpret $\tau \preceq \mathsf{const}\,\tau$?

Preliminaries
Qualifiers (3)

- How should we interpret $\tau \preceq \mathsf{const}\,\tau$?
  - Assuming $\preceq$ corresponds to $\subseteq$, "int $\preceq$ const int" means "(set of all ints) $\subseteq$ (set of all const ints)".
    So each int is constant, all data immutable!?

Preliminaries
Qualifiers (3)

- How should we interpret $\tau \preceq \mathrm{const}\,\tau$?
  - Assuming $\preceq$ corresponds to $\subseteq$, "int $\preceq$ const int" means "(set of all ints) $\subseteq$ (set of all const ints)". So each int is constant, all data immutable!?
  - Rather, think of const as *readonly*. Alternatively, consider the qualifier writable $\tau \preceq \tau$ which is dual notation for const.

Preliminaries

Qualifiers (3)

- How should we interpret $\tau \preceq \text{const}\,\tau$?
  - Assuming $\preceq$ corresponds to $\subseteq$, "int $\preceq$ const int" means "(set of all ints) $\subseteq$ (set of all const ints)". So each int is constant, all data immutable!?
  - Rather, think of const as *readonly*. Alternatively, consider the qualifier writable $\tau \preceq \tau$ which is dual notation for const.

$\Rightarrow$ Intuition: We *lose capabilities* as we move up the lattice.

Preliminaries
Qualifiers (4)

- With each qualifier q_i we associate a qualifier lattice element $\neg q_i$, where

$$\neg q_i := (\top_1, \ldots, \top_{i-1}, \bot_i, \top_{i+1}, \ldots, \top_n) \text{ when } q_i \text{ is } \textit{positive},$$

$$\neg q_i := (\bot_1, \ldots, \bot_{i-1}, \top_i, \bot_{i+1}, \ldots, \bot_n) \text{ when } q_i \text{ is } \textit{negative}.$$

Preliminaries
Qualifiers (4)

- With each qualifier q_i we associate a qualifier lattice element $\neg q_i$, where

$$\neg q_i := (\top_1, \ldots, \top_{i-1}, \bot_i, \top_{i+1}, \ldots, \top_n) \text{ when } q_i \text{ is } \textit{positive},$$

$$\neg q_i := (\bot_1, \ldots, \bot_{i-1}, \top_i, \bot_{i+1}, \ldots, \bot_n) \text{ when } q_i \text{ is } \textit{negative}.$$

- Question: What is $\neg$*nonzero*?

Preliminaries
Qualifiers (4)

- With each qualifier q_i we associate a qualifier lattice element $\neg q_i$, where

$$\neg q_i := (\top_1, \ldots, \top_{i-1}, \bot_i, \top_{i+1}, \ldots, \top_n) \text{ when } q_i \text{ is } \textit{positive},$$

$$\neg q_i := (\bot_1, \ldots, \bot_{i-1}, \top_i, \bot_{i+1}, \ldots, \bot_n) \text{ when } q_i \text{ is } \textit{negative}.$$

- Question: What is $\neg$*nonzero*?
  Answer: $(\bot_{\text{const}}, \bot_{\text{dynamic}}, \top_{\text{nonzero}}) =$ "$\emptyset$"

Preliminaries
Source language

Framework extends a given source language, e.g.

- Expressions of λ-calculus with if and let:

$$
\begin{array}{lcll}
e & ::= & v & \\
 & | & e_1\ e_2 & \\
 & | & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{fi} & \\
 & | & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{ni} & \\
v & ::= & x & x \in PVar \\
 & | & n & n \in \mathbb{Z} \\
 & | & \lambda x.e & \\
\tau & ::= & \alpha & \\
 & | & int & \\
 & | & \tau \rightarrow \tau &
\end{array}
$$

Qualified types

Pairing ordinary types with elements of the qualifier lattice yields *qualified types* for our sample language:

$$\begin{array}{rcl} \rho & ::= & Q\,\tau \\ \tau & ::= & \alpha \mid \mathit{int} \mid (\rho_1 \to \rho_2) \\ Q & ::= & \kappa \mid l \end{array}$$

where κ is a type qualifier variable and
l is an element of qualifier lattice L.

Along with a pair of subtyping rules that naturally connects the type system to our qualifier lattice:

$$\frac{\vdash Q_1 \sqsubseteq Q_2}{\vdash Q_1\ \mathit{int} \preceq Q_2\ \mathit{int}} \qquad \textbf{(SubInt)}$$

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \quad \vdash \rho_2 \preceq \rho_1 \quad \vdash \rho_1' \preceq \rho_2'}{\vdash Q_1\ (\rho_1 \to \rho_1') \preceq Q_2\ (\rho_2 \to \rho_2')} \qquad \textbf{(SubFun)}$$

Qualifier annotations and assertions

- When inferring types, we need to decide on a type qualifier for the outermost type constructor.
  - *Qualifier annotations* are added as a syntactic helper
  - Additionally, dual notion of *qualifier assertions*
- This requires extensions of both syntax and typing rules:

$$
\begin{array}{lll|l}
e & ::= & \dots & \dfrac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash e|_l : Q\,\tau} \quad \textbf{(Assert)} \\
 & \mid & e|_l & \\
 & \mid & l\,e & \dfrac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash l\,e : l\,\tau} \quad \textbf{(Annot)}
\end{array}
$$

Qualifier annotations and assertions

- When inferring types, we need to decide on a type qualifier for the outermost type constructor.
  - *Qualifier annotations* are added as a syntactic helper
  - Additionally, dual notion of *qualifier assertions*
- This requires extensions of both syntax and typing rules:

$$
\begin{array}{rcl|c}
e & ::= & \dots & \dfrac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash e|_l : Q\,\tau} \quad (\text{Assert}) \\
& | & e|_l & \\
& | & l\,e & \dfrac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash l\,e : l\,\tau} \quad (\text{Annot})
\end{array}
$$

- Both enforce e's top-level qualifier Q to be upper-bounded by l, i.e. $Q \sqsubseteq l$
  - Question: What's the difference between the two?

Qualified type systems

$$A \vdash e : \tau \quad \Rightarrow \quad A \vdash e : \rho$$

Extending the source language's type checking system with qualified types leads to a *qualified type system*.

Qualified type systems

Typing rules of sample language (1)

$$\frac{A \vdash e : \rho \quad \vdash \rho \preceq \rho'}{A \vdash e : \rho'} \quad \text{(Sub)}$$

$$\frac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash e|_l : Q\,\tau} \quad \text{(Assert)}$$

$$\frac{A \vdash e : Q\,\tau \quad \vdash Q \sqsubseteq l}{A \vdash l\,e : l\,\tau} \quad \text{(Annot)}$$

$$\frac{}{A \vdash n : \bot\,\mathit{int}} \quad \text{(Int)}$$

$$\frac{}{A \vdash x : A(x)} \quad \text{(Var)}$$

Qualified type systems

Typing rules of sample language (2)

$$\frac{A[x \mapsto \rho_x] \vdash e : \rho}{A \vdash \lambda x.e : \bot\ (\rho_x \to \rho)} \quad \text{(Lam)}$$

$$\frac{A \vdash e_1 : Q\ (\rho_2 \to \rho) \quad A \vdash e_2 : \rho_2}{A \vdash e_1\ e_2 : \rho} \quad \text{(App)}$$

$$\frac{A \vdash e_1 : Q\ \mathit{int} \quad A \vdash e_2 : \rho \quad A \vdash e_2 : \rho}{A \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{fi} : \rho} \quad \text{(If)}$$

$$\frac{A \vdash e_1 : \rho_1 \quad A[x \mapsto \rho_1] \vdash e_2 : \rho_2}{A \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{ni} : \rho_2} \quad \text{(Let)}$$

Qualified type systems
Correspondence

- Type qualifiers should only refine type information,
  but *not* modify type structure.

$\Rightarrow$ Correspondence of the original and resp. qualified type system

Qualified type systems
Correspondence

- Type qualifiers should only refine type information,
  but *not* modify type structure.

$\Rightarrow$ Correspondence of the original and resp. qualified type system

- Helpers:
  - $\text{strip}(\cdot) : \mathit{QTyp} \to \mathit{Typ}$ eliminates qualifiers from types
  - $\perp(\cdot) : \mathit{Typ} \to \mathit{QTyp}$ introduces $\perp$ qualifiers in types

Qualified type systems
Correspondence

- Type qualifiers should only refine type information,
  but *not* modify type structure.

$\Rightarrow$ Correspondence of the original and resp. qualified type system

- Helpers:
  - $\mathsf{strip}(\cdot) : \mathit{QTyp} \to \mathit{Typ}$ eliminates qualifiers from types

    Example: $\mathsf{strip}(\overline{\bot(\mathsf{const\ int} \to \mathsf{nonzero\ int})}) = \overline{(\mathsf{int} \to \mathsf{int})}$
  - $\bot(\cdot) : \mathit{Typ} \to \mathit{QTyp}$ introduces $\bot$ qualifiers in types

Qualified type systems
Correspondence

- Type qualifiers should only refine type information, but *not* modify type structure.

$\Rightarrow$ Correspondence of the original and resp. qualified type system

- Helpers:
  - $\mathrm{strip}(\cdot) : \mathit{QTyp} \rightarrow \mathit{Typ}$ eliminates qualifiers from types

    Example: $\mathrm{strip}(\,\overline{\bot(\mathrm{const\ int} \rightarrow \mathrm{nonzero\ int})}\,) = \overline{(\mathrm{int} \rightarrow \mathrm{int})}$
  - $\bot(\cdot) : \mathit{Typ} \rightarrow \mathit{QTyp}$ introduces $\bot$ qualifiers in types

    Example $\bot(\,\overline{(\mathrm{int} \rightarrow \mathrm{int})}\,) = \overline{\bot(\bot\ \mathrm{int} \rightarrow \bot\ \mathrm{int})}$

Qualified type systems
Correspondence

- Type qualifiers should only refine type information, but *not* modify type structure.

$\Rightarrow$ Correspondence of the original and resp. qualified type system

- Helpers:
  - $\mathsf{strip}(\cdot) : \mathit{QTyp} \to \mathit{Typ}$ eliminates qualifiers from types
    Example: $\mathsf{strip}(\,\overline{\bot(\mathsf{const\ int} \to \mathsf{nonzero\ int})}\,) = \overline{(\mathsf{int} \to \mathsf{int})}$
  - $\bot(\cdot) : \mathit{Typ} \to \mathit{QTyp}$ introduces $\bot$ qualifiers in types
    Example $\bot(\,\overline{(\mathsf{int} \to \mathsf{int})}\,) = \overline{\bot(\bot\ \mathsf{int} \to \bot\ \mathsf{int})}$
  - $\mathsf{strip}(\cdot) : \mathit{Expr} \to \mathit{Expr}$ eliminates annotations & assertions
  - $\bot(\cdot) : \mathit{Expr} \to \mathit{Expr}$ introduces $\bot$ qualifier annotations everywhere

Qualified type systems
Correspondence (ctd.)

Observation 1 Let $\vdash_S$ be the judgment relation of the type system of the simply-typed lambda calculus, and let $\vdash$ be the judgment relation of the type system given in Figure 4. Then

- If $\emptyset \vdash_S e : \tau$, then $\emptyset \vdash \bot(e) : \bot(\tau)$.
- If $\emptyset \vdash e' : \rho$, then $\emptyset \vdash_S \mathsf{strip}(e') : \mathsf{strip}(\rho)$.

Qualified type systems
Correspondence (ctd.)

Observation 1 Let $\vdash_S$ be the judgment relation of the type system of the simply-typed lambda calculus, and let $\vdash$ be the judgment relation of the type system given in Figure 4. Then

- If $\emptyset \vdash_S e : \tau$, then $\emptyset \vdash \bot(e) : \bot(\tau)$.
- If $\emptyset \vdash e' : \rho$, then $\emptyset \vdash_S \mathsf{strip}(e') : \mathsf{strip}(\rho)$.

- Question: Could we use $\top$ instead? Any other qualifier?

Qualifier semantics

- Restrictions on usage of qualifiers can be expressed
  a) using qualifier assertions ($\rightarrow$ program transformation), or
  b) by modifying typing rules.
- Arbitrary modifications may render the type system unsound!

Qualifier semantics
const example

- Let's encode the semantics of a const qualifier!

Qualifier semantics
const example

- Let's encode the semantics of a const qualifier!
- Only makes sense in presence of *references*:

$$
\begin{array}{lcl}
e & ::= & \cdots \mid \mathbf{ref}\ e \mid !e \mid e_1 := e_2 \\
v & ::= & \cdots \mid () \\
\tau & ::= & \cdots \mid \mathit{ref}(\rho) \mid \mathit{unit}
\end{array}
$$

Qualifier semantics

const example (2)

- We first introduce a subtyping rule for ref:

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \tau_1 \preceq \tau_2}{\vdash Q_1\ \mathit{ref}(\tau_1) \preceq Q_2\ \mathit{ref}(\tau_2)}$$

Qualifier semantics

const example (2)

- We first introduce a subtyping rule for ref:

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \tau_1 \preceq \tau_2}{\vdash Q_1\ ref(\tau_1) \preceq Q_2\ ref(\tau_2)}$$

- Unsound in the presence of subtyping!

```
let x = ref(nonzero 37) in
let y = x in
   y := 0;
   (!x)|nonzero
ni ni
```

Qualifier semantics
const example (2)

- We first introduce a subtyping rule for ref:

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \tau_1 \preceq \tau_2}{\vdash Q_1\ ref(\tau_1) \preceq Q_2\ ref(\tau_2)}$$

- Unsound in the presence of subtyping!

```
1  let x = ref(nonzero 37) in
2  let y = x in
3     y := 0;
4     (!x)|nonzero
5  ni ni
```

$\Rightarrow$ Note that lines 3 and 4 type check, e.g.
#3: $A[x, y \mapsto \perp \text{ ref nonzero int}] \vdash y : \neg\text{nonzero int}$ (by (Sub))
#4: $A[x, y \mapsto \perp \text{ ref nonzero int}] \vdash\ !x : \text{nonzero int}$ (unchanged)

Qualifier semantics
const example (3)

- Requiring refs' argument type to be invariant fixes our problem.

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \rho_1 = \rho_2}{\vdash Q_1\ \mathit{ref}(\rho_1) \preceq Q_2\ \mathit{ref}(\rho_2)} \qquad \textbf{(SubRef)}$$

Qualifier semantics
const example (4)

- How do we enforce the semantics of const?

Qualifier semantics
const example (4)

- How do we enforce the semantics of const?
- As mentioned before, there are two ways to encode such restrictions:
    a) Program transformation:
    Replace every assignment $e_1 := e_2$ by $e_1|_{\neg\mathsf{const}} := e_2$.

Qualifier semantics

const example (4)

- How do we enforce the semantics of const?
- As mentioned before, there are two ways to encode such restrictions:
  a) Program transformation:
     Replace every assignment $e_1 := e_2$ by $e_1 |_{\neg\text{const}} := e_2$.
  b) Modify the relevant typing rule(s):

$$\frac{A \vdash e_1 : Q\ ref(\rho_2) \quad A \vdash e_2 : \rho_2}{A \vdash e_1 := e_2 : \bot\ unit} \quad \text{(Assign)}$$

$$\Downarrow$$

$$\frac{A \vdash e_1 : \neg\mathbf{const}\ ref(\rho_2) \quad A \vdash e_2 : \rho_2}{A \vdash e_1 := e_2 : \bot\ unit} \quad \text{(Assign}')$$

Type Inference and Qualifier Polymorphism

Outline

1. Forget about qualifiers
   Study type inference on simply typed lambda calculus
2. Add qualifiers
3. Add polymorphism

Step 1: Forget about qualifiers

$$\frac{}{A \vdash n : int} \quad \text{(Int)}$$

$$\frac{}{A \vdash x : A(x)} \quad \text{(Var)}$$

$$\frac{A[x \to \tau_1] \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \to \tau_2} \quad \text{(Lam)}$$

$$\frac{A \vdash e_1 : \tau_2 \to \tau \qquad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \tau} \quad \text{(App)}$$

$$\frac{A \vdash e_1 : int \qquad A \vdash e_2 : \tau \qquad A \vdash e_3 : \tau}{A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \tau} \quad \text{(If)}$$

$$\frac{A \vdash e_1 : \tau_1 \qquad A[x \to \tau_1] \vdash e_2 : \tau_2}{A \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ ni} : \tau_2} \quad \text{(Let)}$$

The problem

Say we want to typecheck $\lambda x. <$ some expression $>$.

Use

$$\frac{A[x \to \tau_1] \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \to \tau_2} \quad \text{(Lam)}$$

but what τ_1 should we put into A?

Solutions

- Require users to **explicitly state argument type**
  $\lambda(x : \tau_1).e$ instead of $\lambda x.e$
- **Local type inference** using *expected types* (e.g. Scala)
  `List(1, 2, 3).map(x => x+1)`
- **Global type inference** (e.g. ML)

  $\textbf{let } f = \lambda x.\ x + 1 \textbf{ in}$

  $\ldots$

  $myList.map(f)$

  $\ldots$

  $otherList.map(f)$

$\Longrightarrow$ Study *Constraint-Based Typing*

Constraint-Based Typing

Approach:

1. Typecheck using the inference rules
   Whenever we don't know what type to choose, pick a fresh type variable α and record the constraints it has to satisfy.

2. Unify the constraints

Constraints = set of equality constraints:

$$C ::= \{\tau_1 = \tau_2\} \mid C_1 \cup C_2$$

New typing judgement:

$$A \vdash e : \tau; C$$

"In environment A, e has type τ for all solutions of constraints C."

Constraint-Based Typing Rules

$$\frac{}{A \vdash n : int; \emptyset} \quad \text{(Int)}$$

$$\frac{}{A \vdash x : A(x); \emptyset} \quad \text{(Var)}$$

$$\frac{\alpha \text{ fresh} \quad A[x \to \alpha] \vdash e : \tau_2; C}{A \vdash \lambda x.e : \alpha \to \tau_2; C} \quad \text{(Lam)}$$

$$\frac{\alpha \text{ fresh} \quad A \vdash e_1 : \tau_1; C_1 \quad A \vdash e_2 : \tau_2; C_2}{A \vdash e_1 e_2 : \alpha; C_1 \cup C_2 \cup \{\tau_1 = (\tau_2 \to \alpha)\}} \quad \text{(App)}$$

$$\frac{A \vdash e_1 : \tau_1; C_1 \quad A \vdash e_2 : \tau_2; C_2 \quad A \vdash e_3 : \tau_3; C_3}{\begin{array}{c} A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \tau_2; \\ C_1 \cup C_2 \cup C_3 \cup \{\tau_1 = int\} \cup \{\tau_2 = \tau_3\} \end{array}} \quad \text{(If)}$$

$$\frac{A \vdash e_1 : \tau_1; C_1 \quad A[x \to \tau_1] \vdash e_2 : \tau_2; C_2}{A \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ ni} : \tau_2; C_1 \cup C_2} \quad \text{(Let)}$$

Unification

- A solution to a set of constraints C is a substitution

$$S : \mathit{TVar} \rightarrow \mathit{Typ}$$

  mapping type variables to types containing no type variables, such that all equalities in SC hold.

$\Rightarrow$ Goal of unification: Given C, find S.

Unification Algorithm

def unify(C) = C **match** {
  case $\emptyset \Rightarrow$ identity *// empty substitution*
  case $c_1 \cup C_{rest} \Rightarrow c_1$ **match** {
    case $((\tau_1 \rightarrow \tau_2) = (\tau_3 \rightarrow \tau_4)) \Rightarrow \text{unify}(C_{rest} \cup \{\tau_1 = \tau_3\} \cup \{\tau_2 = \tau_4\})$
    case $(\tau = \tau) \Rightarrow \text{unify}(C_{rest})$
    case $(\tau = \alpha)$ **if** $\alpha \notin FV(\tau) \Rightarrow \text{unify}([\alpha \rightarrow \tau]C_{rest}) \circ [\alpha \rightarrow \tau]$
    case $(\alpha = \tau)$ **if** $\alpha \notin FV(\tau) \Rightarrow \text{unify}([\alpha \rightarrow \tau]C_{rest}) \circ [\alpha \rightarrow \tau]$
    case _ $\Rightarrow$ fail
  }
}

Step 2: Add qualifiers

Without qualifiers	With qualifiers
$A \vdash e : \tau; C$	$A \vdash e : \rho; C$
$\tau ::= \alpha \mid int \mid \tau \to \tau$	$\rho ::= Q\,\tau$ $\tau ::= \alpha \mid int \mid (\rho_1 \to \rho_2)$ $Q ::= \kappa \mid l$
$C ::= \{\tau_1 = \tau_2\} \mid C_1 \cup C_2$	$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$

Step 2: Add qualifiers

Without qualifiers	With qualifiers
$A \vdash e : \tau; C$	$A \vdash e : \rho; C$
$\tau ::= \alpha \mid int \mid \tau \to \tau$	$\rho ::= Q\,\tau$ $\tau ::= \alpha \mid int \mid (\rho_1 \to \rho_2)$ $Q ::= \kappa \mid l$
$C ::= \{\tau_1 = \tau_2\} \mid C_1 \cup C_2$	$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$

Q: Why is there no $\rho_1 = \rho_2$ in the grammar for C?

Step 2: Add qualifiers

Without qualifiers	With qualifiers
$A \vdash e : \tau; C$	$A \vdash e : \rho; C$
$\tau ::= \alpha \mid int \mid \tau \to \tau$	$\rho ::= Q\,\tau$ $\tau ::= \alpha \mid int \mid (\rho_1 \to \rho_2)$ $Q ::= \kappa \mid l$
$C ::= \{\tau_1 = \tau_2\} \mid C_1 \cup C_2$	$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$

Q: Why is there no $\rho_1 = \rho_2$ in the grammar for C?

A: Because we can just use $\{\rho_1 \preceq \rho_2\} \cup \{\rho_2 \preceq \rho_1\}$ instead.

Constraint-Based Typing Rules With Qualifiers

$$\frac{}{A \vdash n : int; \emptyset} \quad \text{(Int)}$$

$$\frac{}{A \vdash x : A(x); \emptyset} \quad \text{(Var)}$$

$$\frac{\alpha \text{ fresh} \quad \kappa \text{ fresh} \quad A[x \to \kappa\alpha] \vdash e : \rho_2; C}{A \vdash \lambda x.e : \kappa\alpha \to \tau_2; C} \quad \text{(Lam)}$$

$$\frac{\alpha \text{ fresh} \quad \kappa, \kappa' \text{ fresh} \quad A \vdash e_1 : \rho_1; C_1 \quad A \vdash e_2 : \rho_2; C_2}{A \vdash e_1 e_2 : \alpha; C_1 \cup C_2 \cup \{\rho_1 = \kappa(\rho_2 \to \kappa'\alpha)\}} \quad \text{(App)}$$

$$\frac{\kappa \text{ fresh} \quad A \vdash e_1 : \rho_1; C_1 \quad A \vdash e_2 : \rho_2; C_2 \quad A \vdash e_3 : \rho_3; C_3}{\begin{array}{c} A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \rho_2; \\ C_1 \cup C_2 \cup C_3 \cup \{\rho_1 = \kappa int\} \cup \{\rho_2 = \rho_3\} \end{array}} \quad \text{(If)}$$

$$\frac{A \vdash e_1 : \rho_1; C_1 \quad A[x \to \rho_1] \vdash e_2 : \rho_2; C_2}{A \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ ni} : \rho_2; C_1 \cup C_2} \quad \text{(Let)}$$

Question

In an if-expression, what if the then-branch has type `const int` and the else-branch has type `int`: Can we still typecheck the expression?

- (If) rule from previous slide:

$$\frac{\kappa \text{ fresh} \quad A \vdash e_1 : \rho_1; C_1 \quad A \vdash e_2 : \rho_2; C_2 \quad A \vdash e_3 : \rho_3; C_3}{\begin{array}{c} A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \rho_2; \\ C_1 \cup C_2 \cup C_3 \cup \{\rho_1 = \kappa int\} \cup \{\rho_2 = \rho_3\} \end{array}}$$

Question

In an if-expression, what if the then-branch has type `const int` and the else-branch has type `int`: Can we still typecheck the expression?

- (If) rule from previous slide:

$$\frac{\kappa \text{ fresh} \quad A \vdash e_1 : \rho_1; C_1 \quad A \vdash e_2 : \rho_2; C_2 \quad A \vdash e_3 : \rho_3; C_3}{\begin{array}{c} A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \rho_2; \\ C_1 \cup C_2 \cup C_3 \cup \{\rho_1 = \kappa int\} \cup \{\rho_2 = \rho_3\} \end{array}}$$

- A better version:

$$\frac{\alpha, \kappa, \kappa' \text{ fresh} \quad A \vdash e_1 : \rho_1; C_1 \quad A \vdash e_2 : \rho_2; C_2 \quad A \vdash e_3 : \rho_3; C_3}{\begin{array}{c} A \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} : \kappa'\alpha; \\ C_1 \cup C_2 \cup C_3 \cup \{\rho_1 = \kappa int\} \cup \{\rho_2 \preceq \kappa'\alpha\} \cup \{\rho_3 \preceq \kappa'\alpha\} \end{array}}$$

Unification Algorithm

- Reminder:

$$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$$

- **First phase:** Repeat
  - $\{(Q_1\tau_1 \rightarrow Q_2\tau_2) \preceq (Q_3\tau_3 \rightarrow Q_4\tau_4)\} \cup C_{rest}$
    $\Rightarrow C_{rest} \cup \{Q_3\tau_3 \preceq Q_1\tau_1\} \cup \{Q_2\tau_2 \preceq Q_4\tau_4\})$
  - $\{Q\tau \preceq Q'\tau\} \cup C_{rest} \quad \Rightarrow \quad C_{rest} \cup \{Q \sqsubseteq Q'\}$
  - $\{Q\tau \preceq Q'\alpha\} \cup C_{rest} \quad \Rightarrow \quad [\alpha \rightarrow \tau]C_{rest} \cup \{Q \sqsubseteq Q'\}$
  - $\{Q\alpha \preceq Q'\tau\} \cup C_{rest} \quad \Rightarrow \quad [\alpha \rightarrow \tau]C_{rest} \cup \{Q \sqsubseteq Q'\}$

  until only lattice constraints are left.
- **Second phase:**
  All constraints now are of the form $\kappa \sqsubseteq L$, $L \sqsubseteq \kappa$, or $L_1 \sqsubseteq L_2$.
  Solve in linear time as described in [HR97].

Step 3: Add Polymorphism

Note: The goal is to be polymorphic in *qualifiers*, not in *types*.

Without polymorphism	**With polymorphism**
A contains (x, ρ) tuples	A contains (x, σ) tuples
$A \vdash e : \rho; C$	$A \vdash e : \rho; C$
$\rho ::= Q\,\tau$ $\tau ::= \alpha \mid int \mid (\rho_1 \to \rho_2)$ $Q ::= \kappa \mid l$	$\sigma ::= \forall\vec{\kappa}.\rho \backslash C$ $\rho ::= Q\,\tau$ $\tau ::= \alpha \mid int \mid \rho_1 \to \rho_2$ $Q ::= \kappa \mid l$

σ: Type scheme carrying constraints

Back to C: Example

```
const int * max_const(const int * p1, const int * p2) {
    if (*p1 < *p2) {
        return p2;
    } else {
        return p1;
    }
}

int * max_nonconst(int * p1, int * p2) { EXACTLY THE SAME BODY }

int main() {
    int v1 = 8;
    int v2 = 5;

    const int * p1_const = &v1;
    const int * p2_const = &v2;
    int bigger = *(max_const(p1_const, p2_const));
    printf("%d\n", bigger);

    int * p1_nonconst = &v1;
    int * p2_nonconst = &v2;
    int * p_bigger = max_nonconst(p1_nonconst, p2_nonconst);
    (*p_bigger)--;
}
```

Qualifier Polymorphism

In C:

```
const int * max_const(const int * p1, const int * p2)
int * max_nonconst(int * p1, int * p2)
```

Translated to the language of the paper:

max_const : $(\textbf{const}\ \mathit{ref}(\bot \mathit{int})) \to (\textbf{const}\ \mathit{ref}(\bot \mathit{int})) \to (\textbf{const}\ \mathit{ref}(\bot \mathit{int}))$

max_nonconst : $(\bot \mathit{ref}(\bot \mathit{int})) \to (\bot \mathit{ref}(\bot \mathit{int})) \to (\bot \mathit{ref}(\bot \mathit{int}))$

Now with qualifier polymorphism:

max: $\forall \kappa.(\kappa \mathit{ref}(\bot \mathit{int})) \to (\kappa \mathit{ref}(\bot \mathit{int})) \to (\kappa \mathit{ref}(\bot \mathit{int}))\backslash\emptyset$

Soundness

Soundness = "Nothing can go wrong during evaluation"

$\Longrightarrow$ Need to define evaluation rules

Evaluation rules

$$
\begin{array}{rcll}
\langle s, R[(l_2\ v)|_{l_1}]\rangle & \rightarrow & \langle s, R[l_2\ v]\rangle & l_2 \sqsubseteq l_1 \\
\langle s, R[l_1\ (l_2\ v)]\rangle & \rightarrow & \langle s, R[l_1\ v]\rangle & l_2 \sqsubseteq l_1 \\
\langle s, R[\mathbf{if}\ (l\ n)\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{fi}]\rangle & \rightarrow & \langle s, R[e_2]\rangle \quad n \neq 0 & \\
\langle s, R[\mathbf{if}\ (l\ 0)\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{fi}]\rangle & \rightarrow & \langle s, R[e_3]\rangle & \\
\langle s, R[(l\ \lambda x.e_1)\ v]\rangle & \rightarrow & \langle s, R[e_1[x \mapsto v]]\rangle & \\
\langle s, R[\mathbf{let}\ x = v\ \mathbf{in}\ e_2\ \mathbf{ni}]\rangle & \rightarrow & \langle s, R[e_2[x \mapsto v]]\rangle & \\
\langle s, R[l\ \mathbf{ref}\ v]\rangle & \rightarrow & \langle s[a \mapsto v], R[l\ a]\rangle & a \text{ fresh} \\
\langle s, R[!(l\ a)]\rangle & \rightarrow & \langle s, R[s(a)]\rangle & a \in dom(s) \\
\langle s, R[(l\ a) := v]\rangle & \rightarrow & \langle s[a \mapsto v], R[\perp\ ()]\rangle & a \in dom(s)
\end{array}
$$

Figure 5: Operational Semantics

Soundness

TAPL: "Soundness = Progress + Preservation"

Question: Where are these proofs in the paper?

Soundness

TAPL: "Soundness = Progress + Preservation"

Question: Where are these proofs in the paper?

Answer:

- Progress:

  Next we observe that *stuck* expressions (expressions that are not values but for which no reduction applies [WF94]) do not typecheck, which is trivial to prove.

- Preservation:

  Theorem 1 (Subject Reduction) If $A \vdash \langle s, e\rangle : \rho; C$ and $\langle s, e\rangle \rightarrow \langle s', e'\rangle$, then there exists an A' such that $A'|_{dom(A)} = A$ and $A' \vdash \langle s', e'\rangle : \rho; C'$ where $C' \subseteq C$.

Helper lemmas for Subject Reduction

Lemma 1 If $A \vdash e : \rho; C$ and S is a substitution such that SC is satisfiable, then $SA \vdash e : S\rho; SC$.

- Question: What's the type of S?
- Question: Why is S not applied to e?
- Question: Don't we need another substitution lemma?

Helper lemmas for Subject Reduction

Lemma 1 If $A \vdash e : \rho; C$ and S is a substitution such that SC is satisfiable, then $SA \vdash e : S\rho; SC$.

- Question: What's the type of S?
  Answer: $\mathit{TVar} \to \mathit{Typ}$
- Question: Why is S not applied to e?
- Question: Don't we need another substitution lemma?

Helper lemmas for Subject Reduction

Lemma 1 If $A \vdash e : \rho; C$ and S is a substitution such that SC is satisfiable, then $SA \vdash e : S\rho; SC$.

- Question: What's the type of S?
  Answer: $\mathit{TVar} \rightarrow \mathit{Typ}$
- Question: Why is S not applied to e?
  Answer: Because e cannot contain type variables.
- Question: Don't we need another substitution lemma?

Helper lemmas for Subject Reduction

Lemma 1 If $A \vdash e : \rho; C$ and S is a substitution such that SC is satisfiable, then $SA \vdash e : S\rho; SC$.

- Question: What's the type of S?
  Answer: $TVar \rightarrow Typ$
- Question: Why is S not applied to e?
  Answer: Because e cannot contain type variables.
- Question: Don't we need another substitution lemma?
  Answer: Yes, we also need a substitution lemma on the term level, to prove that if we step from $(\lambda x.e_1)v$ to $e_1[x \rightarrow v]$, the type is preserved.

Soundness

Corollary 1 (Soundness) If $\emptyset \vdash e : \rho; C$, then either e is a value or e diverges.

Question: What does that mean? Isn't $(\lambda x.x)\,1$ a counterexample?

Benchmarks: Const Inference

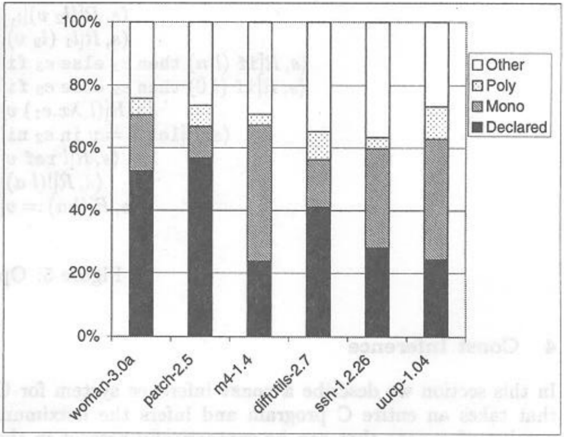


Figure 6: Number of inferred consts for benchmarks

Question: What's the meaning of Other/Poly/Mono/Declared?

Discussion

Question: What's the goal of qualifier inference?

Discussion

Question: What's the goal of qualifier inference?

- Rewrite C source automatically?
- More efficient program execution?
- Make C qualifier-polymorphic?

Thank you ☺