Earlier

Saw structural subtyping
  ▶ constraints over record fields
  ▶ propagate constraints to “bigger” types
  ▶ covariance, contravariance

What if code imposes no constraints on some types?

This Time: Parametric Polymorphism

Some code just doesn’t care what types it’s operating over.

You might even say it works *universally*...
This Time: Parametric Polymorphism

Some code just doesn’t care what types it’s operating over.

You might even say it works universally... ???

Before we figure out what that means, a word from a luminary:

Zach Tatlock
CSE 505 Autumn 2017, Lecture 18

Goal: Everybody Wins!

Understand what this interface means and why it matters:

```ocaml
type 'a mylist;
val empty : 'a mylist
val cons : 'a -> 'a mylist -> 'a mylist
val decons : 'a mylist -> (('a * 'a mylist) option)
val length : 'a mylist -> int
val map : ('a -> 'b) -> 'a mylist -> 'b mylist
```

From two perspectives:

1. Client: Code against this specification
2. Library: Implement this specification

Goal: Client Wins!

1. Reusability (at different types!)
   - Different lists with elements of different types
   - New reusable functions outside of library, e.g.:
     ```ocaml
     val twocons: 'a -> 'a -> 'a mylist -> 'a mylist
     ```

MOVIE TIME!
1. Reusability (at different types!)
   ▶ Different lists with elements of different types
   ▶ New reusable functions outside of library, e.g.:
     val twocons: 'a -> 'a -> 'a mylist -> 'a mylist

2. Easier, faster, more reliable than subtyping
   ▶ No downcast to write, run, maybe-fail (cf. Java 1.4 List)

3. Library must “behave the same” for all “type instantiations”!
   ▶ 'a and 'b held abstract from library
   ▶ e.g., suppose foo: 'a list -> int, then
     foo [1;2;3] totally equivalent to foo [(5,4);(7,2);(9,2)]
   ▶ Why? Still true if we have downcasts?
**Goal: Library Wins!**

1. Reusability — all the same reasons client likes it
2. Abstraction of mylist from clients
   - Clients can only assume interface, no implementation details
   - Free to change/optimize hidden details of *a mylist*
   - Clients typechecked knowing only:
     - `there exists some type constructor mylist`
   - Unlike Java/C++ cannot downcast a `t mylist` to, e.g., a pair

**Start Simple**

The mylist interface has a lot going on:

1. Element types *held abstract* from library
2. List type (constructor) *held abstract* from client
3. Reuse of type variables constrains expressions over abstract types
4. Lists need some form of recursive type

We’ll focus on (1) and (3):

- First using a formal language with explicit type abstraction
- Then compare and contrast with ML

**Note:** Much more interesting than "not getting stuck"
Recipe for Extension

1. Add syntax
2. Add semantics
3. Add typing rules
4. Patch up type safety proof

Zach Tatlock
CSE 505 Autumn 2017, Lecture 18

1. Add Syntax

```
e ::= c | x | λx:τ. e | e e | Λα. e
```

1. Add Syntax

```
e ::= c | x | λx:τ. e | e e
```

Summary of new things:
▶ Terms: Type abstraction and type application
▶ Types: Type variables and universal types
▶ Type contexts: what type variables are in scope
1. Add Syntax

\[
e ::= c \mid x \mid \lambda x:\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e[\tau]
\]

\[
\tau ::= \text{int} \mid \tau \rightarrow \tau
\]

Summary of new things:
▶ Terms: Type abstraction and type application
▶ Types: Type variables and universal types
▶ Type contexts: what type variables are in scope
1. Add Syntax

\[
\begin{align*}
e &::= c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e[\tau] \\
\tau &::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \tau \\
v &::= c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e \\
\Gamma &::= \cdot \mid \Gamma, x: \tau \\
\Delta &::= \cdot \mid \Delta, \alpha
\end{align*}
\]

Summary of new things:
- Terms: Type abstraction and type application
- Types: Type variables and universal types
- Type contexts: what type variables are in scope
2. Add Semantics

What is this Λ (big lambda) thing? Informally:

1. Λα. e: a value that takes some τ, plugs it in for α, then runs e
   ▶ type-check e knowing α is some type, but not which type

2. e[τ]: crunch e down to some Λα. e′, plug in τ for α, run e′
   ▶ choice of τ is irrelevant at run-time
   ▶ τ used for type-checking and proof of Preservation

---

Formal, small-step, CBV, left-to-right operational semantics:

Recall: Λα. e is a value

\[ e \rightarrow e' \]
2. Add Semantics

Formal, small-step, CBV, left-to-right operational semantics:

- Recall: \( \Lambda \alpha. e \) is a value

\[
e' \rightarrow e''
\]

Old:

\[
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2} \quad (\lambda x: \tau. e) v \rightarrow e[v/x]
\]

New:

\[
\frac{e \rightarrow e'}{e[\tau] \rightarrow e'[\tau]} \quad (\Lambda \alpha. e)[\tau] \rightarrow e[\tau/\alpha]
\]

Plus now have 3 different kinds of substitution, all defined in straightforward capture-avoiding way:

- \( e_1[e_2/x] \) (old)
2. Add Semantics

Formal, small-step, CBV, left-to-right operational semantics:

- Recall: $\Lambda \alpha. \ e$ is a value
  
  $e \rightarrow e'$

<table>
<thead>
<tr>
<th>Old:</th>
<th>$e_1 \rightarrow e'_1$</th>
<th>$e_2 \rightarrow e'_2$</th>
<th>$(\lambda x : \tau. \ e) \ v \rightarrow e'[v/x]$</th>
</tr>
</thead>
</table>

| New: | $e \rightarrow e'$ | $e[\tau] \rightarrow e'[\tau]$ | $(\Lambda \alpha. \ e)[\tau] \rightarrow e[\tau/\alpha]$ |

Plus now have 3 different kinds of substitution, all defined in straightforward capture-avoiding way:

- $e_1[e_2/x]$ (old)
- $e[\tau'/\alpha]$ (new)

Example

Example (using addition):

$$(\Lambda \alpha. \Lambda \beta. \ \lambda x : \alpha. \ \lambda f : \alpha \rightarrow \beta. \ f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)$$

$$(\Lambda \alpha. \Lambda \beta. \ \lambda x : \alpha. \ \lambda f : \alpha \rightarrow \beta. \ f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)$$

$$(\Lambda \beta. \ \lambda x : \text{int}. \ \lambda f : \text{int} \rightarrow \beta. \ f \ x) \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)$$

Example (using addition):

$$(\Lambda \alpha. \Lambda \beta. \ \lambda x : \alpha. \ \lambda f : \alpha \rightarrow \beta. \ f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)$$

$$(\Lambda \beta. \ \lambda x : \text{int}. \ \lambda f : \text{int} \rightarrow \beta. \ f \ x) \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)$$
Example

Example (using addition):

\[(\Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)\]

\[\rightarrow (\Lambda \beta. \lambda x : \text{int}. \lambda f : \text{int} \rightarrow \beta. f x) \ [\text{int}] \ 3 \ (\lambda y : \text{int} \ y + y)\]

\[\rightarrow (\lambda x : \text{int}. \lambda f : \text{int} \rightarrow \text{int}. f x) \ 3 \ (\lambda y : \text{int} \ y + y)\]

\[\rightarrow (\lambda f : \text{int} \rightarrow \text{int}. f 3) \ (\lambda y : \text{int} \ y + y)\]

\[\rightarrow (\lambda y : \text{int} \ y + y) \ 3\]

\[\rightarrow 3 + 3\]
Example

Example (using addition):
(Λα. Λβ. λx : α. λf : α → β. f x) [int] [int] 3 (λy : int. y + y)

→ (Λβ. λx : int. λf : int → β. f x) [int] 3 (λy : int. y + y)

→ (λx : int. λf : int → int. f x) 3 (λy : int. y + y)

→ (λf : int → int. f 3) (λy : int. y + y)

→ (λy : int. y + y) 3

→ 3 + 3

→ 6

3. Add Typing Rules

Need to be picky about “no free type variables”

- Typing judgment has the form Δ; Γ ⊢ e : τ (whole program · ⊢ e : τ)
- Uses helper judgment Δ ⊢ τ
  - “all free type variables in τ are in Δ”

```
Δ ⊢ τ

α ∈ Δ

Δ ⊢ α
```

3. Add Typing Rules

Need to be picky about “no free type variables”

- Typing judgment has the form Δ; Γ ⊢ e : τ (whole program · ⊢ e : τ)
- Uses helper judgment Δ ⊢ τ
  - “all free type variables in τ are in Δ”

```
Δ ⊢ τ

α ∈ Δ

Δ ⊢ α

Δ ⊢ int
```
3. Add Typing Rules

Need to be picky about “no free type variables”
- Typing judgment has the form $\Delta; \Gamma \vdash e : \tau$
  (whole program $\cdot; \cdot \vdash e : \tau$
- Uses helper judgment $\Delta \vdash \tau$
  - “all free type variables in $\tau$ are in $\Delta$"

$\Delta \vdash \tau$

<table>
<thead>
<tr>
<th>$\alpha \in \Delta$</th>
<th>$\Delta \vdash \alpha$</th>
<th>$\Delta \vdash \tau_1$</th>
<th>$\Delta \vdash \tau_2$</th>
<th>$\Delta \vdash \tau_1 \rightarrow \tau_2$</th>
</tr>
</thead>
</table>

Rules are boring, but smart people found out the hard way that allowing free type variables is a pernicious source of language/compiler bugs.

Old (with one technical change to prevent free type variables):

$\Delta; \Gamma \vdash x : \Gamma(x)$
$\Delta; \Gamma \vdash c : \text{int}$

$\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2$
$\Delta \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2$

$\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1$
$\Delta; \Gamma \vdash e_2 : \tau_2$

$\Delta; \Gamma \vdash e_1 e_2 : \tau_1$
3. Add Typing Rules

Old (with one technical change to prevent free type variables):

\[
\Delta; \Gamma \vdash x : \Gamma(x) \quad \Delta; \Gamma \vdash c : \text{int}
\]

\[
\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Delta \vdash \tau_1 \\
\Delta; \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2
\]

\[
\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2 \\
\Delta; \Gamma \vdash e_1 \, e_2 : \tau_1
\]

New:

\[
\Delta, \alpha; \Gamma \vdash e : \tau_1 \\
\Delta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau_1
\]

\[
\Delta; \Gamma \vdash e : \forall \alpha. \tau_1 \quad \Delta \vdash \tau_2 \\
\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]
\]

Example

Example (using addition):

\((\Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \, x) \,[\text{int}]\,[\text{int}]\,3\,(\lambda y : \text{int}. y + y)\)

Ouch.

Example (using addition):

\((\Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \, x) \,[\text{int}]\,[\text{int}]\,3\,(\lambda y : \text{int}. y + y)\)
Example

Example (using addition):

$$(\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \to \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y: \text{int}. y + y)$$

Ouch.

Just a syntax-directed derivation by instantiating the typing rules. Still, machines are better suited to this stuff.

System F (Tah Dah!)

$$e ::= c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e[\tau]$$

$$\tau ::= \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha. \tau$$

$$v ::= c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e$$

$$\Gamma ::= \cdot \mid \Gamma, x: \tau$$

$$\Delta ::= \cdot \mid \Delta, \alpha$$

$$\frac{}{\Delta; \Gamma \vdash x: \Gamma(x)} \quad \frac{}{\Delta; \Gamma \vdash c: \text{int}}$$

$$\frac{e \to e'}{e e_2 \to e' e_2} \quad \frac{e \to e'}{v e \to v e'} \quad \frac{e \to e'}{e[\tau] \to e'[\tau]}$$

$$\frac{(\lambda x: \tau. \ e) v \to e[v/x]}{(\Lambda \alpha. \ e)[\tau] \to e[\tau/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash \tau_2 \to \tau_1 \quad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x: \tau_1. \ e: \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e_1: \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash e_2: \tau_2}{\Delta; \Gamma \vdash e_1 \ e_2: \tau_1}$$

$$\frac{\Delta; \Gamma \vdash e: \forall \alpha. \tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_1]: \tau_1[\tau_2/\alpha]}$$

Examples

Perhaps the simplest polymorphic function...

Let $id = \Lambda \alpha. \lambda x: \alpha. \ x$

- $id$ has type $\forall \alpha. \alpha \to \alpha$

Examples

Perhaps the simplest polymorphic function...

Let $id = \Lambda \alpha. \lambda x: \alpha. \ x$

- $id$ has type $\forall \alpha. \alpha \to \alpha$
Examples

Perhaps the simplest polymorphic function...

Let $id = \Lambda \alpha. \lambda x : \alpha. x$

- $id$ has type $\forall \alpha. \alpha \to \alpha$
- $id [\text{int}]$ has type $\text{int} \to \text{int}$
- $id [\text{int} \ast \text{int}]$ has type $\text{int} \ast \text{int} \to \text{int} \ast \text{int}$

In ML you can't do the last one! What?!
Examples

Perhaps the simplest polymorphic function...

Let \( \text{id} = \Lambda \alpha. \lambda x : \alpha. x \)

- \( \text{id} \) has type \( \forall \alpha. \alpha \rightarrow \alpha \)
- \( \text{id \ [\text{int}]} \) has type \( \text{int} \rightarrow \text{int} \)
- \( \text{id \ [\text{int} * \text{int}]} \) has type \( \text{(int * int)} \rightarrow \text{(int * int)} \)
- \( \text{(id \ [\forall \alpha. \alpha \rightarrow \alpha]) \ id} \) has type \( \forall \alpha. \alpha \rightarrow \alpha \)

In ML you can’t do the last one! What?!

More Examples

Let \( \text{apply1} = \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \ x \)

- \( \text{apply1} \) has type \( \forall \alpha. \forall \beta. \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \)
- \( \text{apply1 \ [\text{int}]} \) has type \( \text{int} \rightarrow \text{int} \)
- \( \text{apply1 \ [\text{int} * \text{int}]} \) has type \( \text{(int * int)} \rightarrow \text{(int * int)} \)
- \( \text{(id \ [\forall \alpha. \alpha \rightarrow \alpha]) \ id} \) has type \( \forall \alpha. \alpha \rightarrow \alpha \)

In ML you can’t do the last one! What?!
More Examples

Let apply1 = \(\Lambda\alpha.\Lambda\beta.\lambda x:\alpha.\lambda f:\alpha\to\beta.\ f\ x\)

- apply1 has type \(\forall\alpha.\forall\beta.\alpha\to(\alpha\to\beta)\to\beta\)
- \(\cdot; g:\text{int}\to\text{int} \vdash (\text{apply1} \ [\text{int}][\text{int}] \ 3 \ g) : \text{int}\)

Let apply2 = \(\Lambda\alpha.\lambda x:\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\ f\ x\)

- apply2 has type \(\forall\alpha.\alpha\to(\forall\beta.(\alpha\to\beta)\to\beta)\)
More Examples

Let apply1 = \( \Lambda \alpha . \Lambda \beta . \lambda x : \alpha . \lambda f : \alpha \rightarrow \beta . f \ x \)

- apply1 has type \( \forall \alpha . \forall \beta . \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \)
- \( \cdot ; g : \text{int} \rightarrow \text{int} \vdash (\text{apply1}[\text{int}][\text{int}] 3 \ g) : \text{int} \)

Let apply2 = \( \Lambda \alpha . \lambda x : \alpha . \Lambda \beta . \lambda f : \alpha \rightarrow \beta . f \ x \)

- apply2 has type \( \forall \alpha . \alpha \rightarrow (\forall \beta . (\alpha \rightarrow \beta) \rightarrow \beta) \) (also impossible in ML!)
- \( \cdot ; g : \text{int} \rightarrow \text{string}, h : \text{int} \rightarrow \text{int} \vdash (\text{let} \ z = \text{apply2}[\text{int}] \ \text{in} \ z (3 \ \text{int} \ h) \ \text{[string]} \ g) : \text{string} \)

Let twice = \( \Lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \rightarrow \alpha . f \ (f \ x) \).

- twice has type
4. Type Safety and Metatheory

- Safety: System F is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Even with self application — we saw id [τ] id
- Parametricity, a.k.a. “theorems for free”
  - Example: If ·; · ⊢ e : ∀α.∀β.(α * β) → (β * α), then e is equivalent to Λα. Λβ. λx:α * β. (x:2, x:1).

More Examples

Let apply1 = Λα. Λβ. λx : α. λf : α→β. f x

» apply1 has type ∀α.∀β.α → (α → β) → β

» ·; g:int → int ⊢ (apply1 [int][int] 3 g) : int

Let apply2 = Λα. λx : α. Λβ. λf : α→β. f x

» apply2 has type ∀α.α → (∀β.(α → β) → β) → β

(also impossible in ML!)

» ·; g:int → string, h:int → int ⊢ (let z = apply2 [int] in z (z 3 [int] h) [string] g) :
  string

Let twice = Λα. λx : α. λf : α → α. (f x).

» twice has type ∀α.α → (α → α) → α

» Could this be any more polymorphic?
4. Type Safety and Metatheory

- Safety: System F is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Even with self application — we saw id \([\tau]\) id
- Parametricity, a.k.a. “theorems for free"
  - Example: If \(\cdot \cdot \vdash e : \forall \alpha. \forall \beta.(\alpha \times \beta) \rightarrow (\beta \times \alpha)\),
    then \(e\) is equivalent to \(\Lambda \alpha. \Lambda \beta. \lambda x: \alpha \times \beta. (x.2, x.1)\).
    Every term with this type is the swap function!!

Intuition: \(e\) has no way to make an \(\alpha\) or a \(\beta\) and it cannot tell what \(\alpha\) or \(\beta\) are or raise an exception or diverge...

How many terms have type \(\forall \alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)\)?

Note: Mutation breaks everything :(
What next?

Now that we have System F...

- What hath we wrought? Example of our mighty new powers.
- How/why ML is more restrictive and implicit.

Security from safety?

Example: A process $e$ should not access files it did not open (fopen checks permissions)

Require an untrusted process $e$ to type-check as follows:

$$\vdash e : \forall \alpha. \{fopen : string \rightarrow \alpha, fread : \alpha \rightarrow \text{int}\} \rightarrow \text{unit}$$

This type ensures that a process won’t “forge a file handle” and pass it to fread

So fread doesn’t need to check (faster), file handles don’t need to be encrypted (safer), etc.
Moral of Example

In STLC, type safety just meant not getting stuck
Type abstraction gives us new powers, e.g. secure interfaces!
Suppose we (the system library) implement file-handles as ints.
Then we instantiate $\alpha$ with int, but untrusted code cannot tell

Memory safety is a necessary but insufficient condition for
language-based enforcement of strong abstractions

Are types used at run-time?

We said polymorphism was about “many types for same term”,
but for clarity and easy checking, we changed:
- The syntax via $\Lambda \alpha. e$ and $e[\tau]$
- The operational semantics via type substitution
- The type system via $\Delta$

Claim: The operational semantics did not “really” change; types
need not exist at run-time

More formally: Erasing all types from System F produces an
equivalent program in the untyped lambda calculus

Strengthened induction hypothesis: If $e \rightarrow e_1$ in System F and
$\text{erase}(e) \rightarrow e_2$ in untyped lambda-calculus, then
$e_2 = \text{erase}(e_1)$
Are types used at run-time?

We said polymorphism was about “many types for same term”, but for clarity and easy checking, we changed:

- The syntax via $\Lambda \alpha. e$ and $e \left[ \tau \right]$
- The operational semantics via type substitution
- The type system via $\Delta$

Claim: The operational semantics did not “really” change; types need not exist at run-time

More formally: Erasing all types from System F produces an equivalent program in the untyped lambda calculus

Strengthened induction hypothesis: If $e \rightarrow e_1$ in System F and $\text{erase}(e) \rightarrow e_2$ in untyped lambda-calculus, then $e_2 = \text{erase}(e_1)$

“Erasure and evaluation commute”

### Erasure

Erasure is easy to define:

$$\text{erase}(c) = c$$

$$\text{erase}(x) = x$$

$$\text{erase}(e_1 e_2) = \text{erase}(e_1) \text{erase}(e_2)$$

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

$$\text{erase}(\Lambda \alpha. e) = \lambda . \text{erase}(e)$$

$$\text{erase}(e \left[ \tau \right]) = \text{erase}(e)$$

In pure System F, preserving evaluation order isn’t crucial, but it is with fix, exceptions, mutation, etc.

Zach Tatlock

CSE 505 Autumn 2017, Lecture 18

25
Erasure is easy to define:

\[
\begin{align*}
\text{erase}(c) &= c \\
\text{erase}(x) &= x \\
\text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{ erase}(e_2)
\end{align*}
\]

In pure System F, preserving evaluation order isn't crucial, but it is with fix, exceptions, mutation, etc.

Zach Tatlock
CSE 505 Autumn 2017, Lecture 18
25
Erasure

Erasure is easy to define:

\[
\begin{align*}
erase(c) &= c \\
erase(x) &= x \\
erase(e_1 e_2) &= \erase(e_1) \erase(e_2) \\
erase(\lambda x : \tau. e) &= \lambda x. \erase(e) \\
erase(\Lambda \alpha. e) &= \lambda \alpha. \erase(e) \\
erase(e [\tau]) &= 
\end{align*}
\]

In pure System F, preserving evaluation order isn't crucial, but it is with `fix`, exceptions, mutation, etc.

Zach Tatlock
CSE 505 Autumn 2017, Lecture 18 25
Erasure is easy to define:

\[
\begin{align*}
\text{erase}(c) &= c \\
\text{erase}(x) &= x \\
\text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{ erase}(e_2) \\
\text{erase}(\lambda x: \tau. e) &= \lambda x. \text{ erase}(e) \\
\text{erase}(\Lambda \alpha. e) &= \lambda . \text{ erase}(e) \\
\text{erase}(e[\tau]) &= \text{ erase}(e) 0
\end{align*}
\]

In pure System F, preserving evaluation order isn’t crucial, but it is with fix, exceptions, mutation, etc.

Connection to reality... or at least ML

System F has been one of the most important theoretical PL models since the 1970s and inspires languages like ML.

But you have seen ML polymorphism and it looks different. In fact, it is an implicitly typed restriction of System F.

These two qualifications ((1) implicit, (2) restriction) are deeply related.

ML Restrictions

- All types have the form \( \forall \alpha_1, \ldots, \alpha_n. \tau \) where \( n \geq 0 \) and \( \tau \) has no \( \forall \). (Prenex-quantification; no first-class polymorphism.)
ML Restrictions

- All types have the form ∀α₁,...,αₙ.τ where n ≥ 0 and τ has no ∀. (Prenex-quantification; no first-class polymorphism.)

- Only let (rec) variables (e.g., x in let x = e₁ in e₂) can have polymorphic types. So n = 0 for function arguments, pattern variables, etc. (Let-bound polymorphism)
  - So cannot (always) desugar let to λ in ML

ML Restrictions: Why?

ML-style polymorphism can seem weird after you have seen System F. And the restrictions do come up in practice, though tolerable.

Type inference for ML is decidable and efficient in practice, though pathological programs of size \( O(n^3) \) though undecidable (1992) but value-restriction restores soundness.

Type inference for System F (given untyped O

\( e \) is a “syntactic value”

- A variable, constant, function definition, ...
- Called the “value restriction” (relaxed partially in OCaml)
ML Restrictions: Why?
ML-style polymorphism can seem weird after you have seen System F. And the restrictions do come up in practice, though tolerable.

- Type inference for System F (given untyped $e$, is there a System F term $e'$ such that $\text{erase}(e') = e$) is undecidable (1995)
- Type inference for ML with polymorphic recursion is undecidable (1992)
- Type inference for ML is decidable and efficient in practice, though pathological programs of size $O(n)$ and run-time $O(n)$ can have types of size $O(2^{2^n})$
- The type inference algorithm is unsound in the presence of ML-style mutation, but value-restriction restores soundness

Based on unification
Extensions to the ML type system to be closer to System F:

- Usually require some type annotations

- Are judged by:
  - Soundness: Do programs still not get stuck?
  - Conservatism: Do all (or most) old ML programs still type-check?
  - Power: Does it accept many more useful programs?
  - Convenience: Are many new types still inferred?