CS 4110

Programming Languages & Logics

Lecture 22 Parametric Polymorphism

Roadmap

We've extended a simple type system for the λ -calculus with support for a few interesting language constructs. But the "power" of the underlying type system has remained more or less the same.

Today, we'll develop a far more fundamental change to the simply-typed λ -calculus: *parametric polymorphism*, the concept at the heart of OCaml's type system and underlying generics in Java and similar languages.

Polymorphism

Polymorphism generally falls into one of three broad varieties.

Polymorphism

Polymorphism generally falls into one of three broad varieties.

• Subtype polymorphism allows values of type τ to masquerade as values of type τ' , provided that τ is a subtype of τ' .

Polymorphism generally falls into one of three broad varieties.

- Subtype polymorphism allows values of type τ to masquerade as values of type τ' , provided that τ is a subtype of τ' .
- *Ad-hoc polymorphism*, also called overloading, allows the same function name to be used with functions that take different types of parameters.

Polymorphism generally falls into one of three broad varieties.

- Subtype polymorphism allows values of type τ to masquerade as values of type τ' , provided that τ is a subtype of τ' .
- *Ad-hoc polymorphism*, also called overloading, allows the same function name to be used with functions that take different types of parameters.
- *Parametric polymorphism* refers to code that is written without knowledge of the actual type of the arguments; the code is parametric in the type of the parameters.

Consider a "doubling" function that takes a function *f*, and an integer *x*, applies *f* to *x*, and then applies *f* to the result:

Consider a "doubling" function that takes a function *f*, and an integer *x*, applies *f* to *x*, and then applies *f* to the result:

doubleInt $\triangleq \lambda f$: int \rightarrow int. λx : int. f(fx)

•

Consider a "doubling" function that takes a function *f*, and an integer *x*, applies *f* to *x*, and then applies *f* to the result:

```
doubleInt \triangleq \lambda f: int \rightarrow int. \lambda x: int. f(fx)
```

Now suppose we want the same function for Booleans, or functions...

doubleBool $\triangleq \lambda f$: **bool** \rightarrow **bool**. λx : **bool**. f(fx)doubleFn $\triangleq \lambda f$: (**int** \rightarrow **int**) \rightarrow (**int** \rightarrow **int**). λx : **int** \rightarrow **int**. f(fx)

Definition (Abstraction Principle)

Every major piece of functionality in a program should be implemented in just one place in the code. When similar functionality is provided by distinct pieces of code, the two should be combined into one by abstracting out the varying parts.

Definition (Abstraction Principle)

Every major piece of functionality in a program should be implemented in just one place in the code. When similar functionality is provided by distinct pieces of code, the two should be combined into one by abstracting out the varying parts.

In the doubling functions, the varying parts are the types.

Definition (Abstraction Principle)

Every major piece of functionality in a program should be implemented in just one place in the code. When similar functionality is provided by distinct pieces of code, the two should be combined into one by abstracting out the varying parts.

In the doubling functions, the varying parts are the types.

We need a way to abstract out the type of the doubling operation, and later instantiate it with different concrete types.

Invented independently in 1972–1974 by a computer scientist John Reynolds and a logician Jean-Yves Girard (who called it System F).

Key feature: Function abstraction and application, just like in λ -calculus terms, but *at the type level!*

Notation:

- *NX. e*: type abstraction
- *e*[τ]: type application

Example: $\Lambda X. \lambda x: X. x$

$$e ::= n \mid x \mid \lambda x : \tau. e \mid e_1 e_2$$
$$v ::= n \mid \lambda x : \tau. e$$

$$e ::= n \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \Lambda X . e$$
$$v ::= n \mid \lambda x : \tau . e$$

$$e ::= n \mid x \mid \lambda x : \tau \cdot e \mid e_1 e_2 \mid \Lambda X \cdot e \mid e [\tau]$$
$$v ::= n \mid \lambda x : \tau \cdot e$$

$$e ::= n \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e[\tau]$$
$$v ::= n \mid \lambda x : \tau. e \mid \Lambda X. e$$

Syntax

$$e ::= n \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e[\tau]$$
$$v ::= n \mid \lambda x : \tau. e \mid \Lambda X. e$$

Dynamic Semantics

$$\mathsf{E} ::= [\cdot] \mid \mathsf{E} \mathsf{e} \mid \mathsf{v} \mathsf{E} \mid \mathsf{E} [\tau]$$

Syntax

$$e ::= n \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e[\tau]$$
$$v ::= n \mid \lambda x : \tau. e \mid \Lambda X. e$$

Dynamic Semantics

$$\mathsf{E} ::= [\cdot] \mid \mathsf{E} \mathsf{e} \mid \mathsf{v} \mathsf{E} \mid \mathsf{E} [\tau]$$

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \overline{(\lambda x : \tau. e) v \to e\{v/x\}}$$

Syntax

$$e ::= n \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e[\tau]$$
$$v ::= n \mid \lambda x : \tau. e \mid \Lambda X. e$$

Dynamic Semantics

$$\mathsf{E} ::= [\cdot] \mid \mathsf{E} \mathsf{e} \mid \mathsf{v} \mathsf{E} \mid \mathsf{E} [\tau]$$

$$\frac{\mathbf{e} \to \mathbf{e}'}{\mathbf{E}[\mathbf{e}] \to \mathbf{E}[\mathbf{e}']} \qquad \overline{(\lambda x : \tau . \mathbf{e}) \, \mathbf{v} \to \mathbf{e}\{\mathbf{v}/x\}} \qquad \overline{(\Lambda X . \mathbf{e}) \, [\tau] \to \mathbf{e}\{\tau/X\}}$$

Type Syntax

 $\tau ::= \operatorname{int} | \tau_1 \to \tau_2$

Type Syntax

 $\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid X$

Type Syntax

$$au ::= \mathsf{int} \mid au_1 o au_2 \mid X \mid orall X. au$$

Type Syntax

$$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X. \ \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

- Γ a mapping from variables to types
- Δ a set of types in scope
- e an expression
- τ a type

Type Syntax

$$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X. \ \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

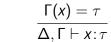
- Γ a mapping from variables to types
- Δ a set of types in scope
- e an expression
- au a type

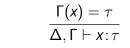
Type Well-Formedness: $\Delta \vdash \tau$ ok

- Δ a set of types in scope
- au a type

$\overline{\Delta, \Gamma \vdash n: int}$

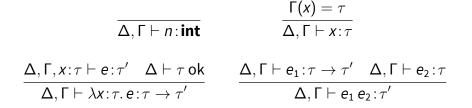
 $\Delta, \overline{\Gamma \vdash n}$: int

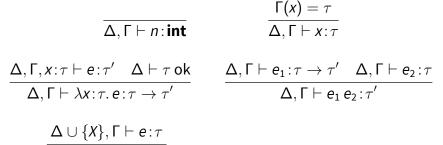




 $\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}$

 $\Delta, \Gamma \vdash n$: int





 $\Delta, \Gamma \vdash \Lambda X. e: \forall X. \tau$

$$\frac{\overline{\Delta}, \Gamma \vdash n: \mathbf{int}}{\overline{\Delta}, \Gamma \vdash n: \mathbf{int}} \qquad \frac{\overline{\Delta}, \Gamma \vdash x: \tau}{\overline{\Delta}, \Gamma \vdash x: \tau}$$

$$\frac{\underline{\Delta}, \Gamma, x: \tau \vdash e: \tau' \quad \Delta \vdash \tau \text{ ok}}{\overline{\Delta}, \Gamma \vdash \lambda x: \tau. e: \tau \rightarrow \tau'} \qquad \frac{\underline{\Delta}, \Gamma \vdash e_1: \tau \rightarrow \tau' \quad \Delta, \Gamma \vdash e_2: \tau}{\overline{\Delta}, \Gamma \vdash e_1 e_2: \tau'}$$

$$\frac{\underline{\Delta} \cup \{X\}, \Gamma \vdash e: \tau}{\overline{\Delta}, \Gamma \vdash \Lambda X. e: \forall X. \tau} \qquad \frac{\underline{\Delta}, \Gamma \vdash e: \forall X. \tau' \quad \Delta \vdash \tau \text{ ok}}{\overline{\Delta}, \Gamma \vdash e [\tau]: \tau' \{\tau/X\}}$$

$\frac{\textit{X} \in \Delta}{\Delta \vdash \textit{X} \, \mathsf{ok}}$

$\frac{\mathit{X} \in \Delta}{\Delta \vdash \mathit{X} \, \mathsf{ok}}$

$\Delta \vdash \textbf{int} \ ok$

$\frac{\mathit{X} \in \Delta}{\Delta \vdash \mathit{X} \, \mathsf{ok}}$

$\Delta \vdash \textbf{int} \ \mathsf{ok}$

$$\frac{\Delta \vdash \tau_1 \, \mathsf{ok} \quad \Delta \vdash \tau_2 \, \mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2 \, \mathsf{ok}}$$

$\frac{\mathit{X} \in \Delta}{\Delta \vdash \mathit{X} \, \mathsf{ok}}$

$\Delta \vdash \textbf{int} \ \mathsf{ok}$

$$\frac{\Delta \vdash \tau_1 \text{ ok } \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok}}$$
$$\frac{\Delta \cup \{X\} \vdash \tau \text{ ok}}{\Delta \vdash \forall X. \tau \text{ ok}}$$

Example: Doubling Redux

Let's consider the doubling operation again.

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

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

We can write a polymorphic doubling operation as

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

The type of this expression is: $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

We can write a polymorphic doubling operation as

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

The type of this expression is: $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

We can instantiate this on a type, and provide arguments:

We can write a polymorphic doubling operation as

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

The type of this expression is: $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

We can instantiate this on a type, and provide arguments:

double [int] (λn : int. n + 1) 7

We can write a polymorphic doubling operation as

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

The type of this expression is: $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

We can instantiate this on a type, and provide arguments:

double [int] $(\lambda n : \text{int. } n + 1)$ 7 $\rightarrow (\lambda f : \text{int} \rightarrow \text{int. } \lambda x : \text{int. } f(fx)) (\lambda n : \text{int. } n + 1)$ 7

We can write a polymorphic doubling operation as

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

The type of this expression is: $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

We can instantiate this on a type, and provide arguments:

double [int] (
$$\lambda n$$
: int. $n + 1$) 7
 \rightarrow (λf : int \rightarrow int. λx : int. $f(fx)$) (λn : int. $n + 1$) 7
 \rightarrow^* 9

Example: Self Application

Recall that in the simply-typed λ -calculus, we had no way of typing the expression $\lambda x. x x$.

Recall that in the simply-typed λ -calculus, we had no way of typing the expression $\lambda x. x x$.

In the polymorphic λ -calculus, however, we can type this expression using a polymorphic type:

Recall that in the simply-typed λ -calculus, we had no way of typing the expression $\lambda x. x x$.

In the polymorphic λ -calculus, however, we can type this expression using a polymorphic type:

$$\vdash \quad \lambda x : \forall X. X \to X. x \left[\forall X. X \to X \right] x : \left(\forall X. X \to X \right) \to \left(\forall X. X \to X \right)$$

(However, all expressions in polymorphic λ -calculus still halt. There is no way to give a type to the *self-application* of this term.) We can encode products in polymorphic λ -calculus without adding any additional types!

$$\tau_1 \times \tau_2 \triangleq \forall R. (\tau_1 \to \tau_2 \to R) \to R$$

We can encode products in polymorphic λ -calculus without adding any additional types!

$$\tau_1 \times \tau_2 \triangleq \forall R. (\tau_1 \to \tau_2 \to R) \to R$$

(\cdots, \cdots) \u00e9 \Lambda T_1. \Lambda T_2. \lambda v_1 : T_1 \lambda v_2 : T_2. \Lambda R. \lambda p : (T_1 \u2013 T_2 \u2013 R). p v_1 v_2

We can encode products in polymorphic λ -calculus without adding any additional types!

$$\tau_{1} \times \tau_{2} \triangleq \forall R. (\tau_{1} \to \tau_{2} \to R) \to R$$

(\cdot, \cdot) \u00e9 \u00e9 \u00e7_{1}. \u00e8 T_{2}. \u00e3 v_{1} : T_{1} \u00e3 v_{2} : T_{2}. \u00e9 R. \u00e3 p : (T_{1} \u00e3 T_{2} \u00e3 R). p v_{1} v_{2}
$$\pi_{1} \triangleq \u00e8 T_{1}. \u00e9 T_{2}. \u00e3 v : T_{1} \u00e8 T_{2}. v [T_{1}] (\u00e3 x : T_{1}. \u00e3 y : T_{2}. x)$$

We can encode products in polymorphic λ -calculus without adding any additional types!

$$\tau_{1} \times \tau_{2} \triangleq \forall R. (\tau_{1} \rightarrow \tau_{2} \rightarrow R) \rightarrow R$$

$$(\cdot, \cdot) \triangleq \Lambda T_{1}. \Lambda T_{2}. \lambda v_{1} : T_{1} \lambda v_{2} : T_{2}. \Lambda R. \lambda p : (T_{1} \rightarrow T_{2} \rightarrow R). p v_{1} v_{2}$$

$$\pi_{1} \triangleq \Lambda T_{1}. \Lambda T_{2}. \lambda v : T_{1} \times T_{2}. v [T_{1}] (\lambda x : T_{1}. \lambda y : T_{2}. x)$$

$$\pi_{2} \triangleq \Lambda T_{1}. \Lambda T_{2}. \lambda v : T_{1} \times T_{2}. v [T_{2}] (\lambda x : T_{1}. \lambda y : T_{2}. y)$$

$$\tau_1 + \tau_2 \triangleq \forall R. (\tau_1 \to R) \to (\tau_2 \to R) \to R$$

$$\tau_1 + \tau_2 \triangleq \forall R.(\tau_1 \to R) \to (\tau_2 \to R) \to R$$

inl $\triangleq \Lambda T_1. \Lambda T_2. \lambda v_1 : T_1. \Lambda R. \lambda b_1 : T_1 \to R. \lambda b_2 : T_2 \to R. b_1 v_1$

$$\tau_{1} + \tau_{2} \triangleq \forall R.(\tau_{1} \rightarrow R) \rightarrow (\tau_{2} \rightarrow R) \rightarrow R$$

inl
$$\triangleq \Lambda T_{1}. \Lambda T_{2}. \lambda v_{1} : T_{1}. \Lambda R. \lambda b_{1} : T_{1} \rightarrow R. \lambda b_{2} : T_{2} \rightarrow R. b_{1} v_{1}$$

inr
$$\triangleq \Lambda T_{1}. \Lambda T_{2}. \lambda v_{2} : T_{2}. \Lambda R. \lambda b_{1} : T_{1} \rightarrow R. \lambda b_{2} : T_{2} \rightarrow R. b_{2} v_{2}$$

$$\tau_{1} + \tau_{2} \triangleq \forall R.(\tau_{1} \rightarrow R) \rightarrow (\tau_{2} \rightarrow R) \rightarrow R$$

inl
$$\triangleq \Lambda T_{1}.\Lambda T_{2}.\lambda v_{1}: T_{1}.\Lambda R.\lambda b_{1}: T_{1} \rightarrow R.\lambda b_{2}: T_{2} \rightarrow R.b_{1}v_{1}$$

inr
$$\triangleq \Lambda T_{1}.\Lambda T_{2}.\lambda v_{2}: T_{2}.\Lambda R.\lambda b_{1}: T_{1} \rightarrow R.\lambda b_{2}: T_{2} \rightarrow R.b_{2}v_{2}$$

case
$$\triangleq \Lambda T_{1}.\Lambda T_{2}.\Lambda R.\lambda v: T_{1} + T_{2}.\lambda b_{1}: T_{1} \rightarrow R.\lambda b_{2}: T_{2} \rightarrow R.$$

$$v[R] b_{1} b_{2}$$

Type Erasure

The semantics presented above explicitly passes type but in an implementation, one often wants to eliminate types for efficiency.

Type Erasure

The semantics presented above explicitly passes type but in an implementation, one often wants to eliminate types for efficiency.

$$erase(x) = x$$

$$erase(x) = x$$

 $erase(\lambda x: \tau. e) = \lambda x. erase(e)$

$$erase(x) = x$$

 $erase(\lambda x : \tau. e) = \lambda x. erase(e)$
 $erase(e_1 e_2) = erase(e_1) erase(e_2)$

$$erase(x) = x$$

$$erase(\lambda x: \tau. e) = \lambda x. erase(e)$$

$$erase(e_1 e_2) = erase(e_1) erase(e_2)$$

$$erase(\Lambda X. e) = \lambda z. erase(e)$$
 where z is fresh for e

$$erase(x) = x$$

$$erase(\lambda x:\tau.e) = \lambda x. erase(e)$$

$$erase(e_1 e_2) = erase(e_1) erase(e_2)$$

$$erase(\Lambda X.e) = \lambda z. erase(e)$$
 where z is fresh for e

$$erase(e[\tau]) = erase(e) (\lambda x. x)$$

Type Erasure

The following theorem states this translation is adequate:

Theorem (Erasure Adequacy)

For all expressions e and e', we have $e \rightarrow e'$ iff $erase(e) \rightarrow erase(e')$.

Type Inference

The type inference (or "type reconstruction") problem asks whether, for a given untyped λ -calculus expression e' there exists a well-typed System F expression e such that erase(e) = e'

Type Inference

The type inference (or "type reconstruction") problem asks whether, for a given untyped λ -calculus expression e' there exists a well-typed System F expression e such that erase(e) = e'

It was shown to be undecidable by Wells in 1994.

Type Inference

The type inference (or "type reconstruction") problem asks whether, for a given untyped λ -calculus expression e' there exists a well-typed System F expression e such that erase(e) = e'

It was shown to be undecidable by Wells in 1994.

See Chapter 23 of Pierce for further discussion, as well as restrictions for which type reconstruction is decidable.