Polymorphic types

- Simply typed λ-calculus is “monomorphic”, i.e. a type has no “flexible” pieces
  $\tau ::= * \mid \tau \to \tau$
- "Good" programming languages have polymorphic types
- So we'd like to capture the essense of polymorphic types in our calculus

Polymorphic λ-calculus (System F)

- Extends simply-typed λ:
  - type syntax
  - expression/value syntax
  - typechecking rules
  - evaluation rules

Polymorphic type syntax

- Extend type syntax with a forall type
  $\tau ::= \ldots \mid \forall I.\tau \mid I$
- Can write types of polymorphic values:
  id : $\forall T.\ T \to T$
  map : $\forall T.\ \forall U.\ (T \to U) \to T\ \text{list} \to U\ \text{list}$
  nil : $\forall T.\ T\ \text{list}$

Polymorphic(ally typed) value syntax

- Syntax:
  $E ::= \ldots \mid \Lambda I.E \mid E[\tau]$
  $V ::= \ldots \mid \Lambda I.E$
  - $\Lambda I.E$ is a function that, given a type τ, gives back E with τ substituted for I
  - Use such values by instantiating them: $E[\tau]$
    - $E[\tau]$ is like function application

An example

```
(* fun id x = x
   id:'a -> 'a *)
```

id = $\Lambda T.\ \lambda x{:}T.\ x$
: $\forall T.\ T \to T$

id [int] 3 $\to_\beta$
(λx:int. x) 3 $\to_\beta$
3

id [bool] $\to_\beta$
λx:bool. x

Another example

```
(* fun applyTwice f x = f (f x)
   applyTwice:('a->'a) -> 'a -> 'a *)
```

applyTwice =
$\Lambda T.\ \lambda f{:}T \to T.\ \lambda x{:}T.\ f\ (f\ x)$
: $\forall T.\ (T \to T) \to T \to T$

applyTwice [int] succ 3 $\to_\beta$
(λf:int→int. λx:int. f (f x)) succ 3 $\to_\beta^*$
succ (succ 3) $\to_\beta^*$
5

Yet another example

$map = \Lambda T.\ \Lambda U.\ fix\ (\lambda map{:}(T\to U)\to T\ list\to U\ list.$
$\lambda f{:}T\to U.\ \lambda lst{:}T\ list.$
fold (case (unfold lst) of
<nil=n> => <nil=()>
<cons=r> => <cons={hd=f (#hd r), tl=map f (#tl r)}>))
$: \forall T.\ \forall U.\ (T\to U)\to T\ list\to U\ list$

map [int] [bool] isZero [3,0,5] $\to_\beta^*$ [false,true,false]

- ML infers what the ΛI and [τ] should be

A final example

```
(* fun cool f = (f 3, f true) *)
```

cool $\equiv$ $\lambda f{:}(\forall T.T\to T)$. (f [int] 3, f [bool] true)
$:(\forall T.T\to T)\to(int * bool)$

cool id $\to_\beta$
(id [int] 3, id [bool] true) $\to_\beta^*$
((λx:int. x) 3, (λx:bool. x) true) $\to_\beta^*$
(3, true)

- Note: $\forall$ inside of λ and $\to$
  - **Can't write this in ML; not "prenex" form**
  - Type inference undecidable for full System F (and many interesting subsets); but decidable for ML-style polymorphism

Evaluation and typing rules

- Evaluation:

$$\frac{E \Downarrow (\Lambda I.\ E_1) \quad ([I\to\tau]E_1) \Downarrow V}{(E[\tau]) \Downarrow V}\ [\text{E-INST}]$$

- Typing:

$$\frac{\Gamma, I{::}Type \vdash E : \tau}{\Gamma \vdash (\Lambda I.E) : \forall I.\tau}\ [\text{T-POLY}]$$

$$\frac{\Gamma \vdash E{:}\forall I.\ \tau'}{\Gamma \vdash (E[\tau]) : [I\to\tau]\tau'}\ [\text{T-INST}]$$

Various kinds of functions

- $\lambda I.E$ is a function from *values* to *values*
- $\Lambda I.E$ is a function from *types* to *values*
- What about functions from *types* to *types*?
  - **Type constructors** like $\to$, list, BTree
    - We want them!
- What about functions from *values* to *types*?
  - **Dependent type constructors** like a way to build the type “arrays of length n”, where n is a run-time computed value
    - Pretty fancy, but would be cool

Type constructors

- What's the "type" of *list*?
  - Not a simple type, but a function from types to types
    - e.g. list(int) = int_list
  - There are lots of type constructors that take a single type and return a type
    - They all have the same "meta-type"
  - Other things take two types and return a type:
    - e.g. $\to$, assoc_list
- A "meta-type" is called a **kind**

Kinds

- A *type* describes a *set of values* or value constructors (a.k.a. functions) with a common structure
  $\tau ::= int \mid \tau_1 \to \tau_2 \mid \ldots$
- A *kind* describes a *set of types* or type constructors with a common structure
  $\kappa ::= * \mid \kappa_1 \Rightarrow \kappa_2$
  As in the s.t. λ calculus, * is the “base kind”
- Write $\tau{::}\kappa$ to say that a type τ has kind κ
  - int :: *
  - int$\to$int :: *
  - list :: * $\Rightarrow$ *
  - list int :: *
  - assoc_list :: * $\Rightarrow$ * $\Rightarrow$ *
  - assoc_list string int :: *

Kinded polymorphic λ-calculus (**System F_ω**)

- Full syntax:
  - $\boldsymbol{\kappa ::= * \mid \kappa_1 \Rightarrow \kappa_2}$
  - $\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \forall I{::}\boldsymbol{\kappa}.\tau \mid I \mid \boldsymbol{\lambda_\tau I{::}\kappa.\tau} \mid \boldsymbol{\tau_1\ \tau_2}$
  - $E ::= \lambda I{:}\tau.\ E \mid I \mid E_1\ E_2 \mid \Lambda I{::}\boldsymbol{\kappa}.E \mid E[\tau]$
  - $V ::= \lambda I.E \mid \Lambda I{::}\boldsymbol{\kappa}.E$
  - Functions and applications at both the value and the type level
  - Arrows at both the type and kind level

Examples

pair =
  λ_τT::*. λ_τU::*. {first:T, second:U}
  :: * $\Rightarrow$ * $\Rightarrow$ *
pair int bool "$\rightarrow_\beta$" {first:int, second:bool}

{first=5, second=true} : pair int bool

swap =
  ΛP::type $\Rightarrow$ type $\Rightarrow$ type. ΛT::*. ΛU::* .
    λp:P T U . {first=#second p, second=#first p}
  : $\forall$P::* $\Rightarrow$ * $\Rightarrow$ *. $\forall$ T::*. $\forall$ U::*.
      P T U $\rightarrow$ P U T

swap [pair] [int] [bool] ...

Expression typing rules

$$\frac{\Gamma \vdash \tau_1{::}* \qquad \Gamma, I{:}\tau_1 \vdash E{:}\tau_2}{\Gamma \vdash (\lambda I{:}\tau_1.\ E) : \tau_1 \rightarrow \tau_2}\ [\text{T-ABS}]$$

$$\frac{\Gamma, I{::}\kappa \vdash E{:}\tau}{\Gamma \vdash (\Lambda I{::}\kappa.E) : \forall I{::}\kappa.\tau}\ [\text{T-POLY}]$$

$$\frac{\Gamma \vdash E : \forall I{::}\kappa.\tau' \qquad \Gamma \vdash \tau{::}\kappa}{\Gamma \vdash (E[\tau]) : [I \rightarrow \tau]\tau'}\ [\text{T-INST}]$$

(T-VAR and T-APP unchanged)

Type kinding rules

$$\frac{}{\Gamma \vdash \text{int} :: *}\ [\text{K-INT}] \qquad \frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) :: *}\ [\text{K-ARROW}]$$

$$\frac{\Gamma, I{::}\kappa \vdash \tau :: *}{\Gamma \vdash (\forall I{::}\kappa.\ \tau) :: *}\ [\text{K-FORALL}] \qquad \frac{I{::}\kappa \in \Gamma}{\Gamma \vdash I{::}\kappa}\ [\text{K-VAR}]$$

$$\frac{\Gamma, I{::}\kappa_1 \vdash \tau{::}\kappa_2}{\Gamma \vdash (\lambda_\tau\ I{::}\kappa_1.\ \tau) :: \kappa_1 \rightarrow \kappa_2}\ [\text{K-ABS}] \qquad \frac{\Gamma \vdash \tau_1 :: \kappa_2 \rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash (\tau_1\ \tau_2) :: \kappa_1}\ [\text{K-APP}]$$

Higher-order kinds?

- Could "lift" polymorphism to type level...
  $\kappa ::= \ldots \mid \forall I.\kappa \mid I$
  $\tau ::= \ldots \mid \Lambda_\tau\ I{::}\kappa\ .\ \tau \mid \kappa[\tau]$
- Could "lift" meta-kinding to kind level…
  $M ::= * \mid M \Rightarrow M$
  $\kappa ::= \ldots \mid \lambda_\kappa\ I{::}M.\kappa \mid \kappa_1\ \kappa_2$
- …and so on to arbitrary "tower" of meta-levels of language

Phase distinction

- Could also collapse all levels of language down to one:
  $E ::= I \mid \lambda I{:}E.E \mid E_1\ E_2$
- Loses **phase distinction** between run-time and typecheck-time
  - Fundamental to achieving benefits of type systems
  - (More generally, might be desirable to have many phases: compile, link, initialize, run, etc.; could use meta-levels in language to encode these phase distinctions.)

Summary

- Saw ever more powerful static type systems for the λ-calculus
  - Simply typed λ-calculus
  - Polymorphic λ-calculus, a.k.a. System F
  - Kinded poly. λ-calculus, a.k.a. System F_ω
- Exponential ramp-up in power, once build up sufficient critical mass
- Real languages typically offer some of this power, but in restricted ways
  - Could benefit from more expressive approaches

Other uses

- Compiler internal representations for advanced languages
  - E.g. FLINT: compiles ML, Java, …
- Checkers for interesting non-type properties, e.g.:
  - proper initialization
  - static null pointer dereference checking
  - safe explicit memory management
  - thread safety, data-race freedom