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

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

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.

Example

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

Example

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

$$\text{doubleInt} \triangleq \lambda f\!:\!\mathbf{int} \rightarrow \mathbf{int}.\ \lambda x\!:\!\mathbf{int}.\ f\,(f\,x)$$

Example

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

$$\text{doubleInt} \triangleq \lambda f\!:\!\mathbf{int} \to \mathbf{int}.\ \lambda x\!:\!\mathbf{int}.\ f\,(f\,x)$$

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

$$\begin{aligned}
\text{doubleBool} &\triangleq \lambda f\!:\!\mathbf{bool} \to \mathbf{bool}.\ \lambda x\!:\!\mathbf{bool}.\ f\,(f\,x) \\
\text{doubleFn} &\triangleq \lambda f\!:\!(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int}).\ \lambda x\!:\!\mathbf{int} \to \mathbf{int}.\ f\,(f\,x) \\
&\vdots
\end{aligned}$$

Abstraction

These examples on the preceding slides violate a fundamental principle of software engineering:

Abstraction

These examples on the preceding slides violate a fundamental principle of software engineering:

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.

Abstraction

These examples on the preceding slides violate a fundamental principle of software engineering:

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.

Abstraction

These examples on the preceding slides violate a fundamental principle of software engineering:

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.

Polymorphic λ-Calculus

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:

- $\Lambda X.\, e$: type abstraction
- $e[\tau]$: type application

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

Polymorphic λ-Calculus

Syntax

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

Polymorphic λ-Calculus

Syntax

$$e ::= n \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \Lambda X.\, e$$

$$v ::= n \mid \lambda x{:}\tau.\, e$$

Polymorphic λ-Calculus

Syntax

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

Polymorphic λ-Calculus

Syntax

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

Polymorphic λ-Calculus

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

$$E ::= [\cdot] \mid E\, e \mid v\, E \mid E\,[\tau]$$

Polymorphic λ-Calculus

Syntax

$$
\begin{aligned}
e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda X.\, e \mid e\,[\tau] \\
v &::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda X.\, e
\end{aligned}
$$

Dynamic Semantics

$$
E ::= [\cdot] \mid E\, e \mid v\, E \mid E\,[\tau]
$$

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

Polymorphic λ-Calculus

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

$$E ::= [\cdot] \mid E\, e \mid v\, E \mid E\, [\tau]$$

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

Typing Judgment

Type Syntax

$$\tau ::= \textbf{int} \mid \tau_1 \rightarrow \tau_2$$

Typing Judgment

Type Syntax

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

Typing Judgment

Type Syntax

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

Typing Judgment

Type Syntax

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

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

Typing Judgment

Type Syntax

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

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

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

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

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

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \mathbf{int}}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

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

$$\frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

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

$$\frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

$$\frac{\Delta \cup \{X\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda X.\, e : \forall X.\, \tau}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

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

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

Type Well-Formedness

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

Type Well-Formedness

$$\frac{X \in \Delta}{\Delta \vdash X \; \mathsf{ok}}$$

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

Type Well-Formedness

$$\frac{X \in \Delta}{\Delta \vdash X \text{ ok}}$$

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

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

Type Well-Formedness

$$\frac{X \in \Delta}{\Delta \vdash X \text{ ok}}$$

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

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

$$\frac{\Delta \cup \{X\} \vdash \tau \text{ ok}}{\Delta \vdash \forall X.\, \tau \text{ ok}}$$

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

$$\text{double} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\,(f\,x).$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

$$\text{double} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\,(f\,x).$$

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

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

$$\text{double} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\,(f\,x).$$

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

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

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

$$\text{double} \triangleq \Lambda X.\, \lambda f\!:\!X \to X.\, \lambda x\!:\!X.\, f\,(f\,x).$$

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

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

$$\text{double}\ [\mathbf{int}]\ (\lambda n\!:\!\mathbf{int}.\, n + 1)\ 7$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

$$\text{double} \triangleq \Lambda X.\, \lambda f\!:\!X \to X.\, \lambda x\!:\!X.\, f\,(f\,x).$$

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

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

$$\begin{array}{ll} & \text{double}\, [\textbf{int}]\, (\lambda n\!:\!\textbf{int}.\, n+1)\; 7 \\ \to & (\lambda f\!:\!\textbf{int} \to \textbf{int}.\, \lambda x\!:\!\textbf{int}.\, f\,(f\,x))\, (\lambda n\!:\!\textbf{int}.\, n+1)\; 7 \end{array}$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as

$$\text{double} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\,(f\,x).$$

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

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

$$\begin{array}{rl} & \text{double}\,[\textbf{int}]\,(\lambda n{:}\textbf{int}.\, n + 1)\; 7 \\ \to & (\lambda f{:}\textbf{int} \to \textbf{int}.\, \lambda x{:}\textbf{int}.\, f\,(f\,x))\,(\lambda n{:}\textbf{int}.\, n + 1)\; 7 \\ \to^{*} & 9 \end{array}$$

Example: Self Application

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

Example: Self Application

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:

Example: Self Application

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:

$$\vdash \quad \lambda x{:}\forall X.\, X \to X.\, x\, [\forall X.\, X \to X]\, x \,:\, (\forall X.\, X \to X) \to (\forall X.\, X \to X)$$

(However, all expressions in polymorphic λ-calculus still halt. There is no way to give a type to the *self-application* of this term.)

Example: Products

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

The encodings are based on the (untyped) Church encodings:

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

Example: Products

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

The encodings are based on the (untyped) Church encodings:

$$
\begin{aligned}
\tau_1 \times \tau_2 &\triangleq \forall R.\, (\tau_1 \to \tau_2 \to R) \to R \\
(\cdot, \cdot) &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1\; \lambda v_2 : T_2.\, \Lambda R.\, \lambda p : (T_1 \to T_2 \to R).\, p\, v_1\, v_2
\end{aligned}
$$

Example: Products

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

The encodings are based on the (untyped) Church encodings:

$$
\begin{aligned}
\tau_1 \times \tau_2 &\triangleq \forall R.\, (\tau_1 \to \tau_2 \to R) \to R \\
(\cdot, \cdot) &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1\, \lambda v_2 : T_2.\, \Lambda R.\, \lambda p : (T_1 \to T_2 \to R).\, p\, v_1\, v_2 \\
\pi_1 &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v : T_1 \times T_2.\, v\, [T_1]\, (\lambda x : T_1.\, \lambda y : T_2.\, x)
\end{aligned}
$$

Example: Products

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

The encodings are based on the (untyped) Church encodings:

$$
\begin{aligned}
\tau_1 \times \tau_2 &\triangleq \forall R.\, (\tau_1 \to \tau_2 \to R) \to R \\
(\cdot, \cdot) &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1\, \lambda v_2 : T_2.\, \Lambda R.\, \lambda p : (T_1 \to T_2 \to R).\, p\, v_1\, v_2 \\
\pi_1 &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v : T_1 \times T_2.\, v\, [T_1]\, (\lambda x : T_1.\, \lambda y : T_2.\, x) \\
\pi_2 &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v : T_1 \times T_2.\, v\, [T_2]\, (\lambda x : T_1.\, \lambda y : T_2.\, y)
\end{aligned}
$$

Example: Sums

Similarly, we can encode sums in polymorphic λ-calculus without adding any additional types!

Again, the encodings are based on the (untyped) Church encodings:

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

Example: Sums

Similarly, we can encode sums in polymorphic λ-calculus without adding any additional types!

Again, the encodings are based on the (untyped) Church encodings:

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

$$\mathsf{inl} \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_1\, v_1$$

Example: Sums

Similarly, we can encode sums in polymorphic λ-calculus without adding any additional types!

Again, the encodings are based on the (untyped) Church encodings:

$$
\begin{aligned}
\tau_1 + \tau_2 &\triangleq \forall R.(\tau_1 \to R) \to (\tau_2 \to R) \to R \\
\text{inl} &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_1\, v_1 \\
\text{inr} &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_2 : T_2.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_2\, v_2
\end{aligned}
$$

Example: Sums

Similarly, we can encode sums in polymorphic λ-calculus without adding any additional types!

Again, the encodings are based on the (untyped) Church encodings:

$$
\begin{aligned}
\tau_1 + \tau_2 &\triangleq \forall R.(\tau_1 \to R) \to (\tau_2 \to R) \to R \\
\mathsf{inl} &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_1\, v_1 \\
\mathsf{inr} &\triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_2 : T_2.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_2\, v_2 \\
\mathsf{case} &\triangleq \Lambda T_1.\, \Lambda T_2.\, \Lambda R.\, \lambda v : T_1 + T_2.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R. \\
&\quad\; v\, [R]\, b_1\, b_2
\end{aligned}
$$

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.

The following translation “erases” the types from a polymorphic λ-calculus expression.

Type Erasure

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

The following translation "erases" the types from a polymorphic λ-calculus expression.

$$\mathit{erase}(x) \quad = \quad x$$

Type Erasure

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

The following translation "erases" the types from a polymorphic λ-calculus expression.

$$\begin{aligned} \mathit{erase}(x) &= x \\ \mathit{erase}(\lambda x\!:\!\tau.\, e) &= \lambda x.\, \mathit{erase}(e) \end{aligned}$$

Type Erasure

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

The following translation "erases" the types from a polymorphic λ-calculus expression.

$$
\begin{aligned}
\mathit{erase}(x) &= x \\
\mathit{erase}(\lambda x{:}\tau.\, e) &= \lambda x.\, \mathit{erase}(e) \\
\mathit{erase}(e_1\ e_2) &= \mathit{erase}(e_1)\ \mathit{erase}(e_2)
\end{aligned}
$$

Type Erasure

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

The following translation "erases" the types from a polymorphic λ-calculus expression.

$$
\begin{array}{rcll}
\mathit{erase}(x) & = & x & \\
\mathit{erase}(\lambda x\!:\!\tau.\, e) & = & \lambda x.\, \mathit{erase}(e) & \\
\mathit{erase}(e_1\ e_2) & = & \mathit{erase}(e_1)\ \mathit{erase}(e_2) & \\
\mathit{erase}(\Lambda X.\, e) & = & \lambda z.\, \mathit{erase}(e) & \quad \text{where } z \text{ is fresh for } e
\end{array}
$$

Type Erasure

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

The following translation "erases" the types from a polymorphic λ-calculus expression.

$$
\begin{array}{rcll}
\mathit{erase}(x) & = & x & \\
\mathit{erase}(\lambda x\!:\!\tau.\, e) & = & \lambda x.\, \mathit{erase}(e) & \\
\mathit{erase}(e_1\ e_2) & = & \mathit{erase}(e_1)\ \mathit{erase}(e_2) & \\
\mathit{erase}(\Lambda X.\, e) & = & \lambda z.\, \mathit{erase}(e) & \quad \text{where } z \text{ is fresh for } e \\
\mathit{erase}(e\ [\tau]) & = & \mathit{erase}(e)\ (\lambda x.\, x) &
\end{array}
$$

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 $\mathit{erase}(e) \rightarrow \mathit{erase}(e')$.

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 $\mathit{erase}(e) = e'$

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.