x = 1

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.

let x = 1 in ...

x(1).

!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)
[Pierce] B.C. Pierce: Types and Programming Languages, MIT Press, 2002

Polymorphism -- Why?

- What's the identity function?
- In the simple typed lambda calculus, this depends on the type!
- Examples
  - $\lambda x{:}bool.\ x$
  - $\lambda x{:}nat.\ x$
  - $\lambda x{:}bool \rightarrow bool.\ x$
  - $\lambda x{:}bool \rightarrow nat.\ x$
  - ...

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")**
  - ✦ Existential types (''exists as opposed to for all'')
  - ✦ Bounded polymorphism (''subtypes'')
  - ✦ Ad-hoc polymorphism (''some types'')

System F -- Syntax

$$
\begin{aligned}
t &::= x \mid v \mid t\ t \mid t[T] \\
v &::= \lambda x : T . t \mid \Lambda X . t \\
T &::= X \mid T \to T \mid \forall X . T
\end{aligned}
$$

Type application

Type abstraction

Polymorphic type

System F -- Typing rules

Type variables are in the context

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T.u : T \to U}$$

Type variables are subject to alpha conversion.

T-Application

$$\frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

T-TypeAbstraction

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

T-TypeApplication

$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[T_1] : [T_1/X]T}$$

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

System F -- Evaluation rules

E-AppFun

$$\frac{t_1 \longrightarrow t_1{}'}{t_1 \; t_2 \longrightarrow t_1{}' \; t_2}$$

E-AppArg

$$\frac{t \longrightarrow t'}{v \; t \longrightarrow v \; t'}$$

E-AppAbs

$$(\lambda x\!:\!T.t) \; v \longrightarrow [v/x]t$$

E-TypeApp

$$\frac{t_1 \longrightarrow t_1{}'}{t_1[T] \longrightarrow t_1{}'[T]}$$

E-TypeAppAbs

$$(\Lambda X.t)[T] \longrightarrow [T/X]t$$

Examples

Term	Type
$id = \Lambda X.\lambda x:X.x$	$: \forall X.X \to X$
$id[bool]$	$: bool \to bool$
$id[bool]\ true$	$: bool$
$id\ true$	*type error*

There is no inference of type arguments at this point.

The doubling function

$$double = \Lambda X.\lambda f : X \to X.\lambda x : X.f\ (f\ x)$$

- Instantiated with *nat*

  $double\_nat = double\ [nat]$

  $: (nat \to nat) \to nat \to nat$

- Instantiated with $nat \to nat$

  $double\_nat\_arrow\_nat = double\ [nat \to nat]$

  $: ((nat \to nat) \to nat \to nat) \to (nat \to nat) \to nat \to nat$

- Invoking *double*

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

Functions on polymorphic functions

- 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:

  pairid : (*Bool*, *String*)

  pairid = (*id true*, *id* "true")

  Type application left implicit.

- Abstract over id:

  Argument must be polymorphic!

  pairapply : $(\forall X.\, X \to X) \to$ (*Bool*, *String*)

  pairapply = $\lambda f : \forall X.\, X \to X.$ (*f true*, *f* "true")

Self application

- Not typeable in the simply-typed lambda calculus

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

- Typeable in System F

  $\boldsymbol{selfapp : (\forall X.X \to X) \to (\forall X.X \to X)}$

  $selfapp = \lambda x : \forall X.X \to X.x\ [\forall X.X \to X]\ x$

The fix operator (Y)

- Not typeable in the simply-typed lambda calculus
  - ✦ Extension required

$$\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \texttt{fix}\ t : T}$$

- Typeable in System F.

  $\boldsymbol{fix : \forall X.(X \rightarrow X) \rightarrow X}$

- Encodeable in System F with recursive types.

  fix = ?

  See [TAPL]

Lists in System F

- Types of list operations
  $nil : \forall X.\ List\ X$
  $cons : \forall X. X \to List\ X \to List\ X$
  $isnil : \forall X. List\ X \to bool$
  $head : \forall X. List\ X \to X$
  $tail : \forall X. List\ X \to List\ X$

No new syntax needed!

- 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 "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.

We used this generality for **selfapp**.

Not covered by this lecture

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

Polymorphism

- Kinds of polymorphism
  - ✦ Parametric polymorphism ("all types")
  - ✦ **Existential types ("exists as opposed to for all")**
  - ✦ 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.

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 ***information hiding (encapsulation)***.

Overview

- Syntax of types: $T ::= \cdots \mid \{\exists X, T\}$
- Normal forms: $v ::= \cdots \mid \{\, {*T}, v \,\}$
- Terms: $t ::= \cdots \mid \{\, {*T}, t \,\} \text{ as } T$

  $\mid \text{let } \{X, x\} = t \text{ in } t$

Hidden type

Package (existential)

Packing (hiding)

Unpacking

∀ 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 "*".)

∀ 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

Type before packaging:
{ a : nat, b : nat → nat }

- Consider the following package:

  $p = \{{*}nat, \{a = 1, b = \lambda x{:}nat.\ pred\ x\}\}$

- The type system makes sure that *nat* is ***inaccessible*** from outside.
- Multiple types make sense for the package:
  - $\{\ \exists X, \{a{:}X, b{:}X \rightarrow X\}\ \}$
  - $\{\ \exists X, \{a{:}X, b{:}X \rightarrow nat\}\ \}$

Hence, the programmer must provide an annotation upon construction.

Different annotations for the same packaged value

- $p = \{{*}nat, \{a = 1, b = \lambda x{:}nat.\ pred\ x\}\}$ ***as*** $\{\exists X, \{a{:}X, b{:}X \rightarrow X\}\}$

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

- $p' = \{{*}nat, \{a = 1, b = \lambda x{:}nat.\ pred\ x\}\}$ ***as*** $\{\exists X, \{a{:}X, b{:}X \rightarrow nat\}\}$

  p' has type: $\{\ \exists X, \{a{:}X, b{:}X \rightarrow nat\}\}$

Same existential type with different representation types

- $p1 = \{{*}nat, \{a = 1, b = \lambda x{:}nat.\ iszero\ x\}\}$

  $as\ \ \{\exists X,\ \{a{:}X,\ b{:}X \rightarrow bool\}\}$

- $p2 = \{{*}bool, \{a = false, b = \lambda x{:}bool.\ if\ x\ then\ false\ else\ true\}\}$

  $as\ \ \{\exists X,\ \{a{:}X,\ b{:}X \rightarrow bool\}\}$

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

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) $\rightarrow^*$ true : bool

Effective information hiding

- The representation type must remain abstract.

  t = {**nat*, {*a* = *1*, *b* = λ*x:nat. iszero x*} *as* {∃ *X*, {*a:X*, *b:X* → *bool*}}}

  let {*X,x*} = *t in 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* (λ *y:X. x.b y*) *x.a* →* *false : bool*

Typing rules

T-PackExistential

$$\frac{\Gamma \vdash t : [U/X]T}{\Gamma \vdash \{*U, t\} \texttt{ as } \{\exists X, T\} : \{\exists X, T\}}$$

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

T-UnpackExistential

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \qquad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } \{X, x\} = t_1 \texttt{ in } t_2 : T_2}$$

Only expose abstract type of existential!

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

Evaluation rules

E-Pack

$$\frac{t \longrightarrow t'}{\{*T, t\} \texttt{ as } U \longrightarrow \{*T, t'\} \texttt{ as } U}$$

E-Unpack

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let } \{X, x\} = t_1 \texttt{ in } t_2 \longrightarrow \texttt{let } \{X, x\} = t_1' \texttt{ in } t_2}$$

E-UnpackPack

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

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

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

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*.

Subtype preserves behavior.

Subtype preserves type safety.

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

Why subtyping

- Function in near-to-C:

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

- Function application in near-to-C:

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

- Intuitively, it is safe to pass **k**.
  Subtyping allows it.

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

Subsumption
(Subsititutability of supertypes by subtypes)

- Typing rule:

$$\frac{\Gamma \vdash t : U \qquad U <: T}{\Gamma \vdash t : T}$$

- 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.

S-RecordPermutation

$$\frac{\{l_i : T_i{}^{i \in 1 \ldots n}\} \text{ is a permutation of } \{k_j : U_j{}^{j \in 1 \ldots n}\}}{\{l_i : T_i{}^{i \in 1 \ldots n}\} <: \{k_j : U_j{}^{j \in 1 \ldots n}\}}$$

- Example:

$$\{\text{key} : \texttt{bool}, \text{value} : \texttt{int}\} <: \{\text{value} : \texttt{int}, \text{key} : \texttt{bool}\}$$

Subtyping for records

- We can always add new fields in the end.

S-RecordNewFields

$$\{l_i : T_i{}^{i \in 1 \ldots n+k}\} <: \{l_i : T_i{}^{i \in 1 \ldots n}\}$$

- Example:

$$\{\mathsf{key} : \mathtt{bool}, \mathsf{value} : \mathtt{int}, \mathsf{map} : \mathtt{int} \rightarrow \mathtt{int}\} <: \{\mathsf{key} : \mathtt{bool}, \mathsf{value} : \mathtt{int}\}$$

Subtyping for records

- We can subject the fields to subtyping.

S-RecordElements

$$\frac{\text{for each } i \qquad T_i <: U_i}{\{l_i : T_i{}^{i \in 1\ldots n}\} <: \{l_i : U_i{}^{i \in 1\ldots n}\}}$$

- Example:

$$\{\text{field1} : \texttt{bool}, \text{field2} : \{\text{val} : \texttt{bool}\}\} <: \{\text{field1} : \texttt{bool}, \text{field2} : \{\}\}$$

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

General rules for subtyping

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

Optional material:
not covered in the lecture

General rules for subtyping

- **Reflexivity** $T <: T$

- **Transitivity** $$\frac{T <: U \qquad U <: V}{T <: V}$$

- Example

  Prove that $\{a : \texttt{bool}, b : \texttt{int}, c : \{l : \texttt{int}\}\} <: \{c : \{\}\}$

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*)

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

General rules for subtyping: **Subtyping of functions**

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

$$\frac{T_2 <: T_1 \qquad U_2 <: U_1}{T_1 \to U_2 <: T_2 \to U_1}$$

General rules for subtyping: **Supertype of everything**

- *T ::= ... | top*
  - *The most general type*
  - *The supertype of all types*

$$T <: top$$

Remember type annotation?

- Syntax:

  t ::= ... | t 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:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t : U} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash t : T}}{\Gamma \vdash t \text{ as } T : T}$$

- Example:

$$(\lambda x{:}\mathtt{bool}.\{a = x, b = \mathtt{false}\})\ \mathtt{true}\ \mathtt{as}\ \{a : \mathtt{bool}\}$$

General rules for subtyping: **Annotation as down-casting**

- Typing rule:

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

Potentially too liberal

- Evaluation rules:

$$\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T}$$

$$\frac{\vdash v : T}{v \text{ as } T \rightarrow v}$$

Runtime type check

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

Algorithmic subtyping

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]

Optional material: not covered in the lecture

We violate this definition!

Typing rules so far

T-Record

$$\frac{\text{for each } i,\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i{}^{i \in 1 \ldots n}\} : \{l_i : T_i{}^{i \in 1 \ldots n}\}}$$

T-Projection

$$\frac{\Gamma \vdash t : \{l_i : T_i{}^{i \in 1 \ldots n}\}}{\Gamma \vdash t.l_j : T_j}$$

T-Subsumption

$$\frac{\Gamma \vdash t : U \qquad U <: T}{\Gamma \vdash t : T}$$

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T.u : T \to U}$$

T-Application

$$\frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

T-True

$\vdash \texttt{true} : \texttt{bool}$

T-False

$\vdash \texttt{false} : \texttt{bool}$

Violation of syntax direction

- Consider an application:

  $t\ u$ where t of type $U \to V$ and u 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

$$\frac{\Gamma \vdash t : U \qquad U <: T}{\Gamma \vdash t : T}$$

- 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

$$\frac{T <: U \qquad U <: V}{T <: V}$$

- U does not appear in conclusion.

  Thus, to show $T <: V$, we need to guess a U.

- For instance, try to show the following:

$$\{y : \mathtt{int}, x : \mathtt{int}\} <: \{x : \mathtt{int}\}$$

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

Analysis of transitivity

- What is the purpose of transitivity?

  Chaining together separate subtyping rules for records!

S-RecordPermutation

$$\frac{\{l_i : T_i{}^{i \in 1 \ldots n}\} \text{ is a permutation of } \{k_j : U_j{}^{j \in 1 \ldots n}\}}{\{l_i : T_i{}^{i \in 1 \ldots n}\} <: \{k_j : U_j{}^{j \in 1 \ldots n}\}}$$

S-RecordElements

$$\frac{\text{for each } i \qquad T_i <: U_i}{\{l_i : T_i{}^{i \in 1 \ldots n}\} <: \{l_i : U_i{}^{i \in 1 \ldots n}\}}$$

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.

S-Record

$$\frac{\{l_i{}^{i \in 1 \ldots n}\} \subseteq \{k_j{}^{j \in 1 \ldots m}\} \qquad l_i = k_j \text{ implies } U_i <: T_j}{\{k_j : U_j{}^{i \in 1 \ldots m}\} <: \{l_i : T_i{}^{i \in 1 \ldots n}\}}$$

- Correctness / completeness of new rule can be shown.

- Maintain extra rule for function types.

S-Function

$$\frac{T_1 <: T_2 \qquad U_1 <: U_2}{T_2 \rightarrow U_1 <: T_1 \rightarrow U_2}$$

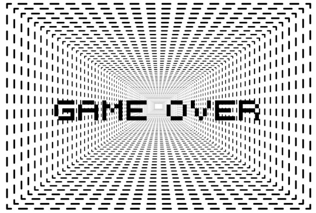
Algorithmic subtyping

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

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \qquad \Gamma \vdash u : V \qquad V <: U}{\Gamma \vdash t\ u : T}$$

- Retire subsumption rule.



- **Summary**: Lambdas with somewhat sexy types
  - *Done:* $\forall$,$\exists$, $<:$, *...*
  - *Not done:* μ, *...*
- **Prepping**: *"Types and Programming Languages"*
  - *Chapters 15, 16, 22, 23, 24*
- **Outlook**:
  - Process calculi
  - Object calculi
  - More paradigms