Motivating Example

- Is \((\lambda f : \text{Int} \to \text{Int}. f \, 3)(\lambda x : \text{Int}. x + 5)\) well-typed in \(F_1\)?

\[
\begin{array}{c}
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} & \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. e) : \tau_1 \to \tau_2} & \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau} \\
\end{array}
\]

\[
\begin{array}{c}
\frac{\Gamma \vdash n : \text{Int}}{\Gamma \vdash n : \text{Int}} & \frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \\
\end{array}
\]
Motivating Example

• Is \((\lambda f : \text{Int} \rightarrow \text{Int}. \ f \ 3)(\lambda x : \text{Int}. \ x + 5)\) well-typed in \(F_1\)?

• We did not need to look at the type labels to determine well-typedness
  – We could have figured the actual types without the labels
  – We could have written the derivation without the labels

\[
\begin{array}{c}
   \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
   \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. \ e) : \tau_1 \rightarrow \tau_2} \\
   \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau} \\
   \frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}
\end{array}
\]
• Consider the simply-typed \( \lambda \)-calculus with integers

\[
e ::= x \mid \lambda x : \tau. \ e \mid e_1 \ e_2 \mid n \mid e_1 + e_2
\]

\[
\tau ::= \text{Int} \mid \tau_1 \rightarrow \tau_2
\]

Type inference

• Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?
• **Problem:** Consider the typing rule for function abstraction

\[
\Gamma, x : \tau \vdash e : \tau' \\
\Gamma \vdash (\lambda x : \tau. e) : \tau \rightarrow \tau'
\]

• Without type annotations, where do we get \(\tau\)?
• Use type variables to stand for as-yet-unknown types

\[
\tau ::= \alpha \mid \text{Int} \mid \tau \rightarrow \tau
\]

• Generate equality constraints \(\tau = \tau\) among the types and type variables
• Solve the constraints to compute a typing
Type Inference Rules

\[ \Gamma \vdash n : \text{Int} \]

\[ \Gamma, x : \alpha \vdash e : \tau' \quad \alpha \text{ fresh} \]
\[ \Gamma \vdash \lambda x.e : \alpha \rightarrow \tau' \]

\[ x : \tau \in \Gamma \]
\[ \Gamma \vdash x : \tau \]

\[ \Gamma \vdash e_1 : \tau_1 \]
\[ \Gamma \vdash e_2 : \tau_2 \]
\[ \tau_1 = \tau_2 \rightarrow \alpha \quad \alpha \text{ fresh} \]
\[ \Gamma \vdash e_1 \ e_2 : \alpha \]

Generated Constraint

\[ \tau_1 = \text{Int} \land \tau_2 = \text{Int} \]

\[ \Gamma \vdash e_1 + e_2 : \text{Int} \]
Example

\[
\Gamma, x : \alpha_2 \vdash x : \alpha_2 \quad \Gamma, x : \alpha_2 \vdash 1 : \text{Int} \quad \alpha_2 = \text{Int}
\]

\[
\Gamma, x : \alpha_2 \vdash x + 1 : \text{Int} \\
\Gamma \vdash (\lambda x . x + 1) : \alpha_2 \rightarrow \text{Int} \\
\Gamma \vdash 2 : \text{Int} \quad \alpha_2 \rightarrow \text{Int} = \text{Int} \rightarrow \alpha_1
\]

\[
\Gamma \vdash (\lambda x . x + 1) \ 2 : \alpha_1
\]

- We collect all constraints appearing in the derivation into some set \( C \) to be solved.
- Here, \( C \) contains \( \alpha_2 \rightarrow \text{Int} = \text{Int} \rightarrow \alpha_1 \) and \( \alpha_2 = \text{Int} \).
- Solution: \( \alpha_1 = \text{Int} = \alpha_2 \).
- Thus this program is typeable:
  we can derive a typing by replacing \( \alpha_1 \) and \( \alpha_2 \) by Int in the proof.
Solving Equality Constraints

We can solve the equality constraints using the following rewrite rules

- \( C \cup \{ \text{Int} = \text{Int} \} \Rightarrow C \)
- \( C \cup \{ \alpha = \tau \} \Rightarrow C[\alpha \mapsto \tau] \) provided \( \alpha \not\in \text{FV}(\tau) \)
- \( C \cup \{ \tau = \alpha \} \Rightarrow C[\alpha \mapsto \tau] \) provided \( \alpha \not\in \text{FV}(\tau) \)
- \( C \cup \{ \alpha_1 \rightarrow \alpha_2 = \alpha'_1 \rightarrow \alpha'_2 \} \Rightarrow C \cup \{ \alpha_1 = \alpha'_1 \} \cup \{ \alpha_2 = \alpha'_2 \} \)
- \( C \cup \{ \text{Int} = \alpha_1 \rightarrow \alpha_2 \} \Rightarrow \text{unsatisfiable} \)
- \( C \cup \{ \alpha_1 \rightarrow \alpha_2 = \text{Int} \} \Rightarrow \text{unsatisfiable} \)

The condition \( \alpha \not\in \text{FV}(\tau) \) prevents inferring recursive types.
Termination

• We can prove that the constraint solving algorithm terminates
• For each rewriting rule, we either
  – Reduce the size of the constraint set
  – Reduce the number of “arrow” constructors in the constraint set
• As a result, the constraint always gets “smaller” and eventually becomes empty
• A similar argument is made for strong normalization in the simply-typed lambda calculus
Exercise

• Is \((\lambda f. f \ 3)(\lambda x. x + 5)\) well-typed in \(F_1\)?

\[
\begin{array}{c}
\Gamma \vdash n : \text{Int} \\
\Gamma, x : \alpha \vdash e : \tau' \quad \alpha \text{ fresh} \\
\Gamma \vdash \lambda x. e : \alpha \rightarrow \tau' \\
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\tau_1 = \text{Int} \land \tau_2 = \text{Int} \\
\Gamma \vdash e_1 + e_2 : \text{Int}
\end{array}
\]
Polymorphism
Motivating Example

• A type system restricts the class of programs that are considered “legal”

• An expression in the untyped $\lambda$-calculus may be reducible to a value but may not be typeable in a particular type system

\[
\text{let } id = \lambda x.x \text{ in }
\]

\[
\left( \cdots (id \text{ true}) \cdots (id \text{ 1}) \cdots \right)
\]

• This expression is not typeable in the simple type system we have discussed so far
Observation: \( \lambda x.x \) returns its argument exactly and places no constraints on the type of \( x \)

The identity function works for any argument type

We can express this with universal quantification:

\[
\lambda x.x : \forall \alpha. \, \alpha \rightarrow \alpha
\]

For any type \( \alpha \), the identity function has type \( \alpha \rightarrow \alpha \)

This is also known as parametric polymorphism
Polymorphism

You have seen this before

```java
public interface List<E>{
    void add(E x);
    E get();
}
...

List<String> ls = ...
ls.add("Hello");
String hello = ls.get(0);
```

How do we formalize this concept?
Let’s extend our system as follows:

\[
\tau ::= \alpha \mid \text{Int} \mid \tau \to \tau \mid \forall \alpha.\tau
\]

\[
e ::= n \mid x \mid \lambda x : \tau . e \mid e\ e \mid \Lambda \alpha . e \mid e[\tau]
\]

- We add polymorphic types, add explicit type abstraction (generalization)
- Annotated code locations at which a value of polymorphic type is created
- We add type application (instantiation)
• Polymorphic functions map types to terms
• Normal functions map terms to terms
• Examples:
  • \( \Lambda \alpha. \lambda x : \alpha.x : \forall \alpha.\alpha \rightarrow \alpha \)
  • \( \Lambda \alpha. \Lambda \beta. \lambda x : \alpha.\lambda y : \beta.x : \forall \alpha.\forall \beta.\alpha \rightarrow \beta \rightarrow \alpha \)
  • \( \Lambda \alpha. \Lambda \beta. \lambda x : \alpha.\lambda y : \beta.y : \forall \alpha.\forall \beta.\alpha \rightarrow \beta \rightarrow \beta \)
• When we use a parametric polymorphic type, we apply (or instantiate) it with a particular type
• In System F this is done by hand:
  • \((\Lambda \alpha. \lambda x : \alpha . x)[\tau_1] : \tau_1 \to \tau_1\)
  • \((\Lambda \alpha. \lambda x : \alpha . x)[\tau_2] : \tau_2 \to \tau_2\)
• This is where the term parametric comes from
• The type \(\forall \alpha . \alpha \to \alpha\) is a “function” in the domain of types, and it is passed a parameter at instantiation time
Type Abstraction (Generalization)

\[
\begin{align*}
\Gamma, \alpha \vdash e &: \tau \\
\Gamma \vdash \Lambda \alpha . e &: \forall \alpha . \tau
\end{align*}
\]

Type Application (Instantiation)

\[
\begin{align*}
\Gamma \vdash e &: \forall \alpha . \tau \\
\Gamma \vdash e[\tau'] &: \tau[\alpha \mapsto \tau']
\end{align*}
\]
Free Variables of a Type

- Need to perform substitutions on quantified types
- Just like λ-calculus, we need to worry about free variables and capture-free substitution
- Define the free variables of a type
  - $FV(\alpha) = \{\alpha\}$
  - $FV(c) = \emptyset$
  - $FV(\tau \rightarrow \tau') = FV(\tau) \cup FV(\tau')$
  - $FV(\forall \alpha.\tau) = FV(\tau) - \{\alpha\}$
Substitution

Define $\tau[\alpha \rightarrow u]$ as

- $\alpha[\alpha \mapsto u] = u$
- $\beta[\alpha \mapsto u] = \beta$ \hspace{1cm} \text{where } \beta \neq \alpha$
- $(\tau \rightarrow \tau')[\alpha \mapsto u] = \tau[\alpha \mapsto u] \rightarrow \tau'[\alpha \mapsto u]$
- $(\forall \beta. \tau)[\alpha \mapsto u] = \forall \beta.(\tau[\alpha \mapsto u])$ \hspace{1cm} \text{where } \beta \neq \alpha \text{ and } \beta \notin \text{FV}(u)$

Define $e[\alpha \mapsto u]$ as

- $(\lambda x : \tau.e)[\alpha \mapsto u] = \lambda x : \tau[\alpha \mapsto u].e[\alpha \mapsto u]$
- $(\Lambda\beta.e)[\alpha \mapsto u] = \Lambda\beta.e[\alpha \mapsto u]$ \hspace{1cm} \text{where } \beta \neq \alpha \text{ and } \beta \notin \text{FV}(u)$
- $(e_1 \ e_2)[\alpha \mapsto u] = e_1[\alpha \mapsto u] \ e_2[\alpha \mapsto u]$
- $x[\alpha \mapsto u] = x$
- $n[\alpha \mapsto u] = n$
Inference for Polymorphism

- We would like to have the power of System F, and the ease of use of type inference
- Given an untyped $\lambda$-calculus expression, can we discover the annotations necessary for typing the term in System F? (if such a typing is possible)
- No: This problem has been shown to be undecidable
- Can we at least perform some type inference for parametric polymorphism?
- Yes! (Hindley and Milner Type System)