CS 456

Programming Languages
Fall 2024

Week 5

System F

Polymorphism

- In OCaml polymorphic functions can have arguments of different types:

```
let id x = x
val id : ‘a -> ‘a = <fun>
let double f x = f (f x)
val double: (‘a -> ‘a) -> ‘a -> ‘a
let compose f g a = g (f a)
val compose: (‘a -> ‘b) -> (‘b -> ‘c) -> ‘a -> ‘c


let foo = id 1
let bar = double not (id True)
```

Polymorphism

- **Principle of Abstraction**: When similar functions are carried out by distinct piece of code, it is generally a good idea to combine them into one by abstracting out the varying parts.
- In OCaml polymorphic functions can have arguments of different types:

```
let id = (\ x -> x) in (id 1, id true)
let double := (\ f x -> f (f x)) in

if (double plus1 len < 5)
   then (hd (double tl l)) else (hd l)
```

- **Problem**: We can't type id and double in STLC
- **Solution**?

System F

> The fundamental problem ***addressed*** by a type theory is to insure that programs have meaning. The fundamental problem ***caused*** by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.
>
> —*Mark Manasse.*

- We'll be looking at **System F**, a calculus in which polymorphic functions can be written.
  - Name was coined by Jean-Yves Girad, was originally a logic
- A core calculus for **parametric polymorphism**.
  - Can capture module systems and data abstraction
  - Enough for type safe 'pure' OO (w/o inheritance)

System F

★ Here is the syntax of **pure System F**, with new bits **highlighted**.

t ::= x | λx:T.t | t t
| ΛX.t ⇐ Type Abstraction
| t [T] ⇐ Type Application

v ::= λx:T.t | ΛX.t

T ::= T → T
| ∀X.T ⇐ Universal Type
| X ⇐ Type Variable

System F

- Here are the new bits of the operational semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{EApp}_1 \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}\ \text{EApp}_2$$

$$\frac{}{(\lambda x{:}T.e)\ v \longrightarrow e_1\ [x \mapsto v]}\ \text{EAppAbs}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ [T_2] \longrightarrow e_1'\ [T_2]}\ \text{ETApp}$$

Where is ETApp_2?

$$\frac{}{(\Lambda X.e_1)\ [T] \longrightarrow e_1\ [X \coloneqq T]}\ \text{ETAppTAbs}$$

Example

(ΛX. λx:X. x) [bool] true

(λf:(∀X.X→X). **if** f [bool] true **then** f [nat] 1
else 2) id

System F

★ Here are the **new bits** of the typing rules

$$\frac{\Gamma, [x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \rightarrow T_2} \quad \text{TABS}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{TVAR}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \quad \text{TAPP}$$

$$\frac{\Gamma \vdash t : T_2}{\Gamma \vdash \Lambda X.t : \forall X.T_2} \quad \text{TTABS}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_2}{\Gamma \vdash t_1\ [T_1] : T_2[X \coloneqq T_1]} \quad \text{TTAPP}$$

Concept Check

- What is the type of this System F term:

$$\vdash \Lambda T.\ \text{double}\ [T \to T]\ (\text{double}\ [T1]) : \textbf{?}$$

where double $\equiv \Lambda X.\ \lambda f{:}X{\to}X.\ \lambda y{:}X.\ f\ (f\ y)$

System F Metatheory

- System F shares many of STLC's metatheoretic properties:
  - **Theorem** [PROGRESS]: Suppose t is a closed, well-typed **System F term** (i.e. $\vdash p : T$). Then either t is a value or there exists some t' such that t evaluates to t'.
  - **Theorem** [PRESERVATION]: Suppose t is a well-typed **System F term** under context Γ (i.e. $\Gamma \vdash p : T$). Then, if t evaluates to t', t' is also a well-typed term under context Γ, with the same type as t.
  - **Theorem** [NORMALIZATION]: Suppose t is a closed, well-typed **System F term** (i.e. $\vdash p : T$). Then, t halts, that is there must exist some value v, such that t evaluates to v.

System F Metatheory

★ Type Erasure

$\lceil x \rceil = x$

$\lceil \lambda x{:}T.M \rceil = \lambda x.\lceil M \rceil$

$\lceil M_1\ M_2 \rceil = \lceil M_1 \rceil\ \lceil M_2 \rceil$

$\lceil \Lambda X.t \rceil = \lceil t \rceil$

$\lceil t_1\ [T_2] \rceil = \lceil t_1 \rceil$

- **Theorem** [SOUNDNESS OF TYPE ERASURE]: If a **System F term** t evaluates to t', then the erasure of t evaluates to the erasure of t' under the untyped evaluation relation. That is, $t \longrightarrow t'$ implies $\lceil t \rceil \longrightarrow \lceil t' \rceil$.

System F Metatheory

★ OTOH, the metatheory of System F diverges from STLC in key ways with respect to type inference:

$\lceil x \rceil = x$

$\lceil \lambda x{:}T.M \rceil = \lambda x.\lceil M \rceil$

$\lceil M_1\, M_2 \rceil = \lceil M_1 \rceil\ \lceil M_2 \rceil$

$\lceil \Lambda X.t \rceil = \lceil t \rceil$

$\lceil t_1\, [T_2] \rceil = \lceil t_1 \rceil$

- **Theorem** [TYPE INFERENCE IS UNDECIDABLE]: Suppose m is a closed term in the untyped lambda calculus. Then it is undecidable if there exists some well-typed term system F term, t , such that $\lceil t \rceil = m$.

★ Bummer!

System F Fragments

- But, *some* restricted forms of System F have tractable type reconstruction.
- Key Idea: Restrict uses of polymorphism in types to enable type reconstruction.
- Can you think of one?
  - Type schemas from let-polymorphism are restricted from of universal types
  - Quantifiers appear at the start of a formula
  - Also called prenex polymorphism

- **Theorem** [Prenex Type Inference]: Suppose m is a closed term in the untyped lambda calculus. Then it is decidable if there exists some well-typed term system F term, t , *which only contains types in prenex normal form,* such that ⌈t ⌉ = m.

System F Fragments

- ★ Another restriction is **rank-2 polymorphism**.
- ★ A type is said to be or rank 2 if no path from its root to a ∀ quantifier passes to the left of 2 or more arrows, when drawn as a tree.
  - $(\forall X. X \to X) \to Nat$
  - $Nat \to (\forall X. X \to X) \to Nat \to Nat$
  - $((\forall X. X \to X) \to Nat) \to Nat$
- Contrast:

$$f :: \forall r. \forall a. ((a \to r) \to a \to r) \to r$$

with

$$f' :: \forall r. (\forall a. (a \to r) \to a \to r) \to r$$

- ★ **<u>Theorem</u>** <u>[RANK-2 TYPE RECONSTRUCTION]</u>: Suppose m is a closed term in the untyped lambda calculus. Then it is decidable if there exists some well-typed term system F term, t , *which only contains types in of rank-2 or less,* such that ⌈ t ⌉ = m.

System F Fragments

- How high can we go (in rank?)

2

- **Theorem** [RANK-(>2) TYPE RECONSTRUCTION]: Suppose m is a closed term in the untyped lambda calculus. Then it is **undecidable** if there exists some well-typed term system F term, t , *which only contains types in of rank-n or less (where n > 2),* such that ⌈t⌉ = m.

- However, if let-bound parameters with a polymorphic type are annotated, type reconstruction for higher-rank let-polymorphism is possible.

let polyf (f : ∀ a. a → a) := (f 1, f True) in e

Prenex Polymorphism

- ★ In other good news, **some** restricted forms of System F have tractable type reconstruction.
- ★ **Key Idea**: Restrict uses of polymorphism in types to enable type reconstruction.
- ★ Can you think of one?
  - **Quantifiers only appear at the start of a formula**
  - Also called prenex polymorphism
- **Theorem** [Prenex TYPE INFERENCE]: Suppose m is a closed term in the untyped lambda calculus. Then it is decidable if there exists some well-typed term system F term, t , *which only contains types in prenex normal form,* such that ⌈t⌉ = m.

Prenex Predicative Polymorphism

- **Key Idea**: Restrict uses of polymorphism in types to enable type reconstruction.
- Can you think of one?
  - **Quantifiers only appear at the start of a formula and can only be instantiated with monomorphic types**

-

- This restriction can be expressed syntactically

$$\tau ::= b \mid \tau_1 \to \tau_2 \mid t$$
$$\sigma ::= \tau \mid \forall t.\ \sigma$$
$$e ::= x \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \mid \Lambda t.e \mid e\ [\tau]$$

- Type application is restricted to mono types

  $(\forall t.\ t \to t) \to (\forall t.\ t \to t)$ is <u>not</u> a valid type

- Abstraction only on mono types
- Cannot apply "id" to itself anymore
- Simple semantics and termination proof

Expressiveness

- We have simplified too much !

- Not expressive enough to encode

  bool = $\forall t.t \rightarrow t \rightarrow t$

  true = $\Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ x$

  false = $\Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ y$

  But such encodings are only of theoretical interest anyway

Is it expressive enough in practice?

Almost

Cannot write something like

$(\lambda s{:}\forall t.\tau.\ \ldots\ s\ [nat]\ x\ \ldots\ \ s\ [bool]\ y)\ (\Lambda t.\ \ldots$ code for sort)

Because the type of formal argument s cannot be polymorphic

ML's Polymorphic Let

ML solution: slight extension

Introduce "`let x : σ = e1 in e2`"

- With the semantics of "(λx : σ.e2) e1"
- And typed as "[e1/x] e2"

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

This lets us write the polymorphic sort as

```
let
   s : ∀t.τ = Λt. ... code for  polymorphic sort ...
in
  ... s [nat] x .... s [bool] y
```

ML Polymorphism and References

let is evaluated using call-by-value but is typed using call-by-name

What if there are side effects ?

Example:

```
let  x : ∀t. (t -> t) ref = Λt. ref (λx : t. x)
in
   x [bool] := λx: bool. not x
   (! x [int]) 5
```

Will apply "`not`" to 5

Similar examples can be constructed with exceptions

It took 10 years to find and agree on a clean solution

The Value Restriction in ML

A type in a let is generalized only for syntactic values

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

e_1 is a syntactic value or σ is monomorphic

Since e1 is a value, its evaluation cannot have side-effects

In this case call-by-name and call-by-value are the same

In the previous example ref (λx:t. x) is not a value

This is not too restrictive in practice !

Recap

- System F = a core calculus for parametric polymorphism which extends STLC with **type abstraction** and type application
- Embodies meta-theoretic properties of polymorphic languages:

Restriction	Progress + Preservation	Normalization	Sound Type-Erasure Semantics	Type Reconstruction
None	✓	✓	✓	×
Prenex Polymorphism	✓	✓	✓	✓
Rank-2 Polymorphism	✓	✓	✓	✓
Rank-n Let-Polymorphism with polymorphic annotations	✓	✓	✓	✓