fun ty (IFX (e1, e2, e3)) =
  if eqType (ty e1, booltype) then
    let val (tau2, tau3) = (ty e2, ty e3)
    in ... YOU FILL IN 1 ...
    end
  else
    ... YOU FILL IN 2 ...
  end

| ty (SET (x, e)) =
  let val tau_x = find (x, Gamma)
    val tau_e = ty e
  in ... YOU FILL IN 3 ...
  end
fun ty (APPLY (f, actuals)) = 
  let val atys = map ty actuals
  in  case ty f
       of FUNTY (formals, result) =>
          if eqTypes (atys, formals) then
              ... YOU FILL IN 4 ...
          else
              ... YOU FILL IN 5 ...
          | _   => ... YOU FILL IN 6 ...
  end
Monomorphic types are limiting

Each new type constructor requires

- Special syntax
- New type rules
- New internal representation (type formation)
- New code in type checker (intro, elim)
- New or revised proof of soundness
Monomorphic burden: Array types

Formation:  
\[
\text{\( \tau \) is a type} \\
\text{ARRAY(\( \tau \)) is a type}
\]

Introduction:  
\[
\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash \text{AMAKE}(e_1, e_2) : \text{ARRAY}(\tau)
\]

Elimination:  
\[
\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \\
\Gamma \vdash \text{AAT}(e_1, e_2) : \tau \\
\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash \text{APUT}(e_1, e_2, e_3) : \tau \\
\Gamma \vdash e : \text{ARRAY}(\tau) \\
\Gamma \vdash \text{ASIZE}(e) : \text{INT}
\]
Monomorphism hurts programmers too

Monomorphism leads to code duplication

User-defined functions are monomorphic:

```
(define int lengthI ([xs : (list int)])
  (if (null? xs) 0 (+ 1 (lengthI (cdr xs))))
)
(define int lengthB ([xs : (list bool)])
  (if (null? xs) 0 (+ 1 (lengthB (cdr xs))))
)
(define int lengthS ([xs : (list sym)])
  (if (null? xs) 0 (+ 1 (lengthS (cdr xs))))
)```
Quantified types

Heart of polymorphism: \( \forall \alpha_1, \ldots, \alpha_n . \tau. \)

In Typed \( \mu \)Scheme: (forall (‘a1 ... ‘an) type)

Two ideas:

- Type variable ‘a stands for an unknown type
- Quantified type (with forall) enables substitution

length : \( \forall \alpha . \alpha \text{ list} \to \text{int} \)

cons : \( \forall \alpha . \alpha \times \alpha \text{ list} \to \alpha \text{ list} \)

car : \( \forall \alpha . \alpha \text{ list} \to \alpha \)

cdr : \( \forall \alpha . \alpha \text{ list} \to \alpha \text{ list} \)

’() : \( \forall \alpha . \alpha \text{ list} \)
“Type variable”???

Back up here—what types do we have?
Type formation: Composing types

Typed Impcore:
  • Closed world (no new types)
  • Simple formation rules

Typed $\mu$Scheme:
  • Semi-closed world (new type variables)
  • How are types formed (from other types)?

Standard ML:
  • Open world (programmers create new types)
  • How are types formed (from other types)?

Can’t add new syntactic forms and new type formation rules for every new type.
Representing type constructors generically

Start with monomorphic fragment (Typed $\mu$Scheme):

\begin{verbatim}
datatype tyex
    = TYCON of name
    | CONAPP of tyex * tyex list
    | FUNTY of tyex list * tyex (* I’m special *)
\end{verbatim}

Examples: bool, (list int), (int int -> bool)

TYCON "bool"
CONAPP (TYCON "list", [TYCON "int"])
FUNTY ([TYCON "int", TYCON "int"], TYCON "bool")

Hard to read, but easy to write code for.
Well-formed types

We still need to classify type expressions into:

- **types** that classify terms (e.g., `int`)
- **type constructors** that build types (e.g., `list`)
- **nonsense** that means nothing (e.g., `int int`)

Idea: *kinds* classify types

\[ \Delta \vdash \tau :: \kappa \quad \text{“Type } \tau \text{ has kind } \kappa \text{”} \]

\[ \Delta \vdash \tau :: * \quad \text{Special case: “} \tau \text{ is a type”} \]

Replaces one-off type-formation rules

*Kind environment* \( \Delta \) tracks type constructors, vars
Return to quantified types

Heart of polymorphism: $\forall \alpha_1, \ldots, \alpha_n . \tau$.
In Typed $\mu$Scheme: (forall ('a1 ... 'an) type)

Two ideas:
• Type variable 'a stands for an unknown type
• Quantified type (with forall) enables substitution

\[
\begin{align*}
\text{length} & : \forall \alpha . \alpha \text{ list} \rightarrow \text{int} \\
\text{cons} & : \forall \alpha . \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \\
\text{car} & : \forall \alpha . \alpha \text{ list} \rightarrow \alpha \\
\text{cdr} & : \forall \alpha . \alpha \text{ list} \rightarrow \alpha \text{ list} \\
'() & : \forall \alpha . \alpha \text{ list}
\end{align*}
\]
Representing quantified types

Two new alternatives for \texttt{tyex}:

\begin{verbatim}
datatype tyex
    = TYCON of name
    | CONAPP of tyex * tyex list
    | FUNTY of tyex list * tyex
    | TYVAR of name
    | FORALL of name list * tyex
\end{verbatim}
Formation rules for quantified types

Reminder: $\Delta \vdash \tau :: *$ means “$\tau$ is a type”

$$\frac{\Delta \{ \alpha_1 :: *, \ldots, \alpha_n :: * \} \vdash \tau :: *}{\Delta \vdash \text{FORALL}([\alpha_1, \ldots, \alpha_n], \tau) :: *} \quad (\text{KIND}\text{ALL})$$

$$\frac{\alpha \in \text{dom} \Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \quad (\text{KIND}\text{INTRO}\text{VAR})$$

Example: $(\forall [\ 'a] \ ('a \to \ 'a))$
Programming with quantified types

Substitute for quantified variables

\[ \text{-> length} \]
\[ <\text{procedure}> : (\forall ('a) ((\text{list } 'a) \to \text{int})) \]
\[ \text{-> } (@ \text{ length } \text{int}) \]
\[ <\text{procedure}> : ((\text{list } \text{int}) \to \text{int}) \]
\[ \text{-> } (\text{length } '(1 \ 2 \ 3)) \]

type error: function is polymorphic; instantiate before applying

\[ \text{-> } ((@ \text{ length } \text{int}) ' (1 \ 2 \ 3)) \]
\[ 3 : \text{int} \]
Substitute what you like

-> length
<procedure> : (forall ('a) ((list 'a) -> int))
-> (@ length bool)
<procedure> : ((list bool) -> int)
-> (@ length bool) '(#t #f)
2 : int
More “Instantiations”

-> (val length-int (@ length int))
length-int : ((list int) -> int)
-> (val cons-bool (@ cons bool))
cons-bool : ((bool (list bool)) -> (list bool))
-> (val cdr-sym (@ cdr sym))
cdr-sym : ((list sym) -> (list sym))
-> (val empty-int (@ '() int))
() : (list int)
Create your own!

Abstract over unknown type using \texttt{type-lambda}

\[
\rightarrow \text{(val \ id \ (type-lambda \ ['a]

\text{(lambda \ ([x : 'a]) \ x ))})
\]

\text{id : (forall \ ('a) \ ('a -> 'a))}

\text{'a is type parameter (an \textit{unknown} type)}

This feature is \textit{parametric polymorphism}
Power comes at notational cost

Function composition

→ (val o (type-lambda ['a 'b 'c]
        (lambda ([f : ('b -> 'c)]
           [g : ('a -> 'b)])
           (lambda ([x : 'a]) (f (g x))))))

o : (forall ('a 'b 'c)
           (('b -> 'c) ('a -> 'b) -> ('a -> 'c)))

Aka o : ∀α,β,γ . (β → γ) × (α → β) → (α → γ)
Instantiate by substitution

∀ elimination:
  • Concrete syntax ($\forall e \, \tau_1 \ldots \tau_n$)
  • Rule (note new judgment form $\Delta, \Gamma \vdash e : \tau$):

\[
\Delta, \Gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n.\tau \\
\hline
\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]
\]

Substitution is in the book as function $\text{tysubst}$

(Also in the book: instantiate)
Generalize with type-lambda

∀ introduction:
- Concrete syntax \( \text{type-lambda} [\alpha_1 \cdots \alpha_n] e \)
- Rule (forall introduction):

\[
\Delta\{\alpha_1 :: *, \ldots \alpha_n :: *\}, \Gamma \vdash e : \tau \\
\alpha_i \not\in \text{ftv}(\Gamma), \quad 1 \leq i \leq n \\
\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1, \ldots, \alpha_n. \tau
\]

\( \Delta \) is kind environment (remembers \( \alpha_i \)'s are types)
A phase distinction embodied in code

\[
\rightarrow (\text{val } x \ 3)
\]
\[
3 : \text{int}
\]
\[
\rightarrow (\text{val } y \ (+\ x\ x))
\]
\[
6 : \text{int}
\]

fun processDef (d, (delta, gamma, rho)) =
  let val (gamma’, tystring) = elabdef (d, gamma, delta)
    val (rho’, valstring) = evaldef (d, rho)
    val _ = print (valstring ^ " : " ^ tystring)
  in  (delta, gamma’, rho’) 
end
Return to well-formed types

To classify type expressions into:

- **types** that classify terms (e.g., int)
- **type constructors** that build types (e.g., list)
- **nonsense** that means nothing (e.g., int int)

Use judgment

\[ \Delta \vdash \tau :: \kappa \]
Type formation through kinds

Each type constructor has a kind.

Type constructors of kind \( * \) classify terms

\[(\text{int} :: *, \text{bool} :: *)\]

\( * \) is a kind

Type constructors of arrow kinds are “types in waiting”

\[(\text{list} :: * \Rightarrow *, \text{pair} :: * \times * \Rightarrow *)\]

\( \kappa_1, \ldots, \kappa_n \) are kinds \( \kappa \) is a kind

\( \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \) is a kind

\( (\text{KINDFORMATIONTYPE}) \)

\( (\text{KINDFORMATIONARROW}) \)
Use kinds to give arities

Examples: int :: *, list :: * ⇒ *, pair :: * × * ⇒ *

Non-Examples: int int and bool × list have no kind because they are nonsense.

*Kinds classify type expressions just as types classify terms*
The kinding judgment

\[ \Delta \vdash \tau :: \kappa \quad \text{“Type } \tau \text{ has kind } \kappa \text{”} \]

\[ \Delta \vdash \tau :: * \quad \text{Special case: “} \tau \text{ is a type”} \]

Replaces one-off type-formation rules

*Kind environment* \( \Delta \) tracks type constructor names and kinds.
Kinding rules for types

\[
\begin{align*}
\mu & \in \text{dom } \Delta \quad \Delta(\mu) = \kappa \\
\Delta & \vdash \text{TYCON}(\mu) :: \kappa \quad \text{KindIntroCon} \\
\Delta & \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \\
\Delta & \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n \\
\Delta & \vdash \text{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa \quad \text{KindApp}
\end{align*}
\]

These two rules replace all formation rules.

(Check out book functions kindof and asType)
Kinds of primitive type constructors

\[ \Delta(\text{int}) = * \]
\[ \Delta(\text{bool}) = * \]
\[ \Delta(\text{list}) = * \Rightarrow * \]
\[ \Delta(\text{option}) = * \Rightarrow * \]
\[ \Delta(\text{pair}) = * \times * \Rightarrow * \]
\[ \Delta(\text{queue}) = \text{You fill in} \]
\[ \Delta(\text{unit}) = \text{You fill in} \]
Three environments — what happens?

Δ  maps names (of tycons and tyvars) to kinds
Γ  maps names (of variables) to types
ρ  maps names (of variables) to values or locations

New val def
val x = 33

New type def
type 'a transformer = 'a -> 'a

New datatype def
datatype color = RED | GREEN | BLUE
Three environments revealed

$\Delta$ maps names (of tycons and tyvars) to kinds

$\Gamma$ maps names (of variables) to types

$\rho$ maps names (of variables) to values or locations

**New val def modifies** $\Gamma, \rho$

val $x = 33$ means $\Gamma\{x : \text{int}\}, \rho\{x \mapsto 33\}$

**New type def modifies** $\Delta$

type `'a transformer = `'a list * `'a list

means $\Delta\{\text{transformer} :: * \Rightarrow *\}$

**New datatype def modifies** $\Delta, \Gamma, \rho$

datatype color = RED | GREEN | BLUE

means $\Delta\{\text{color} :: *\}, \Gamma\{\text{RED} : \text{color}, \text{GREEN} : \text{color}, \text{BLUE} : \text{color}\},$

$\rho\{\text{RED} \mapsto 0, \text{GREEN} \mapsto 1, \text{BLUE} \mapsto 2\}$
Exercise: Three environments

datatype 'a tree
    = NODE of 'a tree * 'a * 'a tree
    | EMPTY

means

\[ \Delta\{ \text{tree} \to * \Rightarrow * \} , \]
\[ \Gamma\{ \text{NODE} \to \forall 'a . 'a \text{ tree} * 'a * 'a \text{ tree} \to 'a \text{ tree} , \]
\[ \quad \text{EMPTY} \to \forall 'a . 'a \text{ tree} \} , \]
\[ \rho\{ \text{NODE} \to \lambda(l, x, r) . \cdots , \text{EMPTY} \to 1 \} \]