Splash Cascais 2023

香港大學 THE UNIVERSITY OF HONG KONG

Greedy Implicit Bounded Quantification

Chen Cui, Shengyi Jiang, Bruno C. d. S. Oliveira

October 26, 2023

The University of Hong Kong

Bounded Quantification

Mainstream OOP languages (Java, Scala, C#...) have polymorphic type systems with subtyping and **bounded quantification**.

```
public static <S extends Comparable> S min(S a, S b) {
    if (a.compareTo(b) <= 0) {
        return a;
    } else {
        return b;
    }
}
```

Bounded Quantification

Gives subtyping bounds to type variables.

The type of min: $\forall(\mathtt{S} \leq \mathtt{Comparable})\,.\,\mathtt{S} \to \mathtt{S} \to \mathtt{S}$.

However, there is little work on **type inference** algorithms supporting bounded quantification.

Type Inference

Type inference enables removing redundant type annotations.

```
List<String> numbers = Arrays.asList("1", "2", "3", "4", "5");
List<Integer> even = numbers.stream()
                            .map(s -> Integer.valueOf(s))
                            .filter(number -> number % 2 == 0)
                            .collect(Collectors.toList());
```

The type of `map` in Java is:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

- Type argument inference: `map` is instantiated with type `R = Integer`
- Argument inference: `s` has type `String`

Research on OOP Type Inference

Surprisingly little work devoted to practical OOP type inference:

- Most production compilers (Java/C#, etc) use algorithms loosely based on:
  - Benjamin C. Pierce, David N. Turner.
    Local type inference. TOPLAS 2000.
- Scala 2 is based on an improved form of Local type inference:
  - Martin Odersky, Christoph Zenger, Matthias Zenger.
    Colored local type inference. POPL 2001.

Local type inference suffers from some limitations. Next we will identify these limitations in Scala 2*, and compare it with $F^b_\leq$.

*The implementation of Scala 2 contains some improvements. Scala 3 has more improvements, but it has not been formally studied. Scala 2 type inference remains more faithful to the original work of local type inference.

No support for interdependent bounds

Scala 2 provides some basic support but it fails frequently.

```
def idFun[A, B <: A ⇒ A](x: B): (A ⇒ A) = x
def idInt1: (Int ⇒ Int) = (x ⇒ x)
```

✗ In Scala 2, function `idInt2` fails to type-check:

```
def idInt2 = idFun(idInt1)
```

- `A` is instantiated to $\bot$; `B` is instantiated to $\texttt{Int} \to \texttt{Int}$
- ✗ `B <: A ⇒ A` is not true: $\texttt{Int} \to \texttt{Int} \leq \bot \to \bot$ is not true.

No support for interdependent bounds

Scala 2 provides some basic support but it fails frequently.

```
def idFun[A, B <: A => A](x: B): (A => A) = x
def idInt1: (Int => Int) = (x => x)
```

✗ In Scala 2, function `idInt2` fails to type-check:

```
def idInt2 = idFun(idInt1)
```

- `A` is instantiated to $\bot$; `B` is instantiated to $\texttt{Int} \to \texttt{Int}$
- ✗ `B <: A => A` is not true: $\texttt{Int} \to \texttt{Int} \leq \bot \to \bot$ is not true.

✓ In $F^b_\leq$, interdependent bounds are supported:

$\texttt{let idFun}: \forall(a \leq \top).\ \forall(b \leq a \to a).\ b \to a \to a = \Lambda a.\ \Lambda b.\ \lambda x.\ x,$
$\texttt{idInt}: \texttt{Int} \to \texttt{Int} = \lambda x.\ x\ \texttt{in idFun idInt}$

Hard-to-synthesize arguments

```
def map[A, B](f: A => B, xs: List[A]): List[B] = ...
```

✗ In Scala 2, function `mapPlus1` fails to type-check:

```
def mapPlus1: List[Int] = map(x => 1 + x, List(1, 2, 3))
```

Local type inference requires the types of function arguments to be synthesized first, but we can never synthesize the type of `x => 1 + x`.

Hard-to-synthesize arguments

```
def map[A, B](f: A ⇒ B, xs: List[A]): List[B] = ...
```

✗ In Scala 2, function `mapPlus1` fails to type-check:

```
def mapPlus1: List[Int] = map(x ⇒ 1 + x, List(1, 2, 3))
```

Local type inference requires the types of function arguments to be synthesized first, but we can never synthesize the type of `x ⇒ 1 + x`.

✓ Workaround: Provide **type annotations** to the function argument.

```
def mapPlus2: List[Int] = map((x : Int) ⇒ 1 + x, List(1, 2, 3))
```

Hard-to-synthesize arguments

```
def map[A, B](f: A => B, xs: List[A]): List[B] = ...
```

✗ In Scala 2, function `mapPlus1` fails to type-check:

```
def mapPlus1: List[Int] = map(x => 1 + x, List(1, 2, 3))
```

Local type inference requires the types of function arguments to be synthesized first, but we can never synthesize the type of `x => 1 + x`.

✓ Workaround: Provide **type annotations** to the function argument.

```
def mapPlus2: List[Int] = map((x : Int) => 1 + x, List(1, 2, 3))
```

✓ $F^b_{\leq}$ can type-check the program without additional annotations:

`let` `map`: $\forall(a \leq \top).\ \forall(b \leq \top).\ (a \to b) \to [a] \to [b] = \ldots$
 `in` `map` $(\lambda x.\ x + 1)\ [1, 2, 3]$

No best argument

Sometimes invariant type variables cannot decide a **unique** instantiation.

```
def snd[A]: (Int ⇒ A ⇒ A) = (x ⇒ y ⇒ y)
def id = snd(1)
```

✗ In Scala 2, the type of id is inferred as $\bot \to \bot$. Thus id cannot be applied further.

✓ In $F^b_{\leq}$, unification is deferred. snd 1 can be applied further.
let snd: $\forall(a \leq \top).$ Int $\to (a \to a) \to a \to a = \Lambda a.\ \lambda x.\ \lambda y.\ y$ in snd 1

Higher-rank type inference

Take a polymorphic function as the argument of another function.

```
def k(f: Int => Int) = 1
def g(f: ([A <: Int] => A => A) => Int) = 1
```

✗ Scala 3[†], fails to type-check `def f = g(k)`:

- k has type $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$
- g accepts argument with type $(\forall(a \leq \texttt{Int}).\ a \to a) \to \texttt{Int}$
- ✗ $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \leq (\forall(a \leq \texttt{Int}).\ a \to a) \to \texttt{Int}$ is rejected
  - due to its lack of **implicit polymorphism**.

✓ $F^b_{\leq}$ has better support for higher-rank polymorphism:

$\texttt{let}$ k: $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} = \lambda f.\ 1,$
g: $((\forall(a \leq \texttt{Int}).\ a \to a) \to \texttt{Int}) \to \texttt{Int} = \lambda f.\ 1$ $\texttt{in g k}$

[†] Scala 2 does not support higher-rank types

$F^b_\leq$ Calculus

$F^b_\leq$ extends $F^e_\leq$ calculus‡ with bounded quantification.

- a variant of kernel $F_\leq$
  - **Global** type inference (long-distance constraints)
  - **Implicit instantiation** for monotypes (type argument inference)
    - $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \leq (\forall(a \leq \mathsf{Int}).\ a \to a) \to \mathsf{Int}$
  - **Explicit type application** for polytypes (**impredicative polymorphism**)
    - $(\Lambda a.\ \lambda x.\ x : \forall(a \leq \top).\ a \to a)\ @(\forall(b \leq \top).\ b \to b)$

Philosophy

- infer easy instantiations
- use explicit annotations for hard instantiations

‡ Jinxu Zhao, Bruno C. d. S. Oliveira. **Elementary Type Inference.** ECOOP 2022.

Syntax

$$
\begin{array}{llll}
\text{Type variables} & a, b & & \\
\text{Types} & A, B, C & ::= & 1 \mid a \mid \forall(a \leq B).\ A \\
 & & & \mid A \to B \mid \top \mid \bot \\
\text{Expressions} & e, t & ::= & x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A) \\
 & & & \mid e\ @A \mid \Lambda(a \leq B).\ e : A \\
\text{Typing contexts} & \Delta & ::= & \cdot \mid \Delta, x : A \mid \Delta, a \leq A \\
\text{Subtyping contexts} & \Psi & ::= & \Delta \mid \Psi, a \lesssim A
\end{array}
$$

Compared with $F^e_{\leq}$, $F^b_{\leq}$ now incorporates bounds to support bounded quantification.

Declarative Subtyping Rules

$\boxed{\Psi \vdash A \leq B}$ A is a subtype of B

$$\frac{}{\Psi \vdash 1 \leq 1}\ \leq\texttt{Unit} \qquad \frac{}{\Psi \vdash A \leq \top}\ \leq\top \qquad \frac{}{\Psi \vdash \bot \leq A}\ \leq\bot$$

$$\frac{a \mathrel{\underline{\lesssim}} B \in \Psi}{\Psi \vdash a \leq a}\ \leq\texttt{Var} \qquad \frac{a \mathrel{\underline{\lesssim}} B \in \Psi \quad \Psi \vdash B \leq A}{\Psi \vdash a \leq A}\ \leq\texttt{VarTrans}$$

$$\frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to$$

$$\frac{\Psi \vdash^m \tau \quad \Psi \vdash \tau \leq B \quad \Psi \vdash [\tau/a]A \leq C \quad C \text{ is not a } \forall \text{ type}}{\Psi \vdash \forall (a \leq B).\ A \leq C}\ \leq\forall\text{L}$$

$$\frac{\Psi \vdash B_1 \leq B_2 \quad \Psi \vdash B_2 \leq B_1 \quad \Psi, a \lesssim B_2 \vdash A_1 \leq A_2}{\Psi \vdash \forall (a \leq B_1).\ A_1 \leq \forall (a \leq B_2).\ A_2}\ \leq\forall$$

Non-syntactic Monotype

With bounded quantification, if we treat all type variables as monotypes, transitivity breaks due to rule $\leq$`VarTrans`.

- ✓ $\Psi \vdash A \leq B$:

$$\dfrac{b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq \top \quad b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq b}{b \leq \forall(c \leq 1).\ c \to 1 \vdash \forall(a \leq \top).\ a \leq b}\ \text{BY} \leq\forall\text{L}$$

- ✓ $\Psi \vdash B \leq C$:

  $b \leq \forall(c <: 1).\ c \to 1 \vdash b \leq \forall(c <: 1).\ c \to 1$ BY $\leq$`VarTrans`

- ✗ $\Psi \vdash A \not\leq C$: bounds are not equivalent

  $\forall(a \leq \top).\ a \not\leq \forall(c <: 1).\ c \to 1$

Non-syntactic Monotype

With bounded quantification, if we treat all type variables as monotypes, transitivity breaks due to rule $\leq$`VarTrans`.

✓ $\Psi \vdash A \leq B$:

$$\frac{b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq \top \quad b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq b}{b \leq \forall(c \leq 1).\ c \to 1 \vdash \forall(a \leq \top).\ a \leq b} \text{ BY } \leq\forall\text{L}$$

✓ $\Psi \vdash B \leq C$:

$b \leq \forall(c <: 1).\ c \to 1 \vdash b \leq \forall(c <: 1).\ c \to 1$ BY $\leq$`VarTrans`

✗ $\Psi \vdash A \not\leq C$: bounds are not equivalent

$\forall(a \leq \top).\ a \not\leq \forall(c <: 1).\ c \to 1$

In $F^b_{\leq}$, only type variables with bound $\top$ or monotype bounds are regarded as monotypes:

$$\frac{a \leq \top \in \Psi}{\Psi \vdash^m a} \text{ MTVar} \qquad \frac{a \leq A \in \Psi \quad \Psi \vdash^m A}{\Psi \vdash^m a} \text{ MTVarRec}$$

Two Variants of $F^b_{\leq}$: Complete Algorithm or Not?

In existing predicative HRP approaches, finding implicit instantiations is **greedy**. They rely on the property:

$$\tau_1 \leq \tau_2 \Longrightarrow \tau_1 = \tau_2$$

Variant 1: <u>sound</u>, <u>complete</u> and <u>decidable</u>

Type variables with <u>bound $\top$</u> are monotypes

Variant 2: <u>sound</u> but <u>incomplete</u>; type-checks more programs

Type variables with <u>bound $\top$</u> or <u>monotype bounds</u> are monotypes

It breaks the property due to rule $\leq$VarTrans:

$a \leq \mathsf{Int} \vdash a \leq \mathsf{Int}$ but $a \neq \mathsf{Int}$

Contributions

- A declarative bidirectional type system
  - Predicative implicit bounded quantification
  - Impredicative explicit type applications
  - Checking subsumption, type safety and completeness w.r.t. kernel $F_{\leq}$
- A *sound*, *complete* and *decidable* algorithm of variant 1
  - Worklist formulation[§]
- A *sound* algorithm of variant 2 with monotype subtyping
- Mechanical formalization and implementation
  - All theorems are verified in Abella (LOC: 24,919)
  - Haskell implementation

[§] Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. **A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference**. ICFP 2019.

Q&A

Implementation, proofs, and the extended version of the paper are available at: https://doi.org/10.5281/zenodo.8202095