Let $x = 1$ in ...

$x(1)$.

$x.set(1)$

Programming Language Theory

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

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

This slide is derived from Jaakko Järvi’s slides for his course "Programming Languages", CPSC 604 @ TAMU.
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")
  
    ✦ Existential types ("exists as opposed to for all")

  ✦ Bounded polymorphism ("subtypes")

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

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

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

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

Type application

Type abstraction

Polymorphic type

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

Type variables are in the context

Type variables are subject to alpha conversion.

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

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

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

- **E-AppArg**
  \[ t \rightarrow t' \]
  \[ v \rightarrow v' \]

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

- **E-TypeApp**
  \[ t_1 \rightarrow t_1' \]
  \[ t_1[T] \rightarrow t_1'[T] \]

- **E-TypeAppAbs**
  \[ (\forall X.t)[T] \rightarrow [T/X]t \]
There is no inference of type arguments at this point.

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

<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[bool])</td>
<td>(bool \rightarrow 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>
The doubling function

double=\( \lambda X.\lambda f : X \rightarrow X.\lambda x : X. f (f x) \)

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

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

- Invoking \( double \)
  
  \( double \ [\text{nat}] \ (\lambda x : \text{nat}. \text{succ} (\text{succ} x)) \ 5 \rightarrow^* 9 \)
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:
  
  $pairid : (\text{Bool}, \text{String})$

  $pairid = (id \text{ true}, id \text{ "true"})$

• Abstract over $id$:
  
  $pairapply : (\forall X. X \rightarrow X) \rightarrow (\text{Bool}, \text{String})$

  $pairapply = \lambda f : \forall X. X \rightarrow X. (f \text{ true}, f \text{ "true"})$
Self application

• Not typeable in the simply-typed lambda calculus

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

• 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.

\[
\text{fix} = ?
\]

See [TAPL]
Lists in System F

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

• List \( T \) can be encoded.
  \( \forall X. (T \rightarrow U \rightarrow U) \rightarrow U \rightarrow U \)
  (see [TAPL] Chapter 23.4; requires \texttt{fix})

No new syntax needed!
Meaning of ‘all types’

In the 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.

We used this generality for `selfapp`.

Not covered by this lecture
Polymorphism

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

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

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

- Existential types serve a specific purpose:
  
  A means for \textit{information hiding (encapsulation)}. 
Overview

- Syntax of types: \( T ::= \cdots \mid \{ \exists X, T \} \)
- Normal forms: \( v ::= \cdots \mid \{ *T, v \} \)
- Terms: \( t ::= \cdots \mid \{ *T, t \} \text{ as } T \)
  \| \text{let } \{X, x\} = t \text{ in } t \)
∀ vs. ∃ -- Operational view

• $t$ of type $∀X.T$
  ✦ $t$ maps type $S$ to a term of type $[S/X]T$.

• $t$ of type $\{∃X, T\}$
  ✦ $t$ is a pair $\{ *S, u \}$ of a type $S$ and a term $u$ of type $[S/X]T$.
  ✦ $S$ is hidden. (This is indicated with “*”.)

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.
∀ vs. ∃ -- Logical view

- $t$ of type $\forall X.T$
  - $t$ has value of type $[S/X]T$ for any $S$.
- $t$ of type $\exists X, T$
  - $t$ has value of type $[S/X]T$ for some $S$.
Constructing existentials

• Consider the following package:
  \[ p = \{ \ast \text{nat}, \{ a = 1, b = \lambda x: \text{nat}. \text{pred} \ x \} \} \]

• The type system makes sure that nat is \textit{inaccessible} from outside.

• Multiple types make sense for the package:
  
  \[ \{ \exists X, \{ a: X, b: X \to X \} \} \]
  
  \[ \{ \exists X, \{ a: X, b: X \to \text{nat} \} \} \]

  Hence, the programmer must provide an annotation upon construction.
Different annotations for the same packaged value

- \( p = \{ *\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' = \{ *\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: \( \exists X, \{ a: X, b: X \to \text{nat} \} \)
Same existential type with different representation types

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

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

• let \( \{X,x\} = t \) in \( t' \)

  ✦ The value \( x \) of the existential becomes available.
  ✦ The representation type is not accessible (only \( X \)).

• Example:

  let \( \{X,x\} = p2 \) in (x.b x.a) →* true : bool
Effective information hiding

- The representation type must remain abstract.

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

let \( \{X, x\} = t \) in \( \text{pred } x.a \)  // Type error!

- The type must not leak into the resulting type:

let \( \{X, x\} = t \) in \( x.a \)  // Type error!

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

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

This slide is derived from Jaakko Järvi’s slides for his course “Programming Languages”, CPSC 604 @ TAMU.
Typing rules

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

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

E-Pack

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

E-Unpack

\[ t_1 \rightarrow t'_1 \]
\[
\begin{array}{l}
\text{let } \{X, x\} = t_1 \text{ in } t_2 \rightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2
\end{array}
\]

E-UnpackPack

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

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

- Kinds of polymorphism

  - Parametric polymorphism ("all types")

  - Existential types ("exists as opposed to for all")

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

  - Ad-hoc polymorphism ("some types")
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*. 

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.
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 of supertypes by subtypes)

• Typing rule:

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

• Adding this rules requires revisiting other rules.

Subtyping is a crosscutting extension.
Structural subtyping for records

• Simply-typed lambda calculus +
  ✦ Booleans
  ✦ integers
  ✦ extensible records
Subtyping for records

- Order of fields does not matter:

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

- Example:

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

• We can always add new fields in the end.

\[ S-\text{RecordNewFields} \{ l_i : T_i \}_{i=1}^{n+k} \prec \{ l_i : T_i \}_{i=1}^{n} \]

• Example:

\[ \{ \text{key: bool, value: int, map: int -> int} \} \prec \{ \text{key: bool, value: int} \} \]
Subtyping for records

• We can subject the fields to subtyping.

\[ S-\text{RecordElements} \]

\[
\begin{array}{c}
\text{for each } i \\
T_i <: U_i \\
\{l_i : T_i; i \in 1...n\} <: \{l_i : U_i; i \in 1...n\}
\end{array}
\]

• Example:

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

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

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Optional material: not covered in the lecture
General rules for subtyping

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

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

- **Example**

  Prove that \{a : \text{bool}, b : \text{int}, c : \{l : \text{int}\}\} <: \{c : \{\}\}\]
General rules for subtyping:

**Subtyping of 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 $f$)*
General rules for subtyping:

**Subtyping of functions**

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

\[
T_2 <: T_1 \quad U_2 <: U_1 \\
T_1 \rightarrow U_2 <: T_2 \rightarrow U_1
\]
General rules for subtyping:

**Supertype of everything**

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

$T <: \text{top}$
Remember type annotation?

• Syntax:

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

• Typing rule:

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

• Evaluation rules:

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

\[
v \text{ as } T \rightarrow v
\]
General rules for subtyping:

Annotation as up-casting

• Illustrative type derivation:

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

• Example:

\[(\lambda x : \text{bool}.\{a = x, b = \text{false}\}) \text{ true as } \{a : \text{bool}\}\]
General rules for subtyping:

Annotation as down-casting

- Typing rule:
  \[ \Gamma \vdash t : U \quad \Gamma \vdash t \text{ as } T : T \]

- Evaluation rules:
  \[ t \rightarrow u \quad \frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T} \]
  \[ \Gamma \vdash v : T \quad \frac{\Gamma \vdash v : T}{v \text{ as } T \rightarrow v} \]

Potentially too liberal

Runtime type check

Note that this type condition on evaluation relation is a check performed at run-time.
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]

We violate this definition!

Optional material: not covered in the lecture
Typing rules so far

- **T-Record**
  
  \[
  \text{for each } i, \quad \Gamma \vdash t_i : T_i \\
  \Gamma \vdash \{ l_i = t_i \}_{i \in 1...n} : \{ l_i : T_i \}_{i \in 1...n}
  \]

- **T-Projection**
  
  \[
  \Gamma \vdash t : \{ l_i : T_i \}_{i \in 1...n} \\
  \Gamma \vdash t.l_j : T_j
  \]

- **T-Subsumption**
  
  \[
  \Gamma \vdash t : U \\
  U <: T \\
  \Gamma \vdash t : T
  \]

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

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

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

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

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

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

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

• \( U \) does not appear in conclusion.

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

• For instance, try to show the following:

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

• What is the purpose of transitivity?

Chaining together separate subtyping rules for records!

S-RecordPermutation
{\{l_i : T_{i}^{i\in1...n}\}} is a permutation of {\{k_j : U_{j}^{j\in1...n}\}}
{\{l_i : T_{i}^{i\in1...n}\}} \prec {\{k_j : U_{j}^{j\in1...n}\}}

S-RecordElements
for each \(i\) \quad T_{i} \prec U_{i}
{\{l_i : T_{i}^{i\in1...n}\}} \prec {\{l_i : U_{i}^{i\in1...n}\}}

S-RecordNewFields
\{l_i : T_{i}^{i\in1...n+k}\} \prec {\{l_i : T_{i}^{i\in1...n}\}}
Algorithmic subtyping

- Replace all previous rules by a single rule.

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

- Correctness / completeness of new rule can be shown.

- Maintain extra rule for function types.

\[
\text{S-Function} \\
T_1 \preceq T_2 \quad U_1 \preceq U_2 \\
T_2 \rightarrow U_1 \preceq T_1 \rightarrow U_2
\]
Algorithmic subtyping

- The subsumption rule is still not syntax-directed.
- The rule is essentially used in function application.
- Express subsumption through an extra premise.

\[
\text{T-Application} \\
\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : V \quad V <: U \\
\hline
\Gamma \vdash t \ u : T
\]

- Retire subsumption rule.
• **Summary:** Lambdas with somewhat sexy types
  ✦ Done: ∀,∃, <:, ...
  ✦ Not done: μ, ...

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

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