Lambda Calculi With Polymorphism

Ralf Lämmel

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

• What’s the identity function?

• In the simple typed lambda calculus, this depends on the type!

• Examples

  ✦ \( \lambda x : \text{bool}. \ x \)
  ✦ \( \lambda x : \text{nat}. \ x \)
  ✦ \( \lambda x : \text{bool} \rightarrow \text{bool}. \ x \)
  ✦ \( \lambda x : \text{bool} \rightarrow \text{nat}. \ x \)
  ✦ ...
Polymorphism

• Polymorphic function
  ✦ a function that accepts *many types* of arguments.

• Kinds of polymorphism
  ✦ Parametric polymorphism ("all types")
  ✦ Bounded polymorphism ("subtypes")
  ✦ Ad-hoc polymorphism ("some types")

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

• Kinds of polymorphism

✦ **Parametric polymorphism** ("all types")

✦ Bounded polymorphism ("subtypes")

✦ Ad-hoc polymorphism ("some types")
System F -- Syntax

\[ t ::= x \mid \nu \mid t \; t \mid t[T] \]

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

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

Type application

Type abstraction

Polymorphic type
System F -- Typing rules

\[ \frac{\text{T-Variable}}{\Gamma, x : T \vdash x : T} \]
\[ \frac{\Gamma, x : T \vdash u : U}{\Gamma, x : T \vdash \lambda x : T.u : T \rightarrow U} \]

\[ \frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash tu : T} \]

\[ \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash \forall X.t : \forall X.T} \]
\[ \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[T_1] : [T_1/X]T} \]
System F -- Evaluation rules

E-AppFun

\[ t_1 \rightarrow t_1' \]
\[ t_1 \ t_2 \rightarrow t_1' \ t_2 \]

E-AppArg

\[ t \rightarrow t' \]
\[ v \ t \rightarrow v \ t' \]

E-AppAbs

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

E-TypeApp

\[ t_1 \rightarrow t_1' \]
\[ t_1[T] \rightarrow t_1'[T] \]

E-TypeAppAbs

\[ (\forall X . t)[T] \rightarrow [T/X]t \]
Type abstraction and application

• Evaluation

\[
\begin{align*}
\text{E-TypeApp} & \quad t_1 \rightarrow t_1' \\
\frac{t_1[T] \rightarrow t_1'[T]}{}
\end{align*}
\]

\[
\begin{align*}
\text{E-TypeAppAbs} & \quad (\forall X. t)[T] \rightarrow [T/X]t
\end{align*}
\]

• Typing

\[
\begin{align*}
\text{T-TypeAbstraction} & \quad \Gamma, X \vdash t : T \\
\frac{\Gamma \vdash \forall X. t : \forall X. T}{}
\end{align*}
\]

\[
\begin{align*}
\text{T-TypeApplication} & \quad \Gamma \vdash t : \forall X. T \\
\frac{\Gamma \vdash t[T_1] : [T_1/X]T}{}
\end{align*}
\]

♦ Type variables are in the environment.

♦ Type variables are subject to (implicit) renaming.
Examples

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

There is no type-argument deduction at this point.
The doubling function

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

- Instantiated with \textit{nat}

\[
double_{nat} = double \ [nat] \ : (nat \to nat) \to nat \to nat
\]

- Instantiated with \textit{nat \to nat}

\[
double_{nat\_arrow\_nat} = double \ [nat \to nat] \ : ((nat \to nat) \to nat \to nat) \to (nat \to nat) \to nat \to nat
\]

- Invoking \textit{double}

\[
double \ [nat] (\lambda x : nat. \text{succ} (\text{succ} x)) 5 \to^* 9
\]
Self application

• Not typeable in the simply-typed lambda calculus

\[ \lambda x : ? \cdot x \; x \]

• Typeable in System F

\[ \text{selfapp} = \lambda x : \forall X.X \to X.x \left[ \forall X.X \to X \right] x \]

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

- Not typeable in the simply-typed lambda calculus
- Typeable in System F.
  
  $$\text{fix} : \forall X. (X \rightarrow X) \rightarrow X$$

- Encodeable in System F with recursive types.
  
  $$\text{fix} = ?$$

See [TAPL]
Remember lists?
(in the simply-typed lambda calculus)

- New type:  \( \ldots | \text{List} \ T \)

- New syntax:  \( \ldots | \text{nil}[T] | \text{const}[T] \ t \ t | \text{isnil}[T] \ t | \text{head}[T] \ t | \text{tail}[T] \ t \)

- New congruence rules, e.g.:
  \[
  \frac{\ t_1 \rightarrow t'_1}{\text{cons}[T] \ t_1 \ t_2 \rightarrow \text{cons}[T] \ t'_1 \ t_2}
  \]

- New computation rules, e.g.:
  \[
  \text{head}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_1
  \]

- New typing rules, e.g.:
  \[
  \Gamma \vdash t : \text{List} \ T \\
  \Gamma \vdash \text{head}[T] \ t : \overline{T}
  \]
Lists in System F

• Types of list operations
  \(\text{nil} : \forall X. \text{List } X\)
  \(\text{cons} : \forall X. X \to \text{List } X \to \text{List } X\)
  \(\text{isnil} : \forall X. \text{List } X \to \text{bool}\)
  \(\text{head} : \forall X. \text{List } X \to X\)
  \(\text{tail} : \forall X. \text{List } X \to \text{List } X\)

• List \(T\) can be encoded.
  \(\forall X. (T \to U \to U) \to U \to U\)
  (see [TAPL] Chapter 23.4; requires fix)
Meaning of “forall”

In a universal type, \( \forall X \ldots \), 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.
Limits of type schemes

• Consider the polymorphic identity function:

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

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

• 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 \( id \):

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

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

- A means for **information hiding and abstraction**.

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

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

- Syntax: \{\(\exists X, T\}\}
∀ vs. ∃

- $t$ of type $∀X.T$
  - Logical view: $t$ has value of type $[S/X]T$ for any $S$.
  - Operational view:
    $t$ is a mapping from type $S$ to a term of type $[S/X]T$.
- $t$ of type $∃X.T$
  - Logical view: $t$ has value of type $[S/X]T$ for some $S$.
  - Operational view:
    $t$ is a pair of a type $S$ and a term $u$ of type $[S/X]T$.
    $S$ is hidden.
    Notation: $\{ *S, u \}$
Constructing existentials \( \{ *S, u \} \)

- \( \{ *S, u \} \) should be considered a package (module).
- The type system makes sure that \( S \) is inaccessible from outside.
- Consider the following package:
  \[
p = \{ *\text{nat}, \{ a = 1, b = \lambda x:\text{nat}. \text{pred} \ x \} \}
\]
- Multiple types make sense:
  - \( \{ \exists X, \{ a:X, b:X \rightarrow X \} \} \)
  - \( \{ \exists X, \{ a:X, b:X \rightarrow \text{nat} \} \} \)
- Existential types require programmer annotation.
Annotating existentials

\[ p = \{ \ast \text{nat}, \{ a = 1, b = \lambda x: \text{nat}. \text{pred} \ x \} \} \text{ as } \{ \exists X, \{ a: X, b: X \to X \} \} \]

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

\[ p' = \{ \ast \text{nat}, \{ a = 1, b = \lambda x: \text{nat}. \text{pred} \ x \} \} \text{ as } \{ \exists X, \{ a: X, b: X \to \text{nat} \} \} \]

\( p' \) has type: \( \{ \exists X, \{ a: X, b: X \to \text{nat} \} \} \)
Information hiding

Packages with different representation types can have the same existential type.

- $p1 = \{\ast \text{nat}, \{a = 1, b = \lambda x: \text{nat}. \text{iszero } x\}\}$
  
  as $\{\exists X, \{a: X, b: X \rightarrow \text{bool}\}\}$

- $p2 = \{\ast \text{bool}, \{a = \text{false}, b = \lambda x: \text{bool}. \text{if } x \text{ then false else true}\}\}$
  
  as $\{\exists X, \{a: X, b: X \rightarrow \text{bool}\}\}$
Unpacking existentials
(Opening package, importing module)

• The existential becomes available.

• The representation type is not accessible.

• Only the capabilities of the existential type are accessible.

• Example:

\[
\begin{align*}
\text{let } \{X,t\} = \text{p2 in } (t.b \ t.a) & \rightarrow^* \text{true : bool}
\end{align*}
\]
Existential types

Typing rules

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

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!

\[ \Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2 \]
\[ \Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2 \]
Evaluation rules

E-Pack
\[ t \rightarrow t' \]
\[ \{\ast T, t\} \text{ as } U \rightarrow \{\ast T, t'\} \text{ as } U \]

E-Unpack
\[ t_1 \rightarrow t'_1 \]
\[ \text{let } \{X, x\} = t_1 \text{ in } t_2 \rightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2 \]

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

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 representation type must not leak / must remain abstract.

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

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

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

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

• The type cannot be free in the resulting type:

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

• Kinds of polymorphism
  ✦ Parametric polymorphism (“all types”)
  ✦ **Bounded polymorphism (“subtypes”)**
  ✦ Ad-hoc polymorphism (“some types”)

This slide is derived from Jaakko Järvi’s slides for his course “Programming Languages”, CPSC 604 @ TAMU.
What is subtyping anyway?

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

  $S <: T$

- **Substitutability** (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

- Substitutability is captured with the subsumption rule:

\[ \Gamma \vdash t : U \quad U <: T \]
\[ \Gamma \vdash t : T \]

- Adding this rules requires revisiting other rules.

Subtyping is a crosscutting extension.
Subtyping in simplified setting

- Simply-typed lambda calculus +
  - records
  - Booleans
  - integers

- **Structural subtyping for records**
Subtyping for records

• Order of fields does not matter.

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

• Example:

\{key : bool, value : int\} <: \{value : int, key : bool\}
Subtyping for records

- We can always add new fields in the end.

\[
\text{S-RecordNewFields} \\
\{ l_i : T_i \}_{i \in 1 \ldots n+k} <: \{ l_i : T_i \}_{i \in 1 \ldots n}
\]

- Example:

\{
key : bool, value : int, map : int -> int\} \leq \{
key : bool, value : int\}
Subtyping for records

• We can subject the fields to subtyping.

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

• Example:

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

• Reflexivity  \[ T <: T \]

• Transitivity  \[
\begin{align*}
T &: U \\
U &: V \\
\hline
T &: V
\end{align*}
\]

• Example

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

• Assume that a function $f$ of the following type is expected:

$$f : T \rightarrow U$$

• Then it is safe to pass an actual function $g$ such that:

$$g : T' \rightarrow U'$$

$T <: T'$ ($g$ expects less fields than $f$)

$U' <: U$ ($g$ gives more fields than $f$)
Subtyping and functions

- Function subtyping
  - covariant on return types
  - contravariant on parameter types

\[
T_2 <: T_1 \quad U_2 <: U_1
\]

\[
T_1 \rightarrow U_2 <: T_2 \rightarrow U_1
\]
Supertype of everything

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

$T \subseteq \text{top}$
Remember type annotation?

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

• Typing rule:
  \[
  \Gamma \vdash t : T \\
  \Gamma \vdash t \text{ as } T : T
  \]

• Evaluation rules:
  \[
  t \rightarrow u \\
  t \text{ as } T \rightarrow u \text{ as } T
  \]

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

• Illustrative type derivation:

\[ \frac{\Gamma \vdash t : U \quad U \preceq T}{\Gamma \vdash t : T} \]

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

• Example:

\[ (\lambda x: \text{bool}.\{a = x, b = \text{false}\}) \text{ true as } \{a : \text{bool}\} \]
Type annotation with down-casting

• Typing rule:

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

Potentially too liberal

• Evaluation rules:

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

Runtime type check

\[
\frac{v : T}{v \text{ as } T \rightarrow v}
\]
Typing rules so far

T-Record

for each $i$, $\Gamma \vdash t_i : T_i$

$\Gamma \vdash \{ l_i = t_i \}_{i \in 1 \ldots n} : \{ l_i : T_i \}_{i \in 1 \ldots n}$

T-Projection

$\Gamma \vdash t : \{ l_i : T_i \}_{i \in 1 \ldots n}$

$\Gamma \vdash t.l_j : T_j$

T-Subsumption

$\Gamma \vdash t : U \quad U <: T$

$\Gamma \vdash t : T$

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

$\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U$

$\Gamma \vdash t\ u : T$

T-True

$\vdash true : bool$

T-False

$\vdash false : bool$
Reminder

A type system is a 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]
Violation of syntax direction

• Consider an application:

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

• Type checker must figure out that \( S <: U \).

✦ This is hard with the rules so far.

✦ The rules need to be redesigned.
Analysis of subsumption

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.

• E.g. which rule should you apply here?

\[ \Gamma \vdash (\lambda x : U.t) : ? \]

T-Abstraction or T-Subsumption?
Analysis of transitivity

S-Transitivity

\[
T <: U \quad U <: V \\
\hline \\
T <: V
\]

• U does not appear in conclusion.

Thus, to show \( T <: V \), we need to guess a \( U \).

• For instance, try to show the following:

\[
\{y:int, x:int\} <: \{x:int\}
\]
Analysis of transitivity

• What is the purpose of transitivity?

Chaining together separate subtyping rules for records!

\[
\text{S-RecordPermutation} \\
\{ l_i : T_i^{i\in1...n} \} \text{ is a permutation of } \{ k_j : U_j^{j\in1...n} \} \\
\{ l_i : T_i^{i\in1...n} \} <: \{ k_j : U_j^{j\in1...n} \}
\]

\[
\text{S-RecordElements} \\
\text{for each } i \quad T_i <: U_i \\
\{ l_i : T_i^{i\in1...n} \} <: \{ l_i : U_i^{i\in1...n} \}
\]

\[
\text{S-RecordNewFields} \\
\{ l_i : T_i^{i\in1...n+k} \} <: \{ l_i : T_i^{i\in1...n} \}
\]
Algorithmic subtyping

• Replace all previous rules by a single rule.

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

• Correctness / completeness of new rule can be shown.

• Maintain extra rule for function types.

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

• The subsumption rule is still not syntax-directed.

• The rule is essentially used in function application.

• Express subsumption through an extra premise.

\[
\begin{align*}
\text{T-Application} \\
\Gamma \vdash t : U \to T & \quad \Gamma \vdash u : V \quad V <: U \\
\hline
\Gamma \vdash t \ u : T
\end{align*}
\]

• Retire subsumption rule.
• **Summary**: Lambdas with somewhat sexy types
  - *Done*: \(\forall, \exists, <, \ldots\)
  - *Not done*: \(\mu, \ldots\)

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

• **Lab**: “*The Typed Lambda Calculus in Prolog*”

• **Outlook**:
  - Functional programming with Haskell
  - Denotational semantics