Last time: Type-directed coding

Common idea in functional programming: “lifting”

val lift : forall 'a . ('a -> bool) -> ('a list -> bool)

fun lift p [] = false
   | lift p (z::zs) = p z orelse lift p zs
# Types and their C constructs

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduce</td>
<td></td>
<td>Eliminate</td>
</tr>
</tbody>
</table>

- **“initializer”**: `struct { ..., ... }`
- **dot notation**: `e.next, e->next`
- **pointer**: `&`, `*`
- **function (definition form)**: application
Types and their ML constructs

Type | Produce | Consume
--- | --- | ---
| Introduce | Lambda \( \text{(fn)} \) | Eliminate

arrow | Application

constructed \( \text{(algebraic)} \) | Apply constructor

constructed \( \text{(tuple)} \) | \( (e_1, \ldots, e_n) \) | Pattern match!
Type this: Language of expressions

Numbers and Booleans:

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 = INTTY | BOOLTY

Problem to solve: integer register or flags register?
Type checking in ML (no variables!)

val typeof : exp -> ty
exception IllTyped
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)
Type checking in ML (no variables!)

```ml
val typeof : exp -> ty
exception IllTyped

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

\[ \tau \in \{ \text{UNIT, INT, BOOL} \} \]
\[ \Rightarrow \tau \text{ is a type} \]  \hspace{2cm} (\text{BASETYPES})

\[ \tau \text{ is a type} \]
\[ \Rightarrow \text{ARRAY}(\tau) \text{ is a type} \] \hspace{2cm} (\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:

\[ x \in \text{dom} \Gamma \quad \Gamma(x) = \tau \]

\[ \Gamma \vdash x : \tau \quad \text{(VAR)} \]

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

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

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

Boolean condition; matching branches

\[
\frac{\Gamma \vdash e_1 : \text{BOOL} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau}
\] (IF)
Product types: Both x and y

New abstract syntax: PAIR, FST, SND

Γ ⊢ e₁ : τ₁  Γ ⊢ e₂ : τ₂
Γ ⊢ PAIR(e₁, e₂) : τ₁ × τ₂

Γ ⊢ e : τ₁ × τ₂
Γ ⊢ FST(e) : τ₁
Γ ⊢ SND(e) : τ₂

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:

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

ML style: functions takes a tuple:

$$\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{\tau_1 \times \cdots \times \tau_n \rightarrow \tau \text{ is a type}} \quad (\text{MLTARROWFORMATION})$$
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 \texttt{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:
  \[
  \text{(lambda ([n : int] [m : int]) (+ (* n n) (* m m))}
  \]

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

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

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

Elimination:

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

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

\[
\frac{\Gamma \vdash e : \text{ARRAY}(\tau)}{\Gamma \vdash \text{ASIZE}(e) : \text{INT}}
\]
References (similar to C/C++ pointers)

Your turn! Given

\[
\begin{align*}
\text{ref } \tau & \quad \text{REF}(\tau) \\
\text{ref } e & \quad \text{REF-MAKE}(e) \\
!e & \quad \text{REF-GET}(e) \\
e1 := e2 & \quad \text{REF-SET}(e1, e2)
\end{align*}
\]

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

**Formation:**

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

**Introduction:**

\[
\begin{array}{c}
\Gamma \vdash e : \tau \\
\overline{\Gamma \vdash \text{REF-MAKE}(e) : \text{REF}(\tau)}
\end{array}
\]

**Elimination:**

\[
\begin{array}{c}
\Gamma \vdash e : \text{REF}(\tau) \\
\overline{\Gamma \vdash \text{REF-GET}(e) : \tau}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{REF}(\tau) \\
\Gamma \vdash e_2 : \tau \\
\overline{\Gamma \vdash \text{REF-SET}(e_1,e_2) : \tau}
\end{array}
\]

From rule to code

Arrow-introduction

\[
\Gamma \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \quad \tau_i \text{ is a type, } 1 \leq i \leq n
\]

\[
\Gamma \vdash \text{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)
\]
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)
      val formaltypes = map (fn (x, ty) => ty) formals
      in
      FUNTY (formaltypes, bodytype)
      end
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:**
\[ \tau \text{ is a type} \]
\[ \text{ARRAY}(\tau) \text{ is a type} \]

**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

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

```
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"])
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

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

Two new alternatives for $\text{tyex}$:

```plaintext
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: $\Delta \vdash \tau :: *$ means "$\tau$ is a type"

\[
\Delta \{ \alpha_1 :: *, \ldots, \alpha_n :: * \} \vdash \tau :: *
\]

$(\text{KindAll})$

\[
\Delta \vdash \text{FORALL}(\left[\alpha_1, \ldots, \alpha_n\right], \tau) :: *
\]

$(\text{KindIntroVar})$

\[
\alpha \in \text{dom} \Delta \quad \frac{}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)}
\]

Example: $\text{(forall } [\texttt{a}] \ (\texttt{a} \rightarrow \texttt{a})\text{)}$
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`:

\[
\rightarrow \ \text{(val id (type-lambda [\text{'}a]\n\quad \text{(lambda ([x : \text{'}a]) x }]))} \\
\text{id : (forall (\text{'}a) (\text{'}a -> \text{'}a))}
\]

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

This feature is 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

\( \forall \) 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: instantiate)
Generalize with type-lambda

\( \forall \) introduction:

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

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

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

-> (val x 3)
3 : int
-> (val y (+ x x))
6 : 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 \( \ast \) classify terms

\( (\text{int} :: \ast, \text{bool} :: \ast) \)

\( \ast \) is a kind

Type constructors of arrow kinds are “types in waiting”

\( (\text{list} :: \ast \Rightarrow \ast, \text{pair} :: \ast \times \ast \Rightarrow \ast) \)

\( \kappa_1, \ldots, \kappa_n \) 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 :: * \Rightarrow *, pair :: * \times * \Rightarrow *

Non-Examples: int int and bool \times 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

\[
\begin{align*}
\mu & \in \text{dom} \Delta & \Delta(\mu) & = \kappa \\
\Delta & \vdash \text{TYCON}(\mu) :: \kappa & & \text{KINDINTROCON} \\
\end{align*}
\]

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

These two rules replace all formation rules.

(Check out book functions \texttt{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 \texttt{val} def modifies \( \Gamma, \rho \)
\[
\text{val } x = 33 \text{ means } \Gamma\{x : \text{int}\}, \rho\{x \mapsto 33\}
\]

New \texttt{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 \texttt{datatype} def modifies \( \Delta, \Gamma, \rho \)
\[
\text{datatype color} = \text{RED} \mid \text{GREEN} \mid \text{BLUE} \\
\text{means } \Delta\{\text{color} :: \ast\}, \Gamma\{\text{RED} : \text{color}, \text{GREEN} : \text{color}, \text{BLUE} : \text{color}\}, \rho\{\text{RED} \mapsto 0, \text{GREEN} \mapsto 1, \text{BLUE} \mapsto 2\}
\]
Exercise: Three environments

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

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