

## Polymorphism

- Outline
  - *Computability properties in System F*
    - Type checking
    - Type inference



---

---

---

---

---

---

---

---

## The polymorphic lambda calculus

- $e ::= v$  variables  
|  $\lambda x: t. e$  abstraction  
|  $e_1 e_2$  application  
|  $\Lambda X. e$  type abstraction  
|  $e[t]$  type application
- $t ::= X$  type variables  
|  $t_1 \rightarrow t_2$  function types  
|  $\forall X. t$  type quantification



---

---

---

---

---

---

---

---

## Typing judgments

A type judgment is a sequent  $\Gamma \vdash e : t$

- $e$  is a program
- $t$  is a type
- $\Gamma$  is a type assignment list:

- $\Gamma ::= \{ \}$  empty context  
|  $\Gamma, x: t$  term variable  
|  $\Gamma, X$  type variable



---

---

---

---

---

---

---

---

## Sequents

The order in  $\Gamma$  is significant:

$$X, y: X \rightarrow X, x: X \vdash (y \ x) : X$$

Axiom rule:

$$\frac{}{\Gamma_1, x:t, \Gamma_2 \vdash x : t} \text{ axiom}$$



---

---

---

---

---

---

---

---

## Typing rules for simple fragment

Typing rules for the simply-typed  $\lambda$  calculus.

$$\frac{}{\Gamma, x:t \vdash x : t} \text{ var}$$

$$\frac{\Gamma \vdash e_1 : (s \rightarrow t) \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e_1 \ e_2) : t} \text{ app}$$

$$\frac{\Gamma, x:s \vdash t}{\Gamma \vdash (\lambda x:s. e) : (s \rightarrow t)} \text{ abs}$$



---

---

---

---

---

---

---

---

## Typing rules for System F

Additional rules for System F.

$$\frac{\Gamma, X \vdash e : T}{\Gamma \vdash (\Lambda X. e) : (\forall X. T)} \text{ all intro}$$

$$\frac{\Gamma \vdash e : \forall X. T_2}{\Gamma \vdash e[T_1] : T_2[T_1/X]} \text{ all elim}$$



---

---

---

---

---

---

---

---

## Typing of identity

$$\frac{\frac{\frac{X, x: X \vdash x : X}{X \vdash (\lambda x: X. x) : (X \rightarrow X)}}{\vdash (\Lambda X. \lambda x: X. x) : (\forall X. X \rightarrow X)}}{1}$$



---

---

---

---

---

---

---

---

## Type erasure

- Type erasure
  - The types in a  $F$  program are effectively useless
  - We don't ever make decisions based on type information
  - Most implementations delete the types at runtime
- Type reconstruction (inference)
  - We would rather not write down the types in the source program



---

---

---

---

---

---

---

---

## Type erasure

$$\begin{aligned} \text{erase}(x) &= x \\ \text{erase}(\lambda x: t. e) &= \lambda x. \text{erase}(e) \\ \text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{erase}(e_2) \\ \text{erase}(\Lambda X. e) &= \text{erase}(e) \\ \text{erase}(e[t]) &= \text{erase}(e) \end{aligned}$$



---

---

---

---

---

---

---

---

## Type reconstruction

- Given a program  $e$  in the untyped  $\lambda$  calculus, is there a well-typed term  $e'$  in System F such that  $erase(e') = e$ ?
- Undecidable (Wells, 1994).



---

---

---

---

---

---

---

---

## Type inference

- ML uses a restricted type system where all types are in *prenex* form (all quantifiers are outermost).
- A type is prenex if it has the form  $\forall X_1. \dots \forall X_n. t$  and  $t$  has no quantifiers.



---

---

---

---

---

---

---

---

## Type inference

- Type inference: for any term  $e$  in the untyped  $\lambda$  calculus, find a program  $e' : t$  in System F such that  $erase(e') = e$  and  $t$  is in prenex form; or prove that no such typing exists.
- The usual three cases: var, app, abs.
- Algorithm
  - Ignore let-polymorphism for the moment
  - We'll need a type context  $\Gamma$
  - We'll need a type *unification* algorithm



---

---

---

---

---

---

---

---

### Var case

To infer a type for  $e$ , suppose  $e = x$ :

- The typing rule is:

$$\frac{}{\Gamma_1, x:t, \Gamma_2 \vdash x : s} \text{ var}$$

- Choose  $s = t$



---

---

---

---

---

---

---

---

### App case

- Suppose  $e = e_1 e_2$

- The typing rule is

$$\frac{\Gamma \vdash e_1 : (s_1 \rightarrow t) \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash (e_1 e_2) : t} \text{ app}$$

- Recursively infer types  $e_1 : s_1 \rightarrow t$ , and  $e_2 : s_2$
- Unify  $s_1, s_2$  to get  $s_1 = s_2$
- The type is  $t$



---

---

---

---

---

---

---

---

### Abs case

- Suppose  $e = \lambda x. e'$

- The typing rule for simply-typed calculus:

$$\frac{\Gamma, x:s \vdash e' : t}{\Gamma \vdash (\lambda x:s.e') : (s \rightarrow t)} \text{ abs}$$

- Where do we get  $s$ ?
- Make up a new type variable  $\alpha$  for  $s$ , and infer the type of  $e$
- $s$  may get unified



---

---

---

---

---

---

---

---

## Quantify

- Finally, quantify all free type variables with  $\forall$
- Wrap the expression in  $\Lambda X...$  quantifiers



---

---

---

---

---

---

---

---

## Identity

- To infer a type for  $\lambda x.x$
- Choose a new type variable  $\alpha$  for  $x$
- Then the body has type  $\alpha$

$$\frac{x: \alpha \vdash x : \alpha}{\vdash (\lambda x: \alpha. x) : (\alpha \rightarrow \alpha)} \text{ abs}$$

- Quantify:  $(\Lambda \alpha. \lambda x: \alpha. x) : \forall \alpha. \alpha \rightarrow \alpha$



---

---

---

---

---

---

---

---

## Eta, part 1

- Infer a type for  $\lambda f. \lambda x. f(x)$
- Choose types  $\alpha, \beta$  to get

$$f: \alpha, x: \beta \vdash (f x) : ?$$

- $\alpha$  must be an arrow type  $\gamma \rightarrow \delta$

$$f: \gamma \rightarrow \delta, x: \beta \vdash (f x) : ?$$



---

---

---

---

---

---

---

---

## Eta, part 2

- $\gamma$  and  $\beta$  must be the same

$$f: \gamma \rightarrow \delta, x: \gamma \vdash (f x) : ?$$

- Use the **app** rule

$$\frac{f: \gamma \rightarrow \delta, x: \gamma \vdash f : \gamma \rightarrow \delta \quad f: \gamma \rightarrow \delta, x: \gamma \vdash x : \gamma}{f: \gamma \rightarrow \delta, x: \gamma \vdash (f x) : \delta} \text{app}$$

- Quantify

$$\frac{}{\Lambda \gamma. \Lambda \delta. \lambda f. \gamma \rightarrow \delta. \lambda x. \gamma. f(x)} : \forall \gamma. \forall \delta. (\gamma \rightarrow \delta) \rightarrow \gamma \rightarrow \delta$$



---

---

---

---

---

---

---

---

---

---

## Type inference algorithm (W)

Algorithm  $\text{infer}(\Gamma, \sigma, e) : \text{type}$

- $\Gamma$  is a type environment
- $\sigma$  is a type substitution  $\alpha_1 = t_1, \dots, \alpha_n = t_n$
- $e$  is an untyped program
- The result is a type for  $e$



---

---

---

---

---

---

---

---

---

---

## Unification

- Construct a type substitution  $\sigma$  that makes two types  $t_1$  and  $t_2$  the same
- Recursive algorithm:
  - If  $t_1$  is a type variable  $\alpha$ , then add  $\alpha = t_2$  to  $\sigma$
  - If  $t_2$  is a type variable  $\beta$ , then add  $\beta = t_1$  to  $\sigma$
  - Otherwise, the two types must both be arrow types; recursively unify them



---

---

---

---

---

---

---

---

---

---

## Unification

```
let unify( $\sigma, t_1, t_2$ ) =  
  if  $t_1 = \alpha$  then  
    if  $(\alpha = t_3) \in \sigma$  then  
      unify( $\sigma, t_3, t_2$ )  
    else  
       $\sigma \cup \{\alpha = t_2\}$   
  else if  $t_2 = \alpha$  then  
    similar code for  $t_2$   
  else if  $t_1 = (t_{11} \rightarrow t_{12}) \wedge t_2 = (t_{21} \rightarrow t_{22})$  then  
    unify(unify( $\sigma, t_{11}, t_{21}$ ),  $t_{12}, t_{22}$ )  
  else  
    error("program has type  $t_2$  but is used with type  $t_1$ ")
```



---

---

---

---

---

---

---

---

---

---

## Type inference, var, abs

Algorithm  $infer(\Gamma, \sigma, e) : (\sigma, type)$

- If  $e = v$ , return  $\sigma, \Gamma(v)$
- If  $e = \lambda x. e'$ , choose a new type variable  $\alpha$ 
  - let  $\sigma', t_2 = infer(\Gamma \cup \{x : \alpha\}, \sigma, e)$
  - return  $\sigma', \alpha \rightarrow t_2$



---

---

---

---

---

---

---

---

---

---

## Type inference, app

- If  $e = e_1 e_2$ , choose new type variables  $\alpha, \beta$ 
  - let  $\sigma_1, t_1 = infer(\Gamma, \sigma, e_1)$
  - let  $\sigma_2, t_2 = infer(\Gamma, \sigma_1, e_2)$
  - let  $\sigma_3 = unify(\sigma_2, \alpha \rightarrow \beta, t_1)$
  - let  $\sigma_4 = unify(\sigma_3, \alpha, t_2)$
  - return  $\sigma_4, \beta$



---

---

---

---

---

---

---

---

---

---

## Type inference

Quantification phase: given  $e : (\sigma, t)$  apply the substitution to  $t$  and quantify all free type variables.

- Do we get the most general type (for example,  $(\lambda x.x) \in (\mathbb{Z} \rightarrow \mathbb{Z})$  would be too specific)
  - The unification algorithm (Robinson, 1971) always gives the *most general unifier*.
  - The proof is by structural induction (exercise).
- Do we always find a type if one exists? (structural induction)



---

---

---

---

---

---

---

---

## Expressivity

- What do we lose with prenex types?
  - Functions can't return polymorphic values
  - Nested functions are not polymorphic (strict interpretation)
- Let-polymorphism
  - Recover some of the polymorphism of nested functions



---

---

---

---

---

---

---

---

## Let-polymorphism

A let definition has the following equivalence:

$$\mathbf{let } v = e_1 \mathbf{ in } e_2 \equiv (\lambda v.e_2) e_1$$

```
let f = (fun x -> x) in  
let i = f 1 in  
let s = f "hello" in  
let g = f f in
```



---

---

---

---

---

---

---

---

## Let-polymorphism

A let definition has the following equivalence:

$$\mathbf{let } v = e_1 \mathbf{ in } e_2 \equiv (\lambda v. e_2) e_1$$

**let** f = (fun x -> x) **in**  $\alpha \rightarrow \alpha$

**let** i = f 1 **in**  $\mathbb{Z} \rightarrow \mathbb{Z}$

**let** s = f "hello" **in**  $string \rightarrow string$

**let** g = f f **in**  $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$



---

---

---

---

---

---

---

---

## Let-polymorphism

**let** f = (fun x -> x) **in**  $T \equiv \forall \alpha. \alpha \rightarrow \alpha$

**let** i = f 1 **in**  $T[\mathbb{Z}]$

**let** s = f "hello" **in**  $T[string]$

**let** g = f f **in**  $T[T[\alpha]]$



---

---

---

---

---

---

---

---

## Computation interpretation

A let definition has the following equivalence:

$$\mathbf{let } v = e_1 \mathbf{ in } e_2 \equiv (\lambda v. e_2) e_1$$

$$\frac{\Gamma \vdash e_2[e_1/v] : t}{\Gamma \vdash ((\lambda v. e_2) e_1) : t} \text{ def} \quad \frac{}{\Gamma \vdash (\mathbf{let } v = e_1 \mathbf{ in } e_2) : t} \text{ let}$$



---

---

---

---

---

---

---

---

## Inference for let-polymorphism

$\Gamma \vdash (\text{let } v = e_1 \text{ in } e_2) : ?$

- First, infer the type for  $e_1 : t_1$
- Quantify all the free type variables in  $t_1$  **except** those that are already defined in  $\Gamma$
- Infer a type for  $e_2$

$\Gamma, v : \forall \alpha_1 \dots \forall \alpha_n. t_1 \vdash e_2 : ?$

- When getting the value for  $v$ , instantiate all  $\alpha_1, \dots, \alpha_n$  with fresh type variables.



---

---

---

---

---

---

---

---