Type checking in ML (no variables!)

val typeof : exp -> ty
exception IllTypeded

fun typeof (ARITH (_, e1, e2)) =
  (case (typeof e1, typeof e2)
   of (INTTY, INTTY) => INTTY
    | _ => raise IllTyped)

| typeof (CMP (_, e1, e2)) =
  (case (typeof e1, typeof e2)
   of (INTTY, INTTY) => BOOLTY
    | _ => raise IllTyped)

| typeof (LIT _) = INTTY

| typeof (IF (e,e1,e2)) =
  (case (typeof e, typeof e1, typeof e2)
   of (BOOLTY, tau1, tau2) =>
     if eqType (tau1, tau2)
     then tau1 else raise IllTyped
    | _ => raise IllTyped)
Let’s add variables!

datatype exp = ARITH of arithop * exp * exp
    | CMP of relop * exp * exp
    | LIT of int
    | IF of exp * exp * exp
    | VAR of name
    | LET of name * exp * exp

and arithop = PLUS | MINUS | TIMES | ...

and relop = EQ | NE | LT | LE | GT | GE

datatype ty = INTTY | BOOLTY
Examples: Well-formed types

These are types:

• int
• bool
• int * bool
• int * int -> int
Examples: Not yet types, or not types at all

These “types in waiting” don’t classify any terms
  • list (but int list is a type)
  • array (but char array is a type)
  • ref (but (int -> int) ref is a type)

These are utter nonsense
  • int int
  • bool * array
Type-formation rules

We need a way to classify type expressions into:

- **types** that classify terms
- **type constructors** that build types
- **nonsense** that doesn’t mean anything
Type constructors

Technical name for “types in waiting”

Given zero or more arguments, produce a type:
- Nullary int, bool, char also called base types
- Unary list, array, ref
- Binary (infix) \( \rightarrow \)

More complex type constructors:
- records/structs
- function in C, uScheme, Impcore
What’s a good type? (Type formation)

Type formation rules for Typed Impcore

\[ \tau \in \{ \text{UNIT, INT, BOOL} \} \]

(\text{BASETYPES})

\[ \tau \text{ is a type} \]

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

(\text{ARRAYFORMATION})
Type judgments for monomorphic system

Two judgments:

- The familiar typing judgment $\Gamma \vdash e : \tau$
- Today’s judgment “$\tau$ is a type”
Type rules for variables

Lookup the type of a variable:

\[
\frac{x \in \text{dom} \Gamma \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}
\]  
\text{(VAR)}

Types match in assignment (two \( \tau \)'s must be equal):

\[
\frac{x \in \text{dom} \Gamma \quad \Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{SET}(x, e) : \tau}
\]  
\text{(SET)}
Type rules for control

Boolean condition; matching branches

\[
\Gamma \vdash e_1 : \text{BOOL} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau
\]

\[
\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau
\]

(IF)
Product types: Both x and y

New abstract syntax: PAIR, FST, SND

\[ \tau_1 \text{ and } \tau_2 \text{ are types} \]
\[ \frac{\tau_1 \times \tau_2 \text{ is a type}}{\tau_1 \times \tau_2} \]
\[ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2} \]
\[ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{SND}(e) : \tau_2} \]

Pair rules generalize to product types with many elements ("tuples," "structs," and "records")
Arrow types: Function from $x$ to $y$

Syntax: $\texttt{lambda}$, application

Typed $\mu$Scheme style:

$$\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{(\tau_1 \cdots \tau_n \to \tau) \text{ is a type}}$$

(\textsc{arrowformation})

ML style: functions takes a tuple:

$$\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{\tau_1 \times \cdots \times \tau_n \to \tau \text{ is a type}}$$

(\textsc{mlarrowformation})
Arrow types: Function from \( x \) to \( y \)

Eliminate with application:

\[
\Gamma \vdash e : (\tau_1 \cdots \tau_n \rightarrow \tau)
\]

\[
\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n
\]

\[
\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau
\]

Introduce with \textbf{lambda}:

\[
\Gamma \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau
\]

\[
\Gamma \vdash \text{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)
\]
Typical syntactic support for types

Explicit types on lambda and define:

• For lambda, argument types:
  \[(\text{lambda } ([n : \text{int}] [m : \text{int}]) (+ (* n n) (* m m)))\]

• For define, argument and result types:
  \[(\text{define int max } ([x : \text{int}] [y : \text{int}])
  (if (< x y) y x))\]

Abstract syntax:

datatype exp = ...
  | LAMBDA of (name * tyex) list * exp
  ...

datatype def = ...
  | DEFINE of name * tyex * ((name * tyex) list * exp)
  ...
Array types: Array of $x$

**Formation:**

\[
\begin{array}{c}
\text{\(\tau\) is a type} \\
\hline
\text{ARRAY(\(\tau\)) is a type}
\end{array}
\]

**Introduction:**

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \tau \\
\hline
\Gamma \vdash \text{AMAKE}(e_1, e_2) : \text{ARRAY(\(\tau\))}
\end{array}
\]
Array types continued

Elimination:

$$\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT}$$

$$\Gamma \vdash \text{AAT}(e_1, e_2) : \tau$$

$$\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau$$

$$\Gamma \vdash \text{APUT}(e_1, e_2, e_3) : \tau$$

$$\Gamma \vdash e : \text{ARRAY}(\tau)$$

$$\Gamma \vdash \text{ASIZE}(e) : \text{INT}$$
References (similar to C/C++ pointers)

Your turn! Given

\[ \text{ref } \tau \quad \text{REF}(\tau) \]

\[ \text{ref } e \quad \text{REF-MAKE}(e) \]

\[ !e \quad \text{REF-GET}(e) \]

\[ e_1 := e_2 \quad \text{REF-SET}(e_1, e_2) \]

Write formation, introduction, and elimination rules.
Wait for it . . .
Reference Types

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

Introduction: \( \Gamma \vdash e : \tau \)
\[ \frac{}{\Gamma \vdash \text{REF-MAKE}(e) : \text{REF}(\tau)} \]

Elimination: \( \Gamma \vdash e : \text{REF}(\tau) \)
\[ \frac{}{\Gamma \vdash \text{REF-GET}(e) : \tau} \]
\[ \frac{\Gamma \vdash e_1 : \text{REF}(\tau) \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{REF-SET}(e_1, e_2) : \tau} \]
From rule to code

**Arrow-introduction**

\[
\frac{\Gamma \{ x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \} \vdash e : \tau \quad \tau_i \text{ is a type, } 1 \leq i \leq n}{\Gamma \vdash \text{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)}
\]
Type-checking LAMBDA

datatype exp = LAMBDA of (name * tyex) list * exp
...

fun ty (Gamma, LAMBDA (formals, body)) =
  let val Gamma' = (* body gets new env *)
      foldl (fn ((x, ty), g) => bind (x, ty, g))
         Gamma formals
  in
    val bodytype = ty (Gamma', body)
    fun formaltypes =
      map (fn (x, ty) => ty) formals
    in
      FUNTY (formal/types, bodytype)
  end