The Untyped Lambda Calculus

Programming Language Theory

Ralf Lämmel
What’s the lambda calculus?

- It is the core of functional languages.
- ...

\[ x \]
\[ \lambda x. x \]
\[ \lambda f. \lambda g. \lambda x. f(g(x)) \]

The identity function

Function composition
What’s the lambda calculus?

- **It is the core of functional languages.**
- It is a mathematical system for studying programming languages.
  - design, specification, implementation, type systems, et al.
- It comes in variations of typing: implicit/explicit/none.
- Formal systems built on top of simply typed lambda calculus:
  - System F — for studying polymorphism
  - System F <: — for studying subtyping
  - ...

© Ralf Lämmel, 2009-2012 unless noted otherwise
Language constructs

• Abstract syntax:

\[ M ::= x \mid M \ M \mid \lambda x. M \]

• \( M \) is a lambda term.

• An infinite set of variables \( x, y, z, \ldots \) is assumed.

• \( M \ N \) is an application.

  Function \( M \) is applied to the argument \( N \).

• \( \lambda x. M \) is an abstraction.

  The resulting function maps \( x \) to \( M \).

Lambda functions are anonymous.
• Church’s thesis:

All intuitively computable functions are $\lambda$-definable.

• An established equivalence of notions of computability:

  Set of Lambda-definable functions

  $=$  Set of Turing-computable functions
Syntax and semantics

• Syntax

  $t ::= x$
  $\lambda x.t$
  $tt$

  $v ::= \lambda x.t$

• Evaluation

  $t_1 \rightarrow t_1'$
  $t_1 t_2 \rightarrow t_1' t_2$

  $t \rightarrow t'$
  $v t \rightarrow v t'$

  $(\lambda x.t) v \rightarrow [v/x]t$

Terms

A term that cannot be reduced further.

Values (normal forms)

Reduce function position, then reduce argument position, then apply.
Syntactic sugar and conventions

- $M N_1 \ldots N_k$ means $(((M N_1) N_2) \ldots N_k)$.
  Function application groups from left to right.

- $\lambda x. y$ means $(\lambda x. (x y))$.
  Function application has higher precedence.

- $\lambda x_1 x_2 \ldots x_k. M$ means $\lambda x_1. (\lambda x_2. (\ldots (\lambda x_k. (M)) \ldots)))$. 
Variable binding

- \( \lambda \) is a binding operator:
  It binds a variable in the scope of the lambda abstraction.

- Examples:
  - \( \lambda x.M \) \( x \) is bound (in the lambda abstraction)
  - \( \lambda x.x \ y \) \( y \) is not bound (in the lambda abstraction).

- If a variable occurs in an expression without being bound,
  then it is called a **free** occurrence, or a free variable. Other
  occurrences of variables are called **bound**.

- A **closed term** is one without free variable occurrences.
Variable binding — precise definition

\[ \text{FV}(M) \text{ defines the set of free variables in the term } M \]

\[ \text{FV}(x) = \{x\} \]

\[ \text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N) \]

\[ \text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\} \]
Exercise: what are the free and bound variable occurrences in these terms?

\[(\lambda x. y)(\lambda y. y)\]

\[\lambda x. (\lambda y. x \ y) y\]
Substitution and $\beta$-equivalence

• Computation for the $\lambda$-calculus is based on substitution.

• Substitution is defined by the equational axiom:

$$(\lambda x.M)N = [N/x]M$$

• Think of substitution as invoking a function:

  ★ $(\lambda x.M)$ is the function,
  ★ $N$ is the argument,
  ★ Substitution takes care of parameter passing.

The terms on both sides are also called $\beta$-equivalent.

“$\rightarrow$” direction = $\beta$-reduction
\( \alpha \)-equivalence and conversion

- Names of bound variable are insignificant.

\( \lambda x.x \) defines the same function as \( \lambda y.y \)

- Suppose two terms differ only on the names of bound variables.

Then, they are said to be \( \alpha \)-equivalent (\( =\alpha \)).

- Equational axiom:

\[ \lambda x.M = \lambda y.[y/x]M \]

where \( y \) does not appear in \( M \)

and substitution applies to free occurrences only.

Performing such renaming is also called \( \alpha \)-conversion.
Reduction $M \rightarrow N$

- Computation ($\rightarrow$) with the lambda calculus is then a series of
  - $\beta$-reductions, and
  - ("implicit") $\alpha$-conversions.

- Reflexive, transitive closure

- $M \rightarrow^* N$ means $M$ reduces to $N$ in zero or more steps.
Inductive definition of substitution

\[
\begin{align*}
[N/x]x & = N \\
[N/x]y & = y, \quad y \text{ any variable different from } x \\
[N/x](M_1 M_2) & = ([N/x]M_1) ([N/x]M_2) \\
[N/x](\lambda x. M) & = \lambda x. M \\
[N/x](\lambda y. M) & = \lambda y.([N/x]M), \ y \text{ not free in } N
\end{align*}
\]

Examples

\[
\begin{align*}
[z/x]x & \rightarrow z \\
[z/x](\lambda x.x x) & \rightarrow \lambda x.x x \\
[z/x](\lambda y.y x) & \rightarrow \lambda y.y z \\
[z/x](\lambda z.x z) & \rightarrow \lambda a.z a
\end{align*}
\]
Properties of reduction (i.e., semantics)

• How do we select redexes for reduction steps?
• Does the result depend on such a choice?
• Does reduction ultimately terminate with a normal form?
Illustration of different reductions
(We assume natural numbers with “+”.)

Option 1

\[(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) \ 2\]

\[\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) x)) \ 2\]

\[\rightarrow (\lambda x. (\lambda y. y + 1) (x + 1)) \ 2\]

\[\rightarrow (\lambda x. (x + 1 + 1)) \ 2\]

\[\rightarrow (2 + 1 + 1)\]

\[\rightarrow 4\]

Option 2

\[(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) \ 2\]

\[\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) x)) \ 2\]

\[\rightarrow (\lambda y. y + 1) ((\lambda y. y + 1) 2)\]

\[\rightarrow \ldots\]

\[\rightarrow \ldots\]

\[\rightarrow 4\]
Confluence

- Confluence: evaluation strategy is not significant for final value.
- That is: there is (at most) one normal form of a given expression.

Confluence

• $M \rightarrow^* N$ means $M$ reduces to $N$ in zero or more steps.

• Confluence

  If $M \rightarrow^* N$ and $M \rightarrow^* N'$,

  then there exists some $P$

  such that $N \rightarrow^* P$ and $N' \rightarrow^* P$. 
Strong normalization property of a calculus with reduction

• Definition:

For every term \( M \) there is a normal form \( N \) such that \( M \rightarrow^* N \).

• Strong normalization properties for lambda calculi:

- Untyped lambda calculus: \textit{no} \((\lambda x.x x)(\lambda x.x x)\)
- Simply-typed lambda calculus: \textit{yes}
Evaluation/reduction strategies

✦ **Full beta reduction**
Reduce anywhere.

✦ **Applicative order** (“reduce the leftmost innermost redex”)
Reduce argument before applying function.

✦ **Normal order** (“reduce the leftmost outermost redex”)
Apply function before reducing argument.

Choice of strategy may impact termination behavior

- **eager (strict)**
- **lazy (non-strict)**

Extension vs. encoding

• Typical extensions
  (giving rise to so-called applied lambda calculi)
  ✦ Primitive types (numbers, Booleans, ...)
  ✦ Type constructors (tuples, records, ...)
  ✦ Recursive functions
  ✦ Effects (cell, exceptions, ...)
  ✦ ...

• Many extensions can be encoded in theory in terms of pure lambda calculus, except that such encoding is somewhat tedious.
Church Booleans

• Encodings of literals

✦ \( \text{true} = \lambda t.\lambda f.t \)
✦ \( \text{false} = \lambda t.\lambda f.f \)

• Conditional expression (if)

✦ Expectations

★ \( \text{test } b \text{ v w } \rightarrow^* \text{ v, if } b = \text{true} \)
★ \( \text{test } b \text{ v w } \rightarrow^* \text{ w, if } b = \text{false} \)

✦ Encoding

★ \( \text{test} = \lambda l.\lambda m.\lambda n.l \ m \ n = \lambda l. (\lambda m. (\lambda n. ((l \ m) \ n))) \)
Example reduction

$$(\lambda l.\lambda m.\lambda n.l\ m\ n)\ true\ v\ w$$

$$\rightarrow (\lambda m.\lambda n.\true\ m\ n)\ v\ w$$

$$\rightarrow (\lambda n.\true\ v\ n)\ w$$

$$\rightarrow true\ v\ w$$

$$\rightarrow (\lambda t.\lambda f.t)\ v\ w$$

$$\rightarrow (\lambda f.v)w$$

$$\rightarrow v$$
Church pairs

• A Boolean value picks either the 1st or the 2nd value of the pair.

• Construction and projections

  ✦ $\text{pair} = \lambda f. \lambda s. \lambda b. b \ f \ s$

  ✦ $\text{first} = \lambda p. p \ \text{true}$

  ✦ $\text{second} = \lambda p. p \ \text{false}$
Church numerals

• Encodings of numbers
  ✦ $c0 = \lambda s.\lambda z. z$
  ✦ $c1 = \lambda s.\lambda z. s \, z$
  ✦ $c2 = \lambda s.\lambda z. s \, (s \, z)$
  ✦ $c3 = \lambda s.\lambda z. s \, (s \, (s \, z))$
  ✦ ...

• Encodings of functions on numbers
  ✦ $succ = \lambda n.\lambda s.\lambda z. s \, (n \, s \, z)$
  ✦ $plus = \lambda m.\lambda n.\lambda s.\lambda z. m \, s \, (n \, s \, z)$
  ✦ $times = \lambda m.\lambda n. m \, (plus \, n) \, c0$
  ✦ ...

A numeral $n$ is a lambda abstraction that is parameterized by a case for zero, and a case for $\text{succ}$. In the body, the latter is applied $n$ times to the former. This caters for primitive recursion.
Recursive functions

• Let us define the factorial function.

• Suppose we had “recursive function definitions”.

\[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n*f(n - 1) \]

• Let us do such recursion with anonymous functions.

• Fixed point combinators to the rescue!
Fixed points

• Consider a function $f : X \rightarrow X$.
• A fixed point of $f$ is a value $z$ such that $z = f(z)$.
• A function may have none, one, or multiple fixed points.
• Examples (functions and sets of fixed points):
  ✦ $f(x) = 2x$  \quad \{0\}
  ✦ $f(x) = x$  \quad \{0, 1, \ldots\}
  ✦ $f(x) = x + 1$  \quad \emptyset
Fixed point combinators

We call $Y$ a fixed-point combinator if it satisfies the following **definitional property**:

For all $f : X \to X$ it holds that $Yf = f(Yf)$

100(M)$ Question: does such a $Y$ exist?
Defining factorial as a fixed point

- Start from a recursive definition.
  \[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n * f(n - 1) \]

- Eliminate self-reference; receive function as argument.
  \[ g \equiv \lambda h. \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n * h(n - 1) \]
  ★ g takes a function (h) and returns a function.

- Define f as a fixed point.
  \[ f \equiv Y g \]
For example, apply definitional property to factorial g:

\[(Y \ g) \ 2\]

\[= [Y \ \text{def. prop.}] \quad \quad \quad \quad g \ (Y \ g) \ 2\]

\[= [\text{unfold } g] \quad \quad \quad \quad (\lambda \ h. \lambda n. \ \text{if } n == 0 \ \text{then} \ 1 \ \text{else} \ n* h(n - 1)) \ (Y \ g) \ 2\]

\[= [\beta \text{ reduce}] \quad \quad \quad \quad (\lambda \ n. \ \text{if } n == 0 \ \text{then} \ 1 \ \text{else} \ n* ((Y \ g)(n - 1))) \ 2\]

\[= [\text{"-" reduce}] \quad \quad \quad \quad \text{if } 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y \ g)(2 - 1))\]

\[= [\text{"if" reduce}] \quad \quad \quad \quad \text{if } 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y \ g)(1))\]

This is as if we had extended evaluation.

\[= 2\]

Apply these steps one more time.
A lambda term for $\mathbf{Y}$

- One option:
  \[ Y = \lambda f . (\lambda x.f (x x))(\lambda x.f (x x)) \]

- Verification of the definitional property:
  \[ Y g = g (Y g) \]

- Proof:
  \[
  Y g = \begin{array}{l}
  \hline
  \text{[unfold $Y$]} \quad (\lambda f . (\lambda x.f (x x))(\lambda x.f (x x))) g \\
  \text{[beta reduce]} \quad (\lambda x.g(x x))(\lambda x.g(x x)) \\
  \text{[beta reduce]} \quad g ((\lambda x.g(x x))(\lambda x.g(x x))) \\
  \text{[fold $Y$]} \quad g (Y g)
  \end{array}
  \]
CBV fixed point combinator

\[
\lambda(f, \lambda(x, \lambda(y, \lambda(x, \lambda(x, \lambda(y, \lambda(y)))))))
\]
• **Summary**: The untyped lambda calculus
  - A concise core of functional programming.
  - A foundation of computability.
  - A Prolog model is straightforward.

• **Prepping**: “Types and Programming Languages”
  - Chapter 5

• **Outlook**:
  - The simply-typed lambda calculus
  - Polymorphism
The Simply Typed Lambda Calculus

Ralf Lämmel

[Jarvi] Slides by J. Jarvi: “Programming Languages”, CPSC 604 @ TAMU (2009)
Towards typed lambda calculus

• Now suppose you want to distinguish values of different types:
  ✦ Booleans
  ✦ Numbers
  ✦ Functions on Booleans
  ✦ Functions on functions on Booleans
  ✦ Products, sums of ...
  ✦ ...

• These types need to be ...
  ✦ specified in the program, and
  ✦ checked to be correct.
Setting up the simply typed lambda calculus

• Define syntax for types including function types.
• Extend lambda abstractions for explicit types.
• Define typing rules.
• Revise reduction semantics.
• Establish type safety.
• Consider extensions.

There is also a type system for lambda terms without explicit types, which we do not discuss in this
Revised lambda abstraction

• Lambda abstractions are annotated with types:

\[ \lambda x : T.t \]

• Grammar of types:

\[ T ::= \text{bool} \]

\[ \text{natt} \]

\[ T \rightarrow T \]

We only consider these simple types here for simplicity.
Examples

• What are the types of these terms?
  ✦ \( \lambda x : \text{bool}.x \)
  ✦ \( \lambda f : \text{bool} \to \text{bool}.f \ x \)

• Here are the same terms for the untyped calculus:
  ✦ \( \lambda x . \ x \)
  ✦ \( \lambda f.f \ x \)

Note that lambda variables are typed explicitly.
Meaningless terms

• Some terms diverge.

• Some applications are ill-typed, e.g.:

\( (\lambda f : \text{bool} \rightarrow \text{bool}.f \ x) \text{ true} \)

• Goal: a type system to reject ill-typed terms.

... unless we end up showing strong normalization.
Typing relation with context

\[ \Gamma \vdash t : T \]

*Term* \( t \) *has type* \( T \) *in the typing context* \( \Gamma \)

- A typing context is a sequence of bindings.
- Each binding is a variable-type pair, e.g.: \( x : T \).
- Contexts are composed as in \( \Gamma, x : T \).
- All variable names are distinct for a given \( \Gamma \).
- \( \Gamma \) can be omitted if it is empty.
- \( \Gamma \) can be empty for closed terms.
Typing rules for simply-typed lambda calculus

- $x, y, z, f, g$ range over variables
- $s, t, u$ range over terms
- $S, T, U$ range over types

**T-Variable**

$x : T \in \Gamma$

\[ \frac{}{\Gamma \vdash x : T} \]

**T-Abstraction**

$\Gamma, x : T \vdash u : U$

\[ \frac{}{\Gamma \vdash \lambda x : T. u : T \to U} \]

**T-Application**

$\Gamma \vdash t : U \to T$

$\Gamma \vdash u : U$

\[ \frac{}{\Gamma \vdash t \ u : T} \]
Rules for bool

T-True  
\[ \vdash \text{true : bool} \]

T-False  
\[ \vdash \text{false : bool} \]

These typing rules illustrate one option to add specific types and their operations to a basic lambda calculus. Basically, we need to add one rule per operation.
Evaluation rules

- Syntax (terms, values, types)

  \[ t ::= x \mid v \mid t \ t \]
  \[ v ::= \lambda x : T \ t \mid \text{true} \mid \text{false} \]
  \[ T ::= \text{bool} \mid T \to T \]

- Evaluation rules

  \[
  \frac{t_1 \rightarrow t_1'}{t_1 \ t_2 \rightarrow t_1' \ t_2'} \quad \frac{t \rightarrow t'}{v \ t \rightarrow v \ t'}
  \]
  \[
  (\lambda x : T \ t) \ v \rightarrow [v/x]t
  \]

Evaluation rules do not bother with types.
Typing derivations

Construct derivations as proofs of terms having a certain type.

\[
\begin{align*}
\text{f: bool} & \rightarrow \text{bool} \in f: \text{bool} \rightarrow \text{bool} \\
\text{f: bool} & \rightarrow \text{bool} \vdash f: \text{bool} \rightarrow \text{bool} & \text{f: bool} & \rightarrow \text{bool} \vdash \text{false: bool} & \text{g: bool} & \in g: \text{bool} \\
\text{f: bool} & \rightarrow \text{bool} \vdash f \text{ false: bool} & \vdash (\lambda f: \text{bool} \rightarrow \text{bool}. f \text{ false}): (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} & \vdash \lambda g: \text{bool}. g: \text{bool} \rightarrow \text{bool} \\
\vdash (\lambda f: \text{bool} \rightarrow \text{bool}. f \text{ false}) \lambda g: \text{bool}. g: \text{bool}
\end{align*}
\]

T-Variable
\[
\Gamma, x: T \vdash u: U
\]

T-Abstraction
\[
\Gamma \vdash x: T \rightarrow U
\]

T-Application
\[
\Gamma \vdash t: U \rightarrow T \quad \Gamma \vdash u: U \\
\Gamma \vdash t u: T
\]
Type safety
= progress + preservation

**Progress:** If \( t \) is a closed, well-typed term, then either \( t \) is a value, or there exists some \( u \), such that \( t \rightarrow u \).

**Preservation:** If \( \Gamma \vdash t : T \) and \( t \rightarrow u \), then \( \Gamma \vdash u : T \)

Requires several trivial lemmas (properties) that are omitted here.
A few extensions

• Recursion (fixed point combinator)
• Type annotation (for documentation, abstraction)
• Pairs (as a simple form of type construction)
• Lists (another example of type construction)
• Records (as a first step towards objects)
• Mutable variables (not discussed in this lecture)
• ...
Recursion

• A fixed point combinator is definable in the untyped calculus.
• It is not definable in the simply typed version.
  ♦ A special combinator \textit{fix} is added to the formal system.
  ♦ Alternatively, a more powerful type system is needed.
Why is $Y$ not typeable?

• Consider again the fixed-point combinator:

$$Y = \lambda f . (\lambda x. f (x \ x)) (\lambda x. f (x \ x))$$

• Let us simplify.

$Y$ contains self-application:

$$\lambda x. \ x \ x$$

• How could self-application be typeable?

$$\lambda x : \ ? \ . \ x \ x$$

The type of $x$ would need to be obviously a function type and the argument type of the same type.
iseven : nat → bool
iseven = fix iseven’

iseven’ : (nat → bool) → nat → bool
iseven’ = λ f:nat → bool. λ x:nat. if iszero x then true
else if iszero (pred x) then false
else f (pred (pred x))
We think of unfolding the recursive definition.

Intuition

- $\text{fix even' } 0 \Rightarrow \text{true}$
- $\text{fix even' } 1 \Rightarrow \text{false}$
- $\text{fix even' } 2 \Rightarrow \text{true}$
- $\text{fix even' } 3 \Rightarrow \text{false}$
- $\text{fix even' } 4 \Rightarrow \text{true}$

- $\text{even' } ? 0 \Rightarrow \text{true}$
- $\text{even' } ? 1 \Rightarrow \text{false}$
- $\text{even' } ? 2 \Rightarrow ?$
- $\text{even' } (\text{even' } ?) 2 \Rightarrow \text{true}$
- $\text{even' } (\text{even' } ?) 3 \Rightarrow \text{false}$
- $\text{even' } (\text{even' } ?) 4 \Rightarrow ?$
Recursion in the presence of types

Add a primitive **fix**.

\[ t ::= \ldots \mid \text{fix } t \]

**Typing rule**

\[
\Gamma \vdash t : T \rightarrow T \\
\Gamma \vdash \text{fix } t : T
\]

**Evaluation rules**

\[
\frac{t \rightarrow t'}{\text{fix } t \rightarrow \text{fix } t'} \\
\text{fix } (\lambda x : T. t) \rightarrow [(\text{fix } (\lambda x : T. t))/x] t
\]
Type annotation (ascription)

• Syntax:

\[ t ::= \ldots | t \text{ as } T \]

• Typing rule:

\[
\begin{array}{c}
\Gamma \vdash t : T \\
\hline
\Gamma \vdash t \text{ as } T : T
\end{array}
\]

• Evaluation rules:

\[
\begin{array}{c}
t \rightarrow u \\
\hline
\Gamma \vdash t \text{ as } T \rightarrow u \text{ as } T
\end{array}
\]

\[
\begin{array}{c}
v \rightarrow v \\
\text{as } T \rightarrow v
\end{array}
\]
• **Summary:** The typed lambda calculus
  ✦ Typing relation carries argument for context.
  ✦ Recursion requires built-in Y combinator.

• **Prepping:** “Types and Programming Languages”
  ✦ Chapters 7 and 11

• **Outlook:**
  ✦ Lambda calculi with polymorphism
  ✦ Object calculi
  ✦ Process calculi
Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

\[ x = 1 \]

\[ \text{let } x = 1 \text{ in } \ldots \]

\[ x(1). \]

\[ !x(1) \]

\[ x.\text{set}(1) \]

Programming Language Theory

System F

Ralf Lämmel

[Järvi] Slides by J. Järvi: “Programming Languages”, CPSC 604 @ TAMU (2009)
Polymorphism -- Why?

• What's the identity function?

• In the simply typed lambda calculus, we need many!

• Examples
  ✦ \( \lambda x : \text{bool}. \ x \)
  ✦ \( \lambda x : \text{nat}. \ x \)
  ✦ \( \lambda x : \text{bool} \rightarrow \text{bool}. \ x \)
  ✦ \( \lambda x : \text{bool} \rightarrow \text{nat}. \ x \)
  ✦ ...
Kinds of polymorphism

- Parametric polymorphism ("all types")
- Bounded polymorphism ("subtypes")
- Ad-hoc polymorphism ("some types")
- Existential types ("exists as opposed to for all")
System F -- Syntax

\[ t ::= x | v | t \ t \mid t[T] \]
\[ v ::= \lambda x : T . t \mid \forall X . t \]
\[ T ::= X | T \to T \mid \forall X . T \]

Example:
\[ id : \forall X . X \to X \]
\[ id = \forall X . \lambda x : X . x \]

System F [Girard72, Reynolds74] =
(simply-typed) lambda calculus
+ type abstraction & application
### Examples

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id = \lambda X. \lambda x : X. x)</td>
<td>(\forall X. X \to X)</td>
</tr>
<tr>
<td>(id[bool])</td>
<td>: bool \to bool</td>
</tr>
<tr>
<td>(id[bool] \text{ true})</td>
<td>: bool</td>
</tr>
<tr>
<td>(id \text{ true})</td>
<td>type error</td>
</tr>
</tbody>
</table>

In actual programming languages, type application may be *implicit*. 
System F -- Typing rules

T-Variable
\[ x : T \in \Gamma \]
\[ \Gamma \vdash x : T \]

T-Abstraction
\[ \Gamma, x : T \vdash u : U \]
\[ \Gamma \vdash \lambda x : T. u : T \rightarrow U \]

T-Application
\[ \Gamma \vdash t : U \rightarrow T \]
\[ \Gamma \vdash u : U \]
\[ \Gamma \vdash t u : T \]

T-TypeAbstraction
\[ \Gamma, X \vdash t : T \]
\[ \Gamma \vdash \forall X. t : \forall X. T \]

T-TypeApplication
\[ \Gamma \vdash t : \forall X. T \]
\[ \Gamma \vdash t[T_1] : [T_1/X] T \]

Example:
\[ id : \forall X. X \rightarrow X \]
\[ id = \forall X. \lambda x : X. x \]
System F -- Evaluation rules

\[
\begin{align*}
E\text{-AppFun} & \quad t_1 \rightarrow t_1' \\
& \quad \frac{}{t_1 \ t_2 \rightarrow t_1' \ t_2} \\
E\text{-AppArg} & \quad t \rightarrow t' \\
& \quad \frac{}{\nu \ t \rightarrow \nu \ t'} \\
E\text{-AppAbs} & \quad (\lambda x : T.t) \nu \rightarrow [\nu/x]t \\
\end{align*}
\]

Example:
\[
id : \forall X.X \rightarrow X \\
id = \lambda X.\lambda x : X.x
\]
The doubling function

\[
double: \forall X. (X \to X) \to X \to X
\]

\[
double = \lambda X. \lambda f : X \to X. \lambda x : X. f (f x)
\]

- Instantiated with \( \text{n}at \)
  
  \[
  double_{-nat} = double [\text{n}at]
  \]
  
  : (\text{n}at \to \text{n}at) \to \text{n}at \to \text{n}at

- Instantiated with \( \text{n}at \to \text{n}at \)
  
  \[
  double_{-nat\_arrow\_nat} = double [\text{n}at \to \text{n}at]
  \]
  
  : ((\text{n}at \to \text{n}at) \to \text{n}at \to \text{n}at) \to (\text{n}at \to \text{n}at) \to \text{n}at \to \text{n}at

- Invoking \( double \)
  
  \[
  double [\text{n}at] (\lambda x : \text{n}at.\text{succ} (\text{succ} x)) 5 \to^* 9
  \]
Functions on polymorphic functions

- Consider the polymorphic identity function:
  \[ id : \forall X. X \rightarrow X \]
  \[ id = \Lambda X. \lambda x : X. x \]
- Use \( id \) to construct a pair of Boolean and String:
  \[ \text{pairid} : (\text{Bool, String}) \]
  \[ \text{pairid} = (id \text{ true, id "true"}) \]
- Abstract over \( id \):
  \[ \text{pairapply} : (\forall X. X \rightarrow X) \rightarrow (\text{Bool, String}) \]
  \[ \text{pairapply} = \lambda f : \forall X. X \rightarrow X. (f \text{ true, f "true"}) \]
Self application

• Not typeable in the simply-typed lambda calculus

\( \lambda x : ? \ . \ x \ x \)

• Typeable in System F

\( \text{selfapp} : (\forall X. X \to X) \to (\forall X. X \to X) \)

\( \text{selfapp} = \lambda x : \forall X. X \to X. x \ [\forall X. X \to X] \ x \)
The fix operator ($Y$)

- Not typeable in the simply-typed lambda calculus
  - Extension required
- Typeable in System F.

$$fix : \forall X. (X \rightarrow X) \rightarrow X$$

- Encodeable in System F with recursive types.

$$fix = ?$$

See [TAPL]
Meaning of “all types”

In the type ∀X. ..., we quantify over “all types”.

- **Predicative polymorphism**
  - X ranges over simple types.
  - Polymorphic types are “type schemes”.
  - Type inference is decidable.

- **Impredicative polymorphism**
  - X also ranges over polymorphic types.
  - Type inference is undecidable.

- **type:type polymorphism**
  - X ranges over all types, including itself.
  - Computations on types are expressible.
  - Type checking is undecidable.

Generality is used for selfapp.

Not covered by this lecture
End of Lambda slides