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:
  \[ \text{if true then } 1+2 \text{ else } 1+\text{false} \]
• 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):

\[
\begin{align*}
5+7 \\
\text{let } f x = x + 7 \text{ in } f 2 \text{ end} \\
\text{let } \text{fac } x = \text{if } x=0 \text{ then } 1 \text{ else } x \times \text{fac}(x - 1) \text{ in } \text{fac} 10 \text{ end}
\end{align*}
\]
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

Letfun ("f", "x", Prim ("+", Var "x", CstI 7), Call (Var "f", CstI 2))
Runtime values, function closures

- **Run-time values**: integers and functions

```ocaml
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:

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

- Should always have value 11

- Evaluate as `3 + y`
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

```ocaml
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

\[ \begin{align*}
\rho \vdash i \Rightarrow i & \quad (e1) \\
\rho \vdash b \Rightarrow b & \quad (e2) \\
\rho(x) = v & \quad (e3) \\
\rho \vdash x \Rightarrow v & \\
\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 & \quad v = v_1 + v_2 \quad (e4) \\
\rho \vdash e_1 + e_2 \Rightarrow v & \\
\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 & \quad b = (v_1 < v_2) \quad (e5) \\
\rho \vdash e_{r} \Rightarrow v_{r} \quad \rho[x \mapsto v_{r}] \vdash e_{b} \Rightarrow v & \quad (e6) \\
\rho \vdash \text{let } x = e_{r} \text{ in } e_{b} \text{ end } \Rightarrow v & \\
\rho \vdash e_1 \Rightarrow \text{true} \quad \rho \vdash e_2 \Rightarrow v & \quad (e7t) \\
\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v & \\
\rho \vdash e_1 \Rightarrow \text{false} \quad \rho \vdash e_3 \Rightarrow v & \quad (e7f) \\
\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v & 
\end{align*} \]
Evaluation by logical rules:
Function declaration and call

• Compare these with the `eval` interpreter:

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

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

• 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
```

Evaluate as $3 + y$
A dynamic scope variant of Micro-ML

- Very minimal change in interpreter:

```ocaml
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
```

- 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
    \[ 1 + (* 33 (* \text{was} 44 *) *) 22 \]

• Parser:
  – To parse applications \( e_1 e_2 e_3 \) 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

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

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

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

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

```ocaml
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
```

```ocaml
TyPF (TypI, TypI)
```
Type checking by recursive function

- Using a type environment ["x", TypI]:

```ocaml
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=eRh in letBody end`
- Checking `if e1 then e2 else e3`

```ocaml
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`

```ocaml
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 \texttt{typ} and the interpreter \texttt{eval} have similar structure.

• Type checking can be thought of as \textit{abstract interpretation} of the program.

• We calculate “\texttt{TypI + TypI gives TypI}” instead of “\texttt{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

\[
\rho \vdash i : \text{int} \\
\rho \vdash b : \text{bool} \\
\rho(x) = t \\
\rho \vdash x : t \\
\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int} \\
\rho \vdash e_1 + e_2 : \text{int} \\
\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int} \\
\rho \vdash e_1 < e_2 : \text{bool} \\
\rho \vdash e_r : t_r \quad \rho[x \leftarrow t_r] \vdash e_b : t \\
\rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} : t \\
\rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t \\
\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \\
\rho[x \leftarrow t_x, f \leftarrow t_x \rightarrow t_r] \vdash e_r : t_r \quad \rho[f \leftarrow t_x \rightarrow t_r] \vdash e_b : t \\
\rho \vdash \text{let } f(x : t_x) = e_r : t_r \text{ in } e_b : t \\
\rho \vdash e : t_x \\
\rho \vdash f e : t_r
\]
How to read a type rule

• IF
  – in environment $\rho$, expression $e_1$ has type int, and
  – in environment $\rho$, expression $e_2$ has type int

• THEN
  – in environment $\rho$, expression $e_1 < e_2$ has type bool
Joint exercise: How read these?

An integer constant has type int

\[ \rho \vdash i : \text{int} \]

\[ \rho(x) = t \quad \Rightarrow \quad \rho \vdash x : t \]

\[ \rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t \]

\[ \rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

\[ \rho \vdash e_r : t_r \quad \rho[x \mapsto t_r] \vdash e_b : t \]

\[ \rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} : t \]
Combining type rules to trees

- Stacking type rules on top of each other
- One rule’s conclusion is another’s premise
- Checking \( \text{let } x=1 \text{ in } x<2 \text{ end : bool} \) in some environment \( \rho \):

\[
\begin{align*}
\rho[x \mapsto \text{int}] \vdash x : \text{int} & \quad \rho[x \mapsto \text{int}] \vdash 2 : \text{int} \\
\rho \vdash 1 : \text{int} & \quad \rho[x \mapsto \text{int}] \vdash x < 2 : \text{bool} \\
\rho \vdash \text{let } x = 1 \text{ in } x < 2 \text{ end : bool}
\end{align*}
\]

- The \( \texttt{typ} \) 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, ...)

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

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

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

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

OK, gives 11

Compile-time type error

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

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

Why is that necessary?

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