

## Programming Languages, F2002

### Lecture 4, Wednesday 27 February 2002 \*

- A first-order functional language: function declarations and expressions
- Static and dynamic scope
- Run-time representation of values, recursive closures
- Interpretation of the functional language with static scope
- Interpretation with dynamic scope
- Types and type environments
- An explicitly typed first-order functional language
- Type checking rules
- A type checker
- Untyped, dynamically typed, or statically typed

IFC

Programming Languages, F2002

Page 4-1

### Modelling a first-order functional language

Essentially, micro-SML, a subset of Standard ML. Later we shall look at micro-C and micro-Java.

Restrictions:

- The language is first-order: functions cannot be passed as parameters or returned from functions. (Next week we consider a higher-order language: functions as parameters and as results).
- A function takes exactly one argument.
- There are only integer and boolean data.
- No pattern matching, exceptions, etc.

Example programs — every program is just an expression:

```
2
true
x1
let x2 = 2 in x2 + 17 end
x2 + 17
if 2<3 then 17 else 18
let fac n = if n=0 then 1 else n * fac(n-1) in fac 10 end
fac 10
```

IFC

Programming Languages, F2002

Page 4-2

### Abstract syntax for micro-SML

File fun/Absyn.sml:

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

This is just the expression language plus conditional expressions (if), function declarations, and function calls.

There are lexer and parser specifications in fun/FunLex.lex and fun/FunParser.grm.

There is a parser in fun/Parser.sml and instructions in fun/Grammar.txt.

IFC

Programming Languages, F2002

Page 4-3

### Evaluation: static scope or dynamic scope?

The function F has a free variable x:

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

When evaluating F 3 should y in F refer to 11 or to 22?

In a language with *static scope*, Y would refer to 11; in a language with *dynamic scope*, it would refer to 22.

The programming language Lisp (1960) has dynamic scope rules.

Algol, Pascal, Ada, C++, Java, Scheme, Standard ML, ... have static scope rules.

All serious programming languages developed since 1965 have static scope rules.

Static scope is more efficient, more 'logical', and permits compile-time type checking.

With static scope, a function must somehow remember the values of its free variables.

IFC

Programming Languages, F2002

Page 4-4

### Evaluation: Run-time values, run-time environments, and closures

A name can be bound to an integer, or a boolean, or a function.

At runtime, both integers and booleans are represented by integers: false by 0 and true by 1.

### Run-time environments: mapping names to values

As for the expression language, evaluation takes place in a run-time environment.

Notation:  $[x \mapsto 3, y \mapsto 11]$  is the environment that maps  $x$  to 3 and  $y$  to 11.

### Representing functions by closures

The representation of a function  $F$  can record the values of its free variables in an environment.

Thus a function  $F$   $x = ebody$  is represented by a closure  $(F, x, ebody, fenv)$ .

Example closure:  $(\text{"F"}, \text{"x"}, \text{Prim}(\text{"+"}), [\text{Var "x"}, \text{Var "y"}], [y \mapsto 11])$ .

(Actually, a Java object containing a non-static method  $m(\dots)$  is a kind of closure for  $m$ .)

The method's free variables must be fields in the object; the object provides values for those fields.)

### Run-time values and run-time environments

A run-time value is either an integer (Int) or a function closure (RCLO).

A closure contains the function name, parameter name, and body, and a variable-and-function environment.

Let  $(\text{'key}, \text{'data})$  env be the type of environments that map keys of type  $\text{'key}$  to data of type  $\text{'data}$ .

A variable-and-function environment maps names to values; it's a  $(\text{string}, \text{value})$  env.

As a result, the type definitions for values and environments depend on each other:

```
datatype value =
  Int of int
  | RCLO of string * string * expr * vFenv (* (F, x, body, bodyenv) *)
  withtype vFenv = (string, value) env
```

### An SML structure Env with operations on environments

See files Env.sig and Env.sml:

```
type ('key, 'data) env
val empty : ('key, 'data) env
val lookup : ('key, 'data) env -> 'key -> 'data
val bind1 : ('key, 'data) env -> 'key * 'data -> ('key, 'data) env
val plus : ('key, 'data) env * ('key, 'data) env -> ('key, 'data) env
...
```

Must compile with `mosmlc -c Env.sig Env.sml` before use in `Fun/Fun.sml` etc.

### The evaluation function, part 1

```
fun eval (e : expr) (env : vFenv) : int =
  case e of
  | CstI i => i
  | CstB b => bool2int b
  | Var x => (case lookup env x of
    | _ => raise Fail "eval Var")
    | Int i => i
  | Prim(ope, [e1, e2]) =>
    let val i1 = eval e1 env
        val i2 = eval e2 env
    in
      case ope of
        "*" => i1 * i2
        "+" => i1 + i2
        "-" => i1 - i2
        "=" => bool2int (i1 = i2)
        "<" => bool2int (i1 < i2)
        _ => raise Fail "unknown primitive"
      end
    end
  | Let(x, eh1, ebody) =>
    let val xv1 = eval eh1 env
        val env1 = bind1 env (x, xv1)
    in eval ebody env1 end
  | ...
```

### The evaluation function, part 2

```
fun eval (e : expr) (env : vFenv) : int =
  ...
  | If(e1, e2, e3) =>
    let val b = eval e1 env
    in
      if int2bool b then eval e2 env
      else eval e3 env
    end
  | Letfun(F, x, fbody, ebody) =>
    let val env1 = bind1 env (F, fbody, env)
    in eval ebody env1 end
  | Call(Var f, args) =>
    (case lookup env f of
     fclosure as RCLO (f, x, fbody, envf) =>
      let val argv = Int(eval earg env)
          val env2 = bind1 envf (f, fclosure)
          val env3 = bind1 env2 (x, argv)
        in eval fbody env3 end
     | _ => raise Fail "eval Call: not a function")
  | Call _ => raise Fail "eval Call: illegal function expression"
```

### Evaluation with dynamic scope rules

With dynamic scope rules, the environment in a closure for function  $f$   $x = ebody$  is not needed.

Evaluating a call  $f$   $e$  just requires binding the parameter  $x$  to the value of  $e$  in the call-site environment:

```
fun eval (e : expr) (env : vEnv) : int =
  ...
  | Letfun(f, x, fbody, ebody) =>
    let val env1 = bind1 env (f, RClO(f, x, fbody, empty))
      in eval ebody env1 end
  | Call(Var f, earg) =>
    (case Lookup env f of
     fclousure as RClO (f, x, fbody, _) =>
       let val argv = Int(eval earg env)
         val env3 = bind1 env (x, argv)
         in eval fbody env3 end
     | _ => raise Fail "eval Call: illegal function expression")
```

Apparently dynamic scope is simple. But it is a bad idea.

It works only if a function's free variables are bound where the function is called.

This precludes e.g. partial application of curried functions, such as

```
fun plus x y = x + y;
val incr = plus 1;
val res = incr 7;
```

FC

Programming Languages, F2002

Page 4-9

### Informal type checking rules for the functional language

- An integer constant  $(0, 1, -1, \dots)$  has type `int`.
  - A boolean constant (`true`, `false`) has type `bool`.
  - A variable occurrence  $x$  has the type of its binding.
  - An addition expression  $e_1 + e_2$  has type `int` provided  $e_1$  has type `int` and  $e_2$  has type `int`.
  - A comparison expression  $e_1 < e_2$  has type `bool` provided  $e_1$  has type `int` and  $e_2$  has type `int`.
  - A let-binding `let x =  $e_r$  in  $e_b$  end` has the same type  $t$  as the body  $e_b$ .
- First find the type  $t_r$  of  $e_r$ , and then find the type  $t$  of  $e_b$  under the assumption that  $x$  has type  $t_r$ .
- A conditional expression `if  $e_1$  then  $e_2$  else  $e_3$  end` has type  $t$  provided  $e_1$  has type `bool` and  $e_2$  has type  $t$  and  $e_3$  has type  $t$ .
  - A function declaration `let  $f(x : t_x) = e_r$  in  $e_b$  end` has the same type  $t$  as  $e_b$ .
- First check that  $e_r$  has type  $t_r$  under the assumption that  $x$  has type  $t_x$  and  $f$  has type  $t_x \rightarrow t_r$ . Then find the type  $t$  of  $e_b$  under the assumption that  $f$  has type  $t_x \rightarrow t_r$ .
- A function application  `$f$   $e$`  has type  $t_x$  provided  $f$  has type  $t_x \rightarrow t_r$  and  $e$  has type  $t_x$ .

FC

Programming Languages, F2002

Page 4-11

### The language so far is untyped

The interpreter `eval` will happily evaluate expressions such as these:

```
if 2 then 11 else 22           true + 5
```

But evaluation of this expression fails because one cannot add 3 to a function:

```
let f x = x + 1 in f + 3 end
```

### Types for our functional language

A type is `int` or `bool` or a function type  $t_1 \rightarrow t_2$ :

```
 $t ::= \text{int} \mid \text{bool} \mid t_1 \rightarrow t_2$ 
```

Examples: The type of 2 is `int`. The type of `fac` is `int  $\rightarrow$  int`.

### Explicitly typed function declarations

We require function declarations to have types on arguments and results:

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

```
let ge2 (n : int) = (1<n) : bool in if ge2 z then 11 else 22 end
```

Java, ANSI C, C++, C#, Ada, Pascal require explicit types on function/method declarations.

FC

Programming Languages, F2002

Page 4-10

### Formal type checking rules

The type environment  $\rho = [x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$  maps variable names  $x$  to types  $t$ .

The judgement  $\rho \vdash e : t$  asserts that in type environment  $\rho$ , the expression  $e$  has type  $t$ .

Below,  $i$  is an integer constant,  $b$  a boolean constant,  $x$  a variable, and  $e_1, e_1, \dots$  are expressions.

$$\rho \vdash i : \text{int}$$
$$\rho \vdash b : \text{bool}$$
$$\frac{\rho(x) = t}{\rho \vdash x : t}$$
$$\rho \vdash x : t$$
$$\frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 + e_2 : \text{int}}$$
$$\rho \vdash e_1 + e_2 : \text{int}$$
$$\frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 < e_2 : \text{bool}}$$
$$\rho \vdash e_1 < e_2 : \text{bool}$$
$$\frac{\rho \vdash e_r : t_r \quad \rho[x \mapsto t_r] \vdash e_b : t}{\rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} : t}$$
$$\rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} : t$$

FC

Programming Languages, F2002

Page 4-12

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

$$\frac{\rho[x \mapsto t_x; f \mapsto t_x \rightarrow t_r] \vdash e_r : t_r \quad \rho[f \mapsto t_x \rightarrow t_r] \vdash e_b : t}{\rho \vdash \text{let } f(x : t_x) = e_r : t_r \text{ in } e_b : t}$$

$$\frac{\rho(f) = t_x \rightarrow t_r \quad \rho \vdash e : t_r}{\rho \vdash f e : t_r}$$

**Type checking examples**

Type checking the expression `let x = 1 in x < 2 end`:

$$\frac{\rho[x \mapsto \text{int}] \vdash x : \text{int} \quad \rho[x \mapsto \text{int}] \vdash 2 : \text{int}}{\rho \vdash 1 : \text{int} \quad \rho[x \mapsto \text{int}] \vdash x < 2 : \text{bool}}$$

$$\rho \vdash \text{let } x = 1 \text{ in } x < 2 \text{ end} : \text{bool}$$

Type checking the expression `let b = (1 < 2) in if b then 3 else 4 end`:

$$\frac{\rho \vdash 1 : \text{int} \quad \rho \vdash 2 : \text{int} \quad \rho[z \mapsto \text{bool}] \vdash z : \text{bool} \quad \rho' \vdash 3 : \text{int} \quad \rho' \vdash 4 : \text{int}}{\rho \vdash 1 < 2 : \text{bool} \quad \rho[z \mapsto \text{bool}] \vdash \text{if } z \text{ then } 3 \text{ else } 4 \text{ end} : \text{int}}$$

$$\rho \vdash \text{let } z = (1 < 2) \text{ in if } z \text{ then } 3 \text{ else } 4 \text{ end} : \text{int}$$

Above  $\rho' = \rho[z \mapsto \text{bool}]$  for brevity.

What is the type checking tree for `let f (x : int) = x < 2 : bool in f 6 end`? Is there a type checking tree for `1 < true`?

**Modelling types in SML**

Types and type schemas are modelled in SML using these datatype:

```
datatype typ =
  | TypeI
  | TypeB
  | TypeF of typ * typ
  (* int
  (* bool
  (* (argumenttype, resulttype) *)
```

Example types:

TypeI represents type `int`

TypeF (TypeI, TypeB) represents type `int → bool`

A *type environment* maps a variable or function name to its type.

A type environment may have SML type `(string, typ) env`.

**Abstract syntax for an explicitly typed functional language**

The abstract syntax for function declarations involves types.

```
datatype tyexpr =
  | CstI of int
  | CstB of bool
  | Var of string
  | Let of string * tyexpr * tyexpr
  | Prim of string * tyexpr list
  | If of tyexpr * tyexpr * tyexpr
  | Letfun of string * string * typ * tyexpr * typ * tyexpr
  (* (f, x, body, rty, ebody *)
  | Call of tyexpr * tyexpr
```

A typed function declaration `Letfun (f, x, xty, fbody, rty, ebody)` corresponds to this:

```
let f (x : xty) = fbody : bty in ebody end
```

Evaluation is almost as before, see file `fun/tychk.sml`.

Type checking is done in a type environment `env`, with SML type `tyenv = (string, typ) env`.

### An implementation of type checking

```
fun typ (env : tyenv) (e : tyexpr) : typ =
  case e of
  CstI i => TypI
  | CstB b => TypB
  | Var x => lookup env x
  | Prim(ope, [el, e2]) =>
    let fun chk ta tb tr =
        if typ env el=ta andalso typ env e2=tb then tr
        else raise Type "Prim"
    in
      case ope of
      "*" => chk TypI TypI TypI
      | "+" => chk TypI TypI TypI
      | "-" => chk TypI TypI TypI
      | "=" => chk TypI TypI TypB
      | "<" => chk TypI TypI TypB
      | "&" => chk TypB TypB TypB
      | _ => raise Fail "unknown primitive"
    end
  | Let(x, ebody) =>
    let val xty = typ env ebody
      val env1 = bind1 env (x, xty)
    in
      typ env1 ebody
    end
  | ...
```

FC Programming Languages, F2002

Page 4-17

### Type checking and evaluation are rather similar

The structure of the type checker (function `typ`) is similar to that of the interpreter (function `eval`).

Type checking can be thought of as an *abstract interpretation* of the program.

The type checker computes with *types* instead of values.

For instance, `TypI + TypI` is `TypI` instead of `Int 3 + Int 5` is `Int 8`.

The type checker 'interprets' function calls without interpreting the function body, so it always terminates.

For instance, evaluation (by `eval`) of this expression will evaluate the function body 100000 times:

```
let deep x = if x=0 then 1 else deep (x-1) in deep 100000 end
```

But type checking (by `typ`) will check the function body only once.

This is possible because of the explicit types on function declarations.

FC

Programming Languages, F2002

Page 4-19

### Untyped, dynamically typed, or statically typed?

In an **untyped** language, almost all kinds of operands can be mixed with some result (possibly a program crash).

Operations on so-called unions in C are untyped.

In a **dynamically typed** language, an expression may be rejected as ill-typed when it is evaluated.

Scheme is dynamically typed, and will evaluate `(if #f (+ 56 #c) 45)`.

Postscript is dynamically typed, and will evaluate `false { 56 true add } { 45 } ifelse =`.

Java and C# are dynamically typed as concerns collection classes, because all values are cast to `Object`.

Assignments to Java and C# arrays of reference types are dynamically typed also!

In a **statically typed** language, an expression may be rejected early, regardless whether it will ever be evaluated.

Java/C#/C++ are statically typed as concerns base types, and the compiler will reject this expression

```
false ? (56 + true) : 45
```

although the subexpression `56 + true` will never be evaluated.

Standard ML, Haskell, Cyclone are statically typed.

This means that certain kinds of errors cannot appear at run-time.

FC

Programming Languages, F2002

Page 4-18

FC

Programming Languages, F2002

Page 4-20