Outline

- Motivation
Outline

- Motivation
- Definitions
- Examples
Outline

- Motivation
- Definitions
- Examples
- Case study: A more efficient implementation of imperative objects
Outline

- Motivation
- Definitions
- Examples
- Case study: A more efficient implementation of imperative objects
- Type safety
Outline

- Motivation
- Definitions
- Examples
- Case study: A more efficient implementation of imperative objects
- Type safety
- Bounded Existential Type
Motivation

Arises when subtyping is combined with polymorphism. Consider:

\[ f = \lambda x : \{ a : \text{Nat} \}. x \]

\[ \uparrow f : \{ a : \text{Nat} \} \rightarrow \{ a : \text{Nat} \} \]
Motivation

Arises when subtyping is combined with polymorphism. Consider:

\[ f = \lambda x : \{a : \text{Nat}\}.x \]

\[ f : \{a : \text{Nat}\} \rightarrow \{a : \text{Nat}\} \]

Now, what is the type of?

\[ f \{a = 0\} \]

\[ \{a = 0\} : \{a : \text{Nat}\} \]

\[ f \{a = 1, b = 4\} \]

\[ \{a = 1, b = 4\} : \{a : \text{Nat}\} \]
Motivation – Problem

Note that below is ill-typed!

\[
\text{let } c = f \begin{cases} a = 1, b = 4 \end{cases} \text{ in } c.b
\]
Motivation – Problem

Note that below is ill-typed!

\[ \text{let } c = f \{ a = 1, b = 4 \} \text{ in } c.b \]

One solution is to use universal type:

\[ f = \lambda X.\lambda x : X.x \]

\[ \Rightarrow f : \forall X.X \to X \]
Motivation – Problem

Note that below is ill-typed!

\[
\text{let } c = f \{a = 1, b = 4\} \text{ in } c.b
\]

One solution is to use universal type:

\[
f = \lambda X.\lambda x : X.x
\]

\[
\quad \Rightarrow f : \forall X.X \rightarrow X
\]

But how to handle:

\[
f = \lambda x : \{a : \text{Nat}\}.\{x.a, x\}
\]
Motivation – Problem

Note that below is ill-typed!

\[
\text{let } c = f \{a = 1, b = 4\} \text{ in } c.b
\]

One solution is to use universal type:

\[
f = \lambda X.\lambda x : X.x
\]

\[
\triangleright f : \forall X.X \rightarrow X
\]

But how to handle:

\[
f = \lambda x : \{a : \text{Nat}\}.\{x.a, x\}
\]

Certainly not working!

\[
f = \lambda X.\lambda x : X.\{x.a, x\}
\]
Motivation – Solution

Quantified type may be bounded by a subtyping relation:
Motivation – Solution

Quantified type may be bounded by a subtyping relation:

\[ f = \lambda X <: \{a : \text{Nat}\}.\lambda x : X.\{x.a, x\} \]

\[ f : \forall X <: \{a : \text{Nat}\}.X \rightarrow \{\text{Nat}, X\} \]
Definitions – Syntax

\[ t ::= ... \quad \nu ::= ... \]

\[ \lambda X <: T.t \quad \lambda X <: T.t \]

\[ T ::= ... \quad \Gamma ::= ... \]

\[ \forall X <: T.T \quad X <: T \]

Unbounded quantification is gone because
Definitions – Syntax

\[ t ::= \ldots \quad \nu ::= \ldots \]
\[ \lambda X <: T.t \quad \lambda X <: T.t \]

\[ T ::= \ldots \quad \Gamma ::= \ldots \]
\[ \forall X <: T.T \quad X <: T \]

Unbounded quantification is gone because

\[ \forall X . T \overset{\text{def}}{=} \forall X <: T_{op}.T \]
Definitions – Scoping

\[ \Gamma_1 = X <: Top, y : X \rightarrow \text{Nat} \]

well-scoped.
Definitions – Scoping

\[ \Gamma_1 = X <: Top, y : X \to \text{Nat} \]

well-scoped.

\[ \Gamma_2 = y : X \to \text{Nat}, X <: Top \]

bad
Definitions – Scoping

\[ \Gamma_1 = X <: \text{Top}, y : X \rightarrow \text{Nat} \] well-scoped.

\[ \Gamma_2 = y : X \rightarrow \text{Nat}, X <: \text{Top} \] bad

\[ \Gamma_3 = X <: \{ a : \text{Nat}, b : X \} \] F-bounded quantification. But in this chapter, we consider it bad.
Definitions – Scoping

\[ \Gamma_1 = X <: Top, y : X \rightarrow \text{Nat} \]

well-scoped.

\[ \Gamma_2 = y : X \rightarrow \text{Nat}, X <: Top \]

bad

\[ \Gamma_3 = X <: \{a : \text{Nat}, b : X\} \]

F-bounded quantification. But in this chapter, we consider it bad.

\[ \Gamma_4 = X <: \{a : \text{Nat}, b : Y\}, Y <: \{c : \text{Bool}, d : X\} \]

This is a generalization of \( \Gamma_3 \).
Definitions – Subtyping

Ternary relation $\Gamma \vdash S <: T$.

\[
\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad (S\text{-TVAR})
\]
Definitions – Subtyping

Ternary relation $\Gamma \vdash S <: T$.

\[
\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad (S\text{-TVAR})
\]

Kernel rule:

\[
\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1.S_2 <: \forall X <: U_1.T_2} \quad (S\text{-ALL})
\]

Full $F_<$ rule:

\[
\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: T_1.T_2} \quad (S\text{-ALL})
\]
Ternary relation $\Gamma \vdash t : T$.

\[
\Gamma, X <: T \vdash t_2 : T_2 \\
\Gamma \vdash \lambda X <: T.t_2 : \forall X <: T.T_2
\]

(T-TABS)

\[
\Gamma \vdash t_1 : \forall X <: T_{11}.T_{12} \quad \Gamma \vdash T_2 <: T_{11} \\
\Gamma \vdash t_1 [T_2] : [X \mapsto T_2]T_{12}
\]

(T-TAPP)
Examples – Products

\[ \text{Pair} \ T_1 \ T_2 = \forall X. (T_1 \to T_2 \to X) \to X; \]

\[ \text{pair} = \lambda X. \lambda Y. \lambda x : X. \lambda y : Y. \]

\[ (\lambda R. \lambda p : X \to Y \to R.p x y) \text{ as Pair} \ X \ Y \]

The encoding is done in system F. After porting it to \( F_{<:} \), we get the subtyping rule for pairs:

\[
\frac{\Gamma \vdash S_1 <: T_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \text{Pair} \ S_1 \ S_2 <: \text{Pair} \ T_1 \ T_2}
\]
Examples – Records

Flexible tuples:

\[ \{ T_i^{i \in 1..n} \} \overset{\text{def}}{=} \text{Pair } T_1 \ (\text{Pair } T_2 \ldots (\text{Pair } T_n \ \text{Top}) \ldots) \]

In particular, \( \{ \} \overset{\text{def}}{=} \text{Top} \).

Let \( L \) be a subset of the set of all the labels which have a global ordering.

\[ \hat{S}_i = \begin{cases} S_i, & \text{if } \text{label} - \text{with} - \text{index}(i) = l \in L \\ \text{Top}, & \text{if } \text{label} - \text{with} - \text{index}(i) \notin L \end{cases} \]

The record type \( \{ l : S_i^{l \in L} \} \) is defined as the flexible tuple \( \{ \hat{S}_i^{i \in 1..m} \} \).
Examples – Church numerals

System F’s encoding was

\[ \text{CNat} = \forall X. (X \to X) \to X \to X. \]
Examples – Church numerals

System F’s encoding was

$$\text{CNat} = \forall X. (X \to X) \to X \to X.$$ 

In $F_{<:}$, it can be encoded as

$$\text{SNat} = \forall X <: \text{Top}. \forall S <: X. \forall Z <: X.(X \to S) \to Z \to X.$$ 

So it distinguishes between zero and positive integers:
Examples – Church numerals

System F’s encoding was

\[ \text{CNat} = \forall X. (X \to X) \to X \to X. \]

In \( F_{<} \), it can be encoded as

\[ \text{SNat} = \forall X <: \text{Top}. \forall S <: X. \forall Z <: X. (X \to S) \to Z \to X. \]

So it distinguishes between zero and positive integers:

\[ \text{SZero} = \forall X <: \text{Top}. \forall S <: X. \forall Z <: X. (X \to S) \to Z \to Z \]
\[ \text{SPos} = \forall X <: \text{Top}. \forall S <: X. \forall Z <: X. (X \to S) \to Z \to S \]
Section 18.12 improved the efficiency: building the method table once per each object creation instead of once per each method invocation.

\[
\text{setCounterClass} = \\
\lambda r : \text{CounterRep}. \lambda self : \text{Source} \text{ SetCounter.} \\
\{ \text{get} = \lambda \text{Unit}.! (r.x), \text{set} = \lambda i : \text{Nat} . r.x := i, \\
\text{inc} = \lambda _ : \text{Unit}. (!self).\text{set} \\
\text{succ}((!self).\text{get unit}) \}\n\]

We would like to improve further: building method table only once for each class.
A Flawed Encoding

The order of parameters to the classes above is backwards. And it can be fixed by:

```
A Flawed Encoding

The order of parameters to the classes above is backwards. And it can be fixed by:

\[
\text{setCounterClass} = \\
\lambda \text{self} : \text{Source} \ (\text{CounterRep} \rightarrow \text{SetCounter}). \ \\
\lambda r : \text{CounterRep}. \{ \text{get} = \lambda _\_ : \text{Unit}.! (r.x), \\
\text{set} = \lambda i : \text{Nat}.r.x := i, \\
\text{inc} = \lambda _\_ : \text{Unit}.(!\text{self} r).\text{set} \\
(\text{succ}(!\text{self} r).\text{get unit}) \}
\]
A Flawed Encoding

The order of parameters to the classes above is backwards. And it can be fixed by:

\[
\text{setCounterClass} = \lambda \text{self} : \text{Source} \ (\text{CounterRep} \rightarrow \text{SetCounter}).
\]

\[
\lambda r : \text{CounterRep}.\{ \text{get} = \lambda_\_ : \text{Unit}.!(r.x),
\]

\[
\text{set} = \lambda i : \text{Nat}.r.x := i,
\]

\[
\text{inc} = \lambda_\_ : \text{Unit}.(!\text{self} r).\text{set}
\]

\[
(\text{succ}(!\text{self} r).\text{get unit}))
\]

Problem: the state type CounterRep is a contravariant.
Correct Encoding

$$
\text{setCounterClass} = \\
\lambda R <: \text{Counter Rep}. \lambda \text{self f : Source} \ (R \rightarrow \text{SetCounter}). \\
\lambda r : R. \{\text{get} = \lambda . \text{Unit}.! (r.x), \\
\text{set} = \lambda i : \text{Nat}. r.x := i, \\
\text{inc} = \lambda . \text{Unit}.(! \text{self } r).\text{set} \\
(\text{succ}((! \text{self } r).\text{get } \text{unit})))
$$
Correct Encoding (Cont.)

\[ instrCounterClass = \]
\[
\lambda R <: IntrCounterRep. \\
\lambda self : Source \ (R \rightarrow IntrCounter). \\
\lambda r : R.let super = setCounterClass [R] self in \]
\[
\ldots
\]

This is corrected because
Correct Encoding (Cont.)

\[
\text{instrCounterClass} = \lambda R <: \text{IntrCounterRep.} \\
\lambda \text{self} : \text{Source} (R \to \text{InstrCounter}). \\
\lambda r : R. \text{let super} = \text{setCounterClass} [R] \text{ self in} \\
... \\
\text{This is corrected because} \\
\text{Source}(R \to \text{InstrCounter}) <: \text{Source}(R \to \text{SetCounter})
\]
Type Safety

...
counterADT = \{ \texttt{Nat}, \{ \texttt{new} = 1, \texttt{get} = \lambda i : \texttt{Nat}. i, \\
inc = \lambda i : \texttt{Nat}. \texttt{succ}(i) \} \} \text{ as } \{ \exists \texttt{Counter} <: \texttt{Nat}, \\
\{ \texttt{new} : \texttt{Counter}, \texttt{get} : \texttt{Counter} \rightarrow \texttt{Nat}, \\
inc : \texttt{Counter} \rightarrow \texttt{Counter} \} \};

Now, we are allowed to use Counter values directly as numbers:

let \{ \texttt{Counter}, \texttt{counter} \} = \text{counterADT} \text{ in } \\
succ(succ(succ(\texttt{counter.inc \texttt{counter.new} })));

\texttt{\boxed{4 : Nat}}
Bounded Existential Type – Object

It reveals the names and types of some, but not all, of an object’s fields.

c = \{ \ast \{ x : \text{Nat}, \text{private} : \text{Bool} \},

\{ \text{state} = \{ x = 5, \text{private} = \text{False} \},

\text{methods} = \{ \text{get} = \lambda s : \{ x : \text{Nat} \}. s.x,

\text{inc} = \lambda s : \{ x : \text{Nat}, \text{private} : \text{Bool} \}.

\{ x = \text{succ}(s.x), \text{private} = s.\text{private} \} \} \}

\text{as} \ \{ \exists X <: \{ x : \text{Nat} \}, \{ \text{state} : X, \text{methods} :

\{ \text{get} : X \rightarrow \text{Nat}, \text{inc} : X \rightarrow X \} \} \};