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

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

Elimination: \[
\frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash \text{AAT}(e_1, e_2) : \tau}.
\]

\[
\frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{APUT}(e_1, e_2, e_3) : \tau}.
\]

\[
\frac{\Gamma \vdash e : \text{ARRAY}(\tau)}{\Gamma \vdash \text{ASIZE}(e) : \text{INT}}.
\]
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 ('al ... 'an) type)

Two ideas:

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

length : $\forall \alpha . \alpha$ list $\rightarrow$ int

cons : $\forall \alpha . \alpha \times \alpha$ list $\rightarrow$ $\alpha$ list

car : $\forall \alpha . \alpha$ list $\rightarrow$ $\alpha$

cdr : $\forall \alpha . \alpha$ list $\rightarrow$ $\alpha$ list

'() : $\forall \alpha . \alpha$ 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

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):

```plaintext
datatype tyex
  = TYCON of name
  | CONAPP of tyex * tyex list
  | FUNTY of tyex list * tyex (* I’m special *)

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

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

Hard to read, but easy to write code for.
Question: How would you represent an array of pairs of booleans?

datatype tyex
    = TYCON of name
    | CONAPP of tyex * tyex list
    | FUNTY of tyex list * tyex

(bool * bool) array ML
(array (pair bool bool)) Typed μScheme
Question: How would you represent an array of pairs of booleans?

```
datatype tyex
  = TYCON   of name
  | CONAPP of tyex * tyex list
  | FUNTY   of tyex list * tyex

(bool * bool) array       ML
(array (pair bool bool))  Typed μScheme

CONAPP (TYCON "array",
  [ CONAPP (TYCON "pair",
    [TYCON "bool", TYCON "bool"])
  ])
```
Well-formed types

We still need to classify type expressions into:
- types that classify terms (e.g., \texttt{int})
- type constructors that build types (e.g., \texttt{list})
- nonsense that means nothing (e.g., \texttt{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

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

Two new alternatives for tyex:

datatype tyex
  = TYCON of name
  | CONAPP of tyex * tyex list
  | FUNTY of tyex list * tyex
  | TYVAR of name
  | FORALL of name list * tyex
Formation rules for quantified types

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

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

\[
\frac{\alpha \in \text{dom}\Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \quad \text{(KIND\text{INTROVAR})}
\]
Programming with quantified types

Substitute for quantified variables

-> length
<procedure> : (forall ('a) ((list 'a) -> int))
-> (@ length int)
<procedure> : ((list int) -> int)
-> (length '(1 2 3))
  type error: function is polymorphic; instantiate before applying
-> (((@ length int) '(1 2 3))
  3 : 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 type-lambda

→ (val id (type-lambda ['a]
  (lambda ([x : 'a]) x )))

id : (forall ('a) ('a -> 'a))

'a is type parameter (an unknown type)

This feature is parametric polymorphism
Power comes at notational cost

Function composition

\[ \rightarrow \text{val } o \text{ (type-lambda } [\text{'a } \text{'b } \text{'c}] \]

\[ (\lambda ([f : (\text{'b } \rightarrow \text{'c}] ]

\[ [g : (\text{'a } \rightarrow \text{'b}] )]

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

\[ o : (\text{forall } [\text{'a } \text{'b } \text{'c)}

\[ ((\text{'b } \rightarrow \text{'c) } (\text{'a } \rightarrow \text{'b) } \rightarrow (\text{'a } \rightarrow \text{'c})) \]

Aka \[ o : \forall \alpha , \beta , \gamma . (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \]
Instantiate by substitution

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

$$\Delta, \Gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n.\tau$$

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

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

(Also in the book: $\text{instantiate}$)
Generalize with type-lambda

∀ introduction:
  • Concrete syntax (type-lambda [α₁ ⋯ αₙ] e)
  • Rule:

\[ \Delta\{\alpha₁ :: *, \ldots, \alphaₙ :: *\}, \Gamma \vdash e : \tau \]
\[ \alphaᵢ \not\in \text{ftv}(\Gamma), \quad 1 \leq i \leq n \]
\[ \Delta, \Gamma \vdash \text{TYLAMBDINIT}(\alpha₁, \ldots, \alphaₙ, e) : \forall \alpha₁, \ldots, \alphaₙ.\tau \]

(FORALL INTRODUCTION)

\[ \Delta \text{ is kind environment (remembers } \alphaᵢ\text{'s are types) } \]
A phase distinction embodied in code

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

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 $\times$ classify terms

$(\text{int} :: * , \text{bool} :: *)$

$\star$ 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 \hspace{1em} $\kappa$ 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

\[ \mu \in \text{dom} \Delta \]

\[ \frac{\Delta(\mu) = \kappa}{\Delta \vdash \text{TYCON}(\mu) :: \kappa} \]  

(KINDINTROCON)

\[ \Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \]

\[ \frac{\Delta \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n}{\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa} \]  

(KINDAPP)

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?

- \( \Delta \) 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**

```
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** \(\Gamma, \rho\)

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

**New type def modifies** \(\Delta\)

\[
\text{type } 'a\ \text{transformer} = 'a\ \text{list} \ast 'a\ \text{list} \text{ means } \Delta\{\text{transformer} :: \ast \Rightarrow \ast\}
\]

**New datatype def modifies** \(\Delta, \Gamma, \rho\)

\[
\text{datatype color} = \text{RED} | \text{GREEN} | \text{BLUE} \text{ means } \Delta\{\text{color} :: \ast\}, \Gamma\{\text{RED} : \text{color}, \text{GREEN} : \text{color}, \text{BLUE} : \text{color}\}, \\
\rho\{\text{RED} \mapsto 0, \text{GREEN} \mapsto 1, \text{BLUE} \mapsto 2\}
\]
Exercise: Three environments

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

means

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