Lambda Calculi With Polymorphism

Ralf Lämmel
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")
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

\[ t ::= x \mid v \mid t \, t \mid t[T] \]

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

\[ T ::= X \mid T \rightarrow T \mid \forall X . T \]

Type variable

Type application

Type abstraction

Polymorphic type

Example:

\[ id : \forall X . X \rightarrow X \]

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

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

This slide is derived from Jaakko Järvi’s slides for his course "Programming Languages", CPSC 604 @ TAMU.
System F -- Typing rules

### Syntax

- **t ::=** \( x \) | \( v \) | \( tt \) | \( t [ T ] \)
- **v ::=** \( \lambda x : T . t \)
- **T ::=** \( X \) | \( T \rightarrow T \) | \( \forall X . t \)

### Evaluation rules

- **E-AppFun**
  
  \[
  t_1 \rightarrow t_1 \rightarrow t_2 \rightarrow t_1[t_2]
  \]

- **E-AppArg**
  
  \[
  t \rightarrow t \rightarrow vt \rightarrow vt
  \]

- **E-AppAbs**
  
  \[
  (\lambda x : T . t)v \rightarrow [v/x]t
  \]

- **E-TypeApp**
  
  \[
  t_1 : U \rightarrow T \rightarrow t_1 : U \rightarrow t_2 \rightarrow t_1[t_2]
  \]

- **E-TypeAppAbs**
  
  \[
  \Lambda X . t[T_1] : [T_1/X]T
  \]

### Typing rules

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

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

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

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

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

**Example:**

\[
\begin{align*}
id & : \forall X . X \rightarrow X \\
id & = \Lambda X . \lambda x : X . x
\end{align*}
\]
System F -- Evaluation rules

E-AppFun
\[
\frac{t_1 \rightarrow t_1'}{t_1.t_2 \rightarrow t_1'.t_2}
\]

E-AppArg
\[
\frac{t \rightarrow t'}{v.t \rightarrow v.t'}
\]

E-AppAbs
\[
(\lambda x : T.t) \ v \rightarrow [v/x]t
\]

E-TypeApp
\[
\frac{t_1 \rightarrow t_1'}{t_1[T] \rightarrow t_1'[T]}
\]

E-TypeAppAbs
\[
(\forall X.X)[T] \rightarrow [T/X]t
\]

Example:

\[
id : \forall X.X \rightarrow X
\]

\[
id = \forall X.\lambda x : X.x
\]
### Examples

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id =(\forall 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] true</td>
<td>: bool</td>
</tr>
<tr>
<td>id true</td>
<td>type error</td>
</tr>
</tbody>
</table>

In actual programming languages, type application may be *implicit.*
The doubling function

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

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

- Instantiated with \( \text{nat} \to \text{nat} \)
  
  \[ \text{double}_\text{nat}_\text{arrow}_\text{nat} = \text{double} \ [\text{nat} \to \text{nat}] \]
  
  : \(((\text{nat} \to \text{nat}) \to \text{nat} \to \text{nat}) \to (\text{nat} \to \text{nat}) \to \text{nat} \to \text{nat}\)

- Invoking \text{double}
  
  \[ \text{double} \ [\text{nat}] \ (\lambda x : \text{nat}. \text{succ} (\text{succ} x)) \ 5 \to * \ 9 \]
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:

\[ \text{pairid} : (\text{Bool}, \text{String}) \]

\[ \text{pairid} = (id \, \text{true}, id \, \text{“true”}) \]

• Abstract over \( id \):

\[ \text{pairapply} : (\forall X. X \rightarrow X) \rightarrow (\text{Bool}, \text{String}) \]

\[ \text{pairapply} = \lambda f : \forall X. X \rightarrow X. (f \, \text{true}, f \, \text{“true”}) \]
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.

\[
\text{fix} : \forall X.(X \to X) \to X
\]

- Encodeable in System F with recursive types.

\[
\Gamma \vdash t : T \to T \quad \Rightarrow \quad \Gamma \vdash \text{fix} t : T
\]

\[
\text{fix} = ?
\]

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.
Kinds of polymorphism

- Parametric polymorphism ("all types")

- **Bounded polymorphism** ("subtypes")

- Ad-hoc polymorphism ("some types")

- Existential types ("exists as opposed to for all")
What is subtyping anyway?

- We say $S$ is a subtype of $T$.

  $S <: T$

- **Liskov substitution principle**: For each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$.

- **Practical type checking**: Any expression of type $S$ can be used in any context that expects an expression of type $T$, and no type error will occur.
Why subtyping

• Function in near-to-C:

```c
void foo(struct { int a; } r) {
    r.a = 0;
}
```

• Function application in near-to-C:

```c
struct K { int a; int b; } K k;
foo(k); // error
```

• Intuitively, it is safe to pass `k`.
  Subtyping allows it.
Subsumption
(Substitututability of supertypes by subtypes)

\[ \Gamma 
\vdash t : U \\
U \leq T \\
\Gamma 
\vdash t : T \]

This rule implies, for example, that the actual argument’s type in a function application can also be a subtype of the function’s argument type.
Structural subtyping for records

• A subtype may have additional label/type pairs.
• A subtype may permute the label/type pairs.
• A subtype may use subtypes for label/type pairs.
A subtype may have additional label/type pairs.

Example:

\[ \{ \text{key : bool, value : int, map : int \rightarrow int} \} <: \{ \text{key : bool, value : int} \} \]
A subtype may permute the label/type pairs.

S-RecordPermutation
\[ \{ l_i : T_i \}_{i=1 \ldots n} \] is a permutation of \[ \{ k_j : U_j \}_{j=1 \ldots n} \]

\[ \{ l_i : T_i \}_{i=1 \ldots n} \preceq \{ k_j : U_j \}_{j=1 \ldots n} \]

Example:
\[ \{ \text{key} : \text{bool}, \text{value} : \text{int} \} \preceq \{ \text{value} : \text{int}, \text{key} : \text{bool} \} \]
A subtype may use subtypes for label/type pairs.

\[
\begin{align*}
\text{S-RecordElements} \\
\text{for each } i & \quad T_i \ll U_i \\
\{l_i : T_{i} \}_{i\in 1...n} & \ll \{l_i : U_{i} \}_{i\in 1...n}
\end{align*}
\]

Example:

\[
\{\text{field1 : bool, field2 : \{val : bool\}}\} \ll \{\text{field1 : bool, field2 : \{\}}\}
\]
General rules for subtyping

• Subsumption
• Reflexivity of subtyping
• Transitivity of subtyping
• Subtyping for function types
• Supertype of everything
• Up and down cast
General rules for subtyping

- **Reflexivity**
  \[ T <: T \]

- **Transitivity**
  \[ T <: U \quad U <: V \quad \Rightarrow \quad T <: V \]

- Example which needs transitivity

  Prove that \( \{a : \text{bool}, b : \text{int}, c : \{l : \text{int}\}\} <: \{c : \{\}\}\)
Subtyping of functions

• Subsumption provides subtyping for function arguments and results.
• An additional rule is needed to provide subtyping for function types.
• Compare this to the need for subtyping rules for record types.
• Consider the following function application:
  \((\lambda f : \{a:\text{int}, b:\text{int}\} \rightarrow \{c:\text{int}\} . f \{a:42, b:88\}) \ g)\)
• Several types can be permitted for \(g\):
  ✦ \(g : \{a:\text{int}, b:\text{int}\} \rightarrow \{c:\text{int}\}\)
  ✦ \(g : \{a:\text{int}\} \rightarrow \{c:\text{int}, d:\text{int}\}\)
  ✦ ...

The actual function may “use” less fields and “return” more fields.
Subtyping of functions

• Function subtyping
  ✦ covariant on return types
  ✦ contravariant on parameter types

\[
\begin{align*}
T_2 &<: T_1 & U_2 &<: U_1 \\
\hline
T_1 \to U_2 &<: T_2 \to U_1
\end{align*}
\]
Supertype of everything

- $T ::= \ldots \mid \textit{top}$
  - The most general type
  - The supertype of all types

$T \ll\!: \textit{top}$
Remember type annotation?

• Syntax:

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

• Typing rule:

\[ \Gamma \vdash t : T \quad \Gamma \vdash t \text{ as } T : T \]

• Evaluation rules:

\[ t \rightarrow u \]

\[ t \text{ as } T \rightarrow u \text{ as } T \]

\[ v \text{ as } T \rightarrow v \]
Annotation as up-casting

• Illustrative type derivation:

\[ \vdash t : U \quad U <: T \quad \vdash t : T \quad \vdash t \text{ as } T : T \]

That is, type annotation automagically works as up-cast because of the subsumption rule.

• Example:

\((\lambda x : \text{bool}.\{a = x, b = \text{false}\}) \text{ true as } \{a : \text{bool}\}\)
Annotation as down-casting

• Typing rule:

\[
\frac{\Gamma \vdash t : U}{\Gamma \vdash t \text{ as } T : T}
\]

• Evaluation rules:

\[
\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T}
\]

\[
\frac{\vdash v : T}{v \text{ as } T \rightarrow v}
\]

Potentially too liberal

Runtime type check
Reminder: A type system is a \textit{tractable syntactic} method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [B.C. Pierce]

We violate this definition because some rules are not “syntax-driven”!
Violation of syntax direction

• Consider an application:

\[ t \ u \text{ where } t \text{ of type } U \rightarrow V \text{ and } u \text{ of type } S. \]

• A type checker would need to figure out that \( S \leq U. \)
  
  ✦ This is hard with the rules so far (transitivity, subsumption).
  
  ✦ The rules need to be redesigned.
Analysis of subsumption

\[ \text{T-Subsumption} \]
\[ \Gamma \vdash t : U \quad U <: T \]
\[ \Gamma \vdash t : T \]

• The term in the conclusion can be anything.
  It is just a metavariable.

• Example: Which rule should you apply here?
  \[ \Gamma \vdash (\lambda x : U.t) : ? \]

T-Abstraction or T-Subsumption?
Analysis of transitivity

• $U$ does not appear in conclusion.
  
  Thus, to show $T <: V$, we need to guess a $U$.

• For instance, try to show the following:
  
  $$\{y: \text{int}, x: \text{int}\} <: \{x: \text{int}\}$$
Analysis of transitivity

• What is the purpose of transitivity?

Chaining together separate subtyping rules for records!

S-RecordPermutation
\[
\{ l_i : T_i^{i \in 1 \ldots n} \} \text{ is a permutation of } \{ k_j : U_j^{j \in 1 \ldots n} \} \\
\{ l_i : T_i^{i \in 1 \ldots n} \} \:<: \{ k_j : U_j^{j \in 1 \ldots n} \}
\]

S-RecordElements
\[
\text{for each } i \quad T_i \:<: U_i \\
\{ l_i : T_i^{i \in 1 \ldots n} \} \:<: \{ l_i : U_i^{i \in 1 \ldots n} \}
\]

S-RecordNewFields
\[
\{ l_i : T_i^{i \in 1 \ldots n+k} \} \:<: \{ l_i : T_i^{i \in 1 \ldots n} \}
\]
Algorithmic subtyping

• Replace all previous rules by a single rule.

\[
\begin{align*}
\text{S-Record:} & \quad \{l_i \mid i \in 1 \ldots n\} \subseteq \{k_j \mid j \in 1 \ldots m\} \quad l_i = k_j \implies U_i <: T_j \\
& \quad \{k_j : U_j \mid i \in 1 \ldots m\} <: \{l_i : T_i \mid i \in 1 \ldots n\}
\end{align*}
\]

• Correctness / completeness of new rule can be shown.

• Maintain extra rule for function types.

\[
\begin{align*}
\text{S-Function:} & \quad T_1 <: T_2 \quad U_1 <: U_2 \\
& \quad T_2 \rightarrow U_1 <: T_1 \rightarrow U_2
\end{align*}
\]
Algorithmic subtyping

• The subsumption rule is still not syntax-directed.

• The rule is essentially used in function application.

• Express subsumption through an extra premise.

\[
\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : V \quad V <: U
\]

\[
\Gamma \vdash t\ u : T
\]

• Retire subsumption rule.
Kinds of polymorphism

- Parametric polymorphism ("all types")
- Bounded polymorphism ("subtypes")
- Ad-hoc polymorphism ("some types")
- Existential types ("exists as opposed to for all")
Kinds of polymorphism

- Parametric polymorphism ("all types")
- Bounded polymorphism ("subtypes")
- Ad-hoc polymorphism ("some types")
- Existential types ("exists as opposed to for all")
Existential types serve a specific purpose:

A means for **information hiding (encapsulation)**.

- Remember predicate logic. \( \forall x. P(x) \equiv \neg (\exists x. \neg P(x)) \)

- Existential types can be encoded as universal types; see [TAPL].
Overview

- Syntax of types: \[ T ::= \cdots \mid \{ \exists X, T \} \]

- Normal forms: \[ v ::= \cdots \mid \{ *T, v \} \]

- Terms: \[ t ::= \cdots \mid \{ *T, t \} \text{ as } T \]
  \[ \mid \text{let } \{ X, x \} = t \text{ in } t \]
Constructing existentials

- A record:
  \[ r : \{ a : \text{nat}, b : \text{nat} \rightarrow \text{nat} \} \]
  \[ r = \{ a = 1, b = \lambda x : \text{nat} . \text{pred} \, x \} \]

- A package with \( \text{nat} \) as hidden type:
  \[ p : \{ \exists X, \{ a : X, b : X \rightarrow \text{nat} \} \} \]
  \[ p = \{ \ast \text{nat}, r \} \]

- The type system makes sure that \( \text{nat} \) is **inaccessible** from outside.
Multiple types make sense for the package. Hence, the programmer must provide an annotation upon construction.

- \( r : \{ a : \text{nat}, b : \text{nat} \rightarrow \text{nat} \} \)

\[
r = \{ a = 1, b = \lambda x : \text{nat} . \text{pred} \ x \}
\]

- \( p = \{*\text{nat}, r \} \text{ as } \exists X, \{ a : X, b : X \rightarrow X \} \}

\( p \) has type: \( \exists X, \{ a : X, b : X \rightarrow X \} \)

\( p \) *preferred*  

- \( p' = \{*\text{nat}, r \} \text{ as } \exists X, \{ a : X, b : X \rightarrow \text{nat} \} \}

\( p' \) has type: \( \exists X, \{ a : X, b : X \rightarrow \text{nat} \} \)
We can have the same existential type with different representation types.

- \( p_1 = \{\ast \mathsf{nat}, \{a = 1, b = \lambda x: \mathsf{nat}. \text{iszero } x\}\} \)
  
as \( \exists X, \{a: X, b: X \rightarrow \mathsf{bool}\} \)

- \( p_2 = \{\ast \mathsf{bool}, \{a = \text{false}, b = \lambda x: \mathsf{bool}. \text{if } x \text{ then false else true}\}\} \)
  
as \( \exists X, \{a: X, b: X \rightarrow \mathsf{bool}\} \)
Unpacking existentials
(Opening package, importing module)

• Example -- apply \( b \) to \( a \):

\[
\text{let } \{X,x\} = \text{p2 in } (x.b \ x.a) \rightarrow \ast \text{ true : bool}
\]

• Syntax:

\[
\text{let } \{X,x\} = t \text{ in } t'
\]

✦ The value \( x \) of the existential becomes available.

✦ The representation type is not accessible (only \( X \)).
Typing rules

**T-PackExistential**

\[ \Gamma \vdash t : [U/X] T \]

\[ \Gamma \vdash \{U, t\} \text{ as } \{\exists X, T\} : \{\exists X, T\} \]

**T-UnpackExistential**

\[ \Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2 \]

\[ \Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2 \]

Substitution checks that the abstracted type of \( t \) can be instantiated with the hidden type to the actual type of \( t \).

Only expose abstract type of existential!
Evaluation rules

E-Pack

\[
\frac{t \rightarrow t'}{
\{ *T, t \} \text{ as } U \rightarrow \{ *T, t' \} \text{ as } U}
\]

E-Unpack

\[
\frac{t_1 \rightarrow t_1'}{\text{let } \{ X, x \} = t_1 \text{ in } t_2 \rightarrow \text{let } \{ X, x \} = t_1' \text{ in } t_2}
\]

E-UnpackPack

\[
\text{let } \{ X, x \} = (\{ *T, v \} \text{ as } U) \text{ in } t_2 \rightarrow [T/X][v/x]t_2
\]

The hidden type is known to the evaluation, but the type system did not expose it; so \( t_2 \) cannot exploit it.
Illustration of information hiding

• The type can be used in the scope of the unpacked package.

\[
\text{let } \{X, x\} = t \text{ in } (\lambda y:X. x.b y) \ x.a \rightarrow \text{false : bool}
\]

• The representation type must remain abstract.

\[
t = \{\text{*nat, } a = 1, b = \lambda x: \text{nat}. \text{iszero } x\} \text{ as } \{\exists X, \{a:X, b:X \rightarrow \text{bool}\}\}\}
\]

\[
\text{let } \{X,x\} = t \text{ in pred x.a} \quad \text{// Type error!}
\]

• The type must not leak into the resulting type:

\[
\text{let } \{X, x\} = t \text{ in x.a} \quad \text{// Type error!}
\]
• **Summary**: Lambdas with somewhat sexy types
  - *Done*: ∀, ∃, <:, ...
  - *Not done*: μ, ...

• **Prepping**: “Types and Programming Languages”
  - *Chapters*: 15, 16, 22, 23, 24

• **Outlook**:
  - Object calculi
  - Process calculi
  - More paradigms