Programs as data
first-order functional language type checking

Micro-ML: A small functional language

- First-order: A value cannot be a function
- Dynamically typed, so this is OK:

  `if true then 1+2 else 1+false`

- Eager, or call-by-value: In a call **`f(e)`** the argument **`e`** is evaluated before **`f`** is called
- Example Micro-ML programs (an F# subset):

```
5+7

let f x = x + 7 in f 2 end

let fac x = if x=0 then 1 else x * fac(x - 1)
in fac 10 end
```

Abstract syntax of Micro-ML

```
type expr =
   | CstI of int
   | CstB of bool
   | Var of string
   | Let of string * expr * expr
   | Prim of string * expr * expr
   | If of expr * expr * expr
   | Letfun of string * string * expr * expr
   | Call of expr * expr
```

```
let f x = x + 7 in f 2 end
```

(f, x, fBody, letBody)

```
Letfun ("f", "x", Prim ("+", Var "x", CstI 7),
          Call (Var "f", CstI 2))
```

Runtime values, function closures

- Run-time values: integers and functions

```
type value =
  | Int of int
  | Closure of string * string * expr * value env
```

- *Closure*: a package of a function's body and its declaration environment
- A name should refer to a *statically* enclosing binding:

```
let y = 11
in let f x = x + y
   in let y = 22 in f 3 end
   end
end
```

Should always have value 11

Evaluate as 3 + y

(f, x, x+y, [(y,11)])

Interpretation of Micro-ML

- Constants, variables, primitives, let, if: as for expressions
- Letfun: Create function closure and bind f to it
- Function call f(e):
  - Look up f, it must be a closure
  - Evaluate e
  - Create environment and evaluate the function’s body

```
let rec eval (e : expr) (env : value env) : int =
  match e with
  | Letfun (f, x, e1, e2) ->
    let env2 = (f, Closure(f, x, e1, env)) :: env in
    eval e2 env2
  | ...
  | Call (Var f, e) ->
    let c = lookup env f in
    match c with
    | Closure (f, x, b, fenv) ->
      let v = Int (eval e env) in
      let envf = (x, v) :: (f, c) :: fenv in
      eval b envf
| _ -> failwith "eval Call: not a function"
```

Evaluate fBody in declaration environment

Evaluation by logical rules

$$\frac{}{\rho \vdash i \Rightarrow i}\,(e1)$$

$$\frac{}{\rho \vdash b \Rightarrow b}\,(e2)$$

$$\frac{\rho(x) = v}{\rho \vdash x \Rightarrow v}\,(e3)$$

In environment ρ, expression x evaluates to *v*

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v = v_1 + v_2}{\rho \vdash e_1 + e_2 \Rightarrow v}\,(e4)$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad b = (v_1 < v_2)}{\rho \vdash e_1 < e_2 \Rightarrow b}\,(e5)$$

$$\frac{\rho \vdash e_r \Rightarrow v_r \qquad \rho[x \mapsto v_r] \vdash e_b \Rightarrow v}{\rho \vdash \mathtt{let}\ x = e_r\ \mathtt{in}\ e_b\ \mathtt{end} \Rightarrow v}\,(e6)$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathtt{true} \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow v}\,(e7t)$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathtt{false} \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow v}\,(e7f)$$

Evaluation by logical rules: Function declaration and call

- Compare these with the **`eval`** interpreter:

$$\frac{\rho[f \mapsto (f, x, e_r, \rho)] \vdash e_b \Rightarrow v}{\rho \vdash \mathbf{let}\ f\ (x) = e_r\ \mathbf{in}\ e_b\ \mathbf{end}\ \Rightarrow v}\ (e8)$$

$$\frac{\rho(f) = (f, x, e_r, \rho_{fdecl}) \qquad \rho \vdash e \Rightarrow v_x \qquad \rho_{fdecl}[x \mapsto v_x, f \mapsto (f, x, e_r, \rho_{fdecl})] \vdash e_r \Rightarrow v}{\rho \vdash f\ e \Rightarrow v}\ (e9)$$

- Also, note recursive evaluation of f's body

Dynamic scope (instead of static)

- With static scope, a variable refers to the lexically, or statically, most recent binding
- With **dynamic scope**, a variable refers to the dynamically most recent binding:

```
let y = 11
in let f x = x + y
   in let y = 22 in f 3 end
   end
end
```

Evaluate as
`3 + y`

A dynamic scope variant of Micro-ML

- Very minimal change in interpreter:

```
let rec eval (e : expr) (env : value env) : int =
    ...
    | Call(Var f, eArg) ->
      let fClosure = lookup env f
      in match fClosure with
         | Closure (f, x, fBody, fDeclEnv) ->
           let xVal = Int(eval eArg env)
           let fBodyEnv = (x, xVal) :: (f, fClosure) :: env
           in eval fBody fBodyEnv
```

Evaluate fBody in call environment

- fDeclEnv is ignored; function is just (f, x, fBody)
- Good and bad:
  - simple to implement (no closures needed)
  - makes type checking difficult
  - makes efficient implementation difficult
- Used in macro languages, and Lisp, Perl, Clojure

Lexer and parser for Micro-ML

- Lexer:
  - Nested comments, as in F#, Standard ML

```
1 + (* 33 (* was 44 *) *) 22
```

- Parser:
  - To parse applications e1 e2 e3 correctly, distinguish atomic expressions from others

- Problem: f(x-1) parses as f(x(-1))
- Solution:
  - FunLex.fsl: make **`CSTINT`** just **`[0-9]+`** without sign
  - FunPar.fsy: add rule **`Expr := MINUS Expr`**

An explicitly typed fun. language

```
let f (x : int) : int = x+1
in f 12 end
```

```
type typ =
   | TypI
   | TypB
   | TypF of typ * typ
```

```
Letfun("f", "x", TypI,
       Prim("+", Var "x", CstI 1), TypI,
       Call(Var "f", CstI 12));;
```

TypF (TypI, TypI)

```
type tyexpr =
  | CstI of int
  | CstB of bool
  | Var of string
  | Let of string * tyexpr * tyexpr
  | Prim of string * tyexpr * tyexpr
  | If of tyexpr * tyexpr * tyexpr
  | Letfun of string * string * typ * tyexpr * typ * tyexpr
  | Call of tyexpr * tyexpr
```

(f, x, xT, b, bT, letb)

Type checking by recursive function

- Using a type environment [(“x”, TypI)]:

```
let rec typ (e : tyexpr) (env : typ env) : typ =
    match e with
    | CstI i -> TypI
    | CstB b -> TypB
    | Var x  -> lookup env x
    | Prim(op, e1, e2) ->
      let t1 = typ e1 env
      let t2 = typ e2 env
      in match (op, t1, t2) with
         | ("*",  TypI, TypI) -> TypI
         | ("+",  TypI, TypI) -> TypI
         | ("-",  TypI, TypI) -> TypI
         | ("=",  TypI, TypI) -> TypB
         | ("<",  TypI, TypI) -> TypB
         | ("&&", TypB, TypB) -> TypB
         | _    -> failwith "unknown primitive, or type error"
    | ...
```

Type checking, part 2

- Checking **`let x=eRhs in letBody end`**
- Checking **`if e1 then e2 else e3`**

```
let rec typ (e : tyexpr) (env : typ env) : typ =
    match e with
    | Let(x, xE, b) ->
      let xT = typ xE env in
      typ b ((x, xT) :: env)
    | If(e1, e2, e3) ->
      match typ e1 env with
       | TypB -> let t2 = typ e2 env in
                 let t3 = typ e3 env in
                 if t2 = t3 then t2
                 else failwith "If: branch types differ"
       | _    -> failwith "If: condition not boolean"
    | ...
```

Type checking, part 3

- Checking **let** f x **=** fB **in** letB **end**
- Checking f eA

```
let rec typ (e : tyexpr) (env : typ env) : typ =
    match e with
    | ...
    | Letfun(f, x, xT, fB, bT, letB) ->
      let fT = TypF(xT, bT) in
      let fBE = (x, xT) :: (f, fT) :: env in
      let letBE = (f, fT) :: env in
      if typ fB fBE = rT then typ letB letBE
      else failwith "Letfun: wrong return type in function"

    | Call(Var f, eA) ->
      match lookup env f with
      | TypF(xT, bT) ->
        if typ eA env = xT then bT
        else failwith "Call: wrong argument type"
      | _ -> failwith "Call: unknown function"
    | Call(_, _) -> failwith "Call: illegal function in call"
```

Type checking versus evaluation

- The type checker **`typ`** and the interpreter **`eval`** have similar structure
- Type checking can be thought of as *abstract interpretation* of the program
- We calculate "TypI + TypI gives TypI" instead of "Int 3 + Int 5 gives Int 8"
- One major difference:
  - Type checking a function call f(e) does not require type checking the function's body again
  - Interpreting a function call f(e) does require interpreting the function's body
- Type checking always terminates

Type checking by logical rules

$$\rho \vdash i : \texttt{int}$$

$$\rho \vdash b : \texttt{bool}$$

$$\frac{\rho(x) = t}{\rho \vdash x : t}$$

$$\frac{\rho \vdash e_1 : \texttt{int} \qquad \rho \vdash e_2 : \texttt{int}}{\rho \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\rho \vdash e_1 : \texttt{int} \qquad \rho \vdash e_2 : \texttt{int}}{\rho \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\rho \vdash e_r : t_r \qquad \rho[x \mapsto t_r] \vdash e_b : t}{\rho \vdash \texttt{let}\ x = e_r\ \texttt{in}\ e_b\ \texttt{end}\ : t}$$

$$\frac{\rho \vdash e_1 : \texttt{bool} \qquad \rho \vdash e_2 : t \qquad \rho \vdash e_3 : t}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t}$$

$$\frac{\rho[x \mapsto t_x, f \mapsto t_x \to t_r] \vdash e_r : t_r \qquad \rho[f \mapsto t_x \to t_r] \vdash e_b : t}{\rho \vdash \texttt{let}\ f\ (x : t_x) = e_r : t_r\ \texttt{in}\ e_b : t}$$

$$\frac{\rho(f) = t_x \to t_r \qquad \rho \vdash e : t_x}{\rho \vdash f\ e : t_r}$$

How to read a type rule

$$\frac{\rho \vdash e_1 : \text{int} \qquad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 < e_2 : \text{bool}}$$

Environment (rho)

Judgement: In environment ρ, expression e_2 has type int

Premises

Conclusion

- IF
  - in environment ρ, expression e_1 has type int, and
  - in environment ρ, expression e_2 has type int
- THEN
  - in environment ρ, expression $e_1 < e_2$ has type bool

Joint exercise: How read these?

$$\rho \vdash i : \mathtt{int}$$

An integer constant has type int

$$\frac{\rho(x) = t}{\rho \vdash x : t}$$

$$\frac{\rho \vdash e_1 : \mathtt{bool} \qquad \rho \vdash e_2 : t \qquad \rho \vdash e_3 : t}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : t}$$

$$\frac{\rho \vdash e_r : t_r \qquad \rho[x \mapsto t_r] \vdash e_b : t}{\rho \vdash \mathtt{let}\ x = e_r\ \mathtt{in}\ e_b\ \mathtt{end}\ : t}$$

Combining type rules to trees

- Stacking type rules on top of each other
- One rule's conclusion is another's premise
- Checking **`let x=1 in x<2 end : bool`** in some environment ρ:

$$\dfrac{\rho \vdash 1 : \mathtt{int} \qquad \dfrac{\rho[x \mapsto \mathtt{int}] \vdash x : \mathtt{int} \qquad \rho[x \mapsto \mathtt{int}] \vdash 2 : \mathtt{int}}{\rho[x \mapsto \mathtt{int}] \vdash x < 2 : \mathtt{bool}}}{\rho \vdash \mathrm{let}\ x = 1\ \mathrm{in}\ x < 2\ \mathrm{end} : \mathrm{bool}}$$

- The **`typ`** function implements the rules, from conclusion to premise!

Joint exercises: Invent type rules

- For e_1 `&&` e_2 (logical and)
- For e_1 `::` e_2 (list cons operator)
- For `match e with [] ->` e_1 `| x::xr ->` e_2

Dynamically or statically typed

- Dynamically typed:
  - Types are checked during evaluation (micro-ML, Postscript, JavaScript, Python, Ruby, Scheme, …)

```
if (true) {return 11} else {return 22+false}
```

OK, gives 11

- Statically typed:
  - Types are checked before evaluation (our typed fun. language, F#, most of Java and C#)

```
if true then 11 else 22+false
```

Compile-time type error

```
true ? 11 : (22 + false)
```

Compile-time type error

Dynamic typing in Java/C# arrays

- For a Java/C# array whose element type is a reference type, all assignments are type-checked at runtime

```
void M(Object[] a, Object x) {
  a[0] = x;
}
```

Type check needed at run-time

- Why is that necessary?

```
String[] s = new String[1];
M(s, new Object());
String s0 = s[0];
```