Type systems

What they do:
• Guide coding
• Document code (checked by compiler!)
• Rule out certain errors

How they work
• Predict values at run time

World’s most widely deployed static analysis
“Types classify terms”

“Term” is theory word for “syntax”:

\[ n + 1 : \text{int} \]

\[ "\text{hello}\" \wedge "\text{world}\" : \text{string} \]

\[ (\text{fn } n \Rightarrow n \times (n - 1)) : \text{int} \rightarrow \text{int} \]

\[ \text{if } p \text{ then } 1 \text{ else } 0 : \text{int}, \text{ provided } p : \text{bool} \]
Type soundness

Key theorem: *prediction is accurate*

- Will state more precisely next week
- Best explanation for how/why type system works
- Proof beyond the scope of 105
Type-system example

Simple language of machine-level expressions

Two types:

- word predicts a machine word
  (in a general-purpose register)
- flag predicts a single bit
  (in a flags register)
Type this: Language of expressions

Words and flags:

```haskell
datatype exp = ARITH of arithop * exp * exp
           | CMP of relop     * exp * exp
           | LIT of int
           | IF   of exp      * exp * exp

and arithop = PLUS | MINUS | TIMES | ...
and relop    = EQ   | NE   | LT   | LE   | GT   | GE
```

datatype ty = WORDTY | FLAGTY

(Looks a lot like int and bool)
Type checking in ML (no variables!)

val typeof : exp -> ty
exception IllTyped
fun typeof (ARITH (_, e1, e2)) =
  (case (typeof e1, typeof e2)
     of (WORDTY, WORDTY) => WORDTY
     | _              => raise IllTyped)

| typeof (CMP (_, e1, e2)) =
  (case (typeof e1, typeof e2)
     of (WORDTY, WORDTY) => FLAGTY
     | _              => raise IllTyped)

| typeof (LIT _) = WORDTY
| typeof (IF (e,e1,e2)) =
  (case (typeof e, typeof e1, typeof e2)
     of (FLAGTY, 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 = WORDTY | FLAGTY
Type checking for variables

```ocaml
val typeof : exp * ty env -> ty
fun typeof (ARITH ..., Gamma) = <as before>
  | typeof (VAR x, Gamma) =
      (case maybeFind (x, Gamma)
          of SOME tau => tau
          | NONE => raise IllTyped)
  | typeof (LET (x, e1, e2), Gamma) =
      let tau1 = typeof (e1, Gamma) in
      typeof (e2, extend Gamma x tau1) end
```