Polymorphic types

• Simply typed \( \lambda \)-calculus is “monomorphic”, i.e. a type has no “flexible” pieces

\[ \tau ::= * \mid \tau \rightarrow \tau \]

• ”Good” programming languages have polymorphic types

So we’d like to capture the essence of polymorphic types in our calculus

Polymorphic type syntax

• Extend type syntax with a forall type

\[ \tau ::= \ldots \mid \forall I. \tau \mid I \]

• Can write types of polymorphic values:

<table>
<thead>
<tr>
<th>id</th>
<th>( \forall T. T \rightarrow T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>( \forall T. \forall U. (T \rightarrow U) \rightarrow T \text{ list} \rightarrow T \text{ list} )</td>
</tr>
<tr>
<td>nil</td>
<td>( \forall T. T \text{ list} )</td>
</tr>
</tbody>
</table>

Polymorphic\(\lambda\) -calculus (System F)

• Extends simply-typed \( \lambda \):

- type syntax
- expression/value syntax
- typechecking rules
- evaluation rules

Polymorphic(ally typed) value syntax

• Syntax:

<table>
<thead>
<tr>
<th>E</th>
<th>( \ldots \mid \Lambda E \mid E[\tau] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>( \ldots \mid \Lambda E )</td>
</tr>
</tbody>
</table>

- \( \Lambda E \) is a function that, given a type \( \tau \), gives back \( E \) with \( \tau \) substituted for \( I \)
- Use such values by instantiating them: \( E[\tau] \)

• \( E[\tau] \) is like function application

An example

```latex
(* fun id x = x
  id:'a -> 'a *)
id = \Lambda T. \lambda x:T. x
\( \forall T. T \rightarrow T \)
```

id [int] 3 \( \rightarrow \beta \)
\( (\lambda x:int. x) 3 \rightarrow \beta \)
3

id [bool]
\( \lambda x:bool. x \)

Another example

```latex
(* fun applyTwice f x = f (f x)
  applyTwice: ('a->'a) -> 'a -> 'a *)
applyTwice =
\Lambda T. \Lambda T' \rightarrow T'. \lambda x:T. f (f x)
\( \forall T. (T \rightarrow T) \rightarrow T \rightarrow T \)
```

applyTwice [int] succ 3 \( \rightarrow \gamma \)
\( (\lambda x:int. x) \text{ succ 3} \rightarrow \gamma \)
succ (succ 3) \( \rightarrow \beta \)
5
Yet another example

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

map [int] [bool] isZero [3,0,5] \rightarrow \beta [false,true,false]

• ML infers what the \Lambda I and [\tau] should be

A final example

(* fun cool f = (f 3, f true) *)
cool = \lambda (\forall T. (T \rightarrow T) \rightarrow (T \rightarrow T))
(cool f) = (f 3, f true)

map [int] [bool] isZero [3,0,5] \rightarrow \beta [false,true,false]

• Note: \forall inside of \lambda and \rightarrow
– 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:
E \Rightarrow (\Lambda I. E) \Rightarrow V
\Rightarrow (E[\tau]) \Rightarrow V

• Typing:
\Gamma, I::Type
\Rightarrow \Lambda I. E :: \forall I. \tau
\Rightarrow \Gamma :: E :: \forall I. \tau'
\Rightarrow \Gamma :: (E[\tau]) :: \forall I. \tau'

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 \rightarrow, 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. \rightarrow, 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 ::= \text{int} | \text{int} \rightarrow \text{int} | \ldots

• A kind describes a set of types or type constructors with a common structure
  \kappa ::= * | \tau \Rightarrow \kappa

As in the s.t. \lambda calculus, * is the "base kind"

• Write \tau::\kappa to say that a type \tau has kind \kappa
  • \text{int}::* , \text{int}::{\tau \rightarrow \tau}
  • \text{list}::{* \rightarrow *\rightarrow *}
  • \text{assoc_list}::{* \rightarrow *\rightarrow *\rightarrow *\rightarrow *}
  • assoc_list string int::*
Kinded polymorphic $\lambda$-calculus
(System F$_\omega$)

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

Examples

```
pair =
\lambda T:*.*.\lambda U:*.*.\{first:T, second:U\}
:: *
P = \forall I::*.*.I::*

\forall I::*.*.I::*

pair int bool  "\rightarrow\beta" {first:int, second:bool}

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

Expression typing rules

```
\Gamma \vdash \tau_1 ::*  \quad \Gamma, I : \tau_1 \vdash E : \tau_2
\quad \quad \quad ---- [(T-ABS)]
\quad \Gamma, I : \tau_1 \vdash E : \tau_2
\Gamma, I : \tau_1 \vdash E : \tau_2
\quad \quad \quad ---- [(T-APP)]
\quad \Gamma, I : \tau_1 \vdash E : \tau_2
\Gamma, I : \tau_1 \vdash E : \tau_2
\quad \quad \quad ---- [(T-INST)]
\quad \Gamma, I : \tau_1 \vdash E : \tau_2
\quad \quad \quad ---- [(T-VAR)]
```

Higher-order kinds?

- Could “lift” polymorphism to type level...
  $\kappa ::= * | \forall I : \kappa | I$
  $\tau ::= \ldots | I \kappa I | I \kappa[t]$
- Could “lift” meta-kindng to kind level...
  $M ::= * | M \Rightarrow M$
  $\kappa ::= \ldots | \lambda \kappa : M \kappa | \kappa_1 \kappa_2$
- ...and so on to arbitrary “tower” of meta-levels of language

Type kinding rules

```
\Gamma \vdash \tau_1 ::*  \quad \Gamma, I :: \kappa \vdash E : \tau_2
\quad \quad \quad ---- [(K-INT)]
\quad \Gamma, I :: \kappa \vdash E : \tau_2
\Gamma, I :: \kappa \vdash E : \tau_2
\quad \quad \quad ---- [(K-ARROW)]
\quad \Gamma, I :: \kappa \vdash E : \tau_2
\Gamma, I :: \kappa \vdash E : \tau_2
\quad \quad \quad ---- [(K-FORALL)]
\quad \Gamma, I :: \kappa \vdash E : \tau_2
\quad \quad \quad ---- [(K-VAR)]
\quad \Gamma, I :: \kappa \vdash E : \tau_2
\Gamma, I :: \kappa \vdash E : \tau_2
\quad \quad \quad ---- [(K-ABS)]
```

Phase distinction

- Could also collapse all levels of language down to one:
  $E ::= I | \lambda I : E. E | E_1 E_2$
- 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 \( \lambda \)-calculus
  - Simply typed \( \lambda \)-calculus
  - Polymorphic \( \lambda \)-calculus, a.k.a. System F
  - Kinded poly. \( \lambda \)-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