

Programming Languages, F2003

Lecture 4, Wednesday 26 February 2003

- 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

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/Funpar.grm`.

There is a parser in `fun/parse.sml` and instructions for use in `fun/grammar.txt`.

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. (Later we study 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.

Some 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
```

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
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.

Most programming languages developed since 1965 have static scope rules (and Perl has both, of course).

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

Note: With static scope, a function such as `f` must somehow remember the values of its free variables (`y`).

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 $f \ x = \text{ebody}$ is represented by a *closure* $(f, x, \text{ebody}, \text{fenv})$.

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

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.

The evaluation function, part 1: the expression language

```
fun eval (e : expr) (env : vfv) : int =
  case e of
  CstI i => i
  | CstB b => if b then 1 else 0
  | Var x => (case lookup env x of
              Int i => i
              | _ => raise Fail "eval Var")
  | 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
        "=" => if i1 = i2 then 1 else 0
        "<" => if i1 < i2 then 1 else 0
        _ => raise Fail "unknown primitive"
      end
    | Prim _ => raise Fail "eval Prim: unknown arity"
  | Let(x, erhs, ebody) =>
    let val xval = eval erhs env
        val env1 = bind1 env (x, Int xval)
    in eval ebody env1 end
  | ...
```

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

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

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

```
datatype value =
  Int of int
  | RClo of string * string * expr * vfv          (* (f, x, body, bodyenv) *)
withtype vfv = (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 2: conditionals, functions

```
fun eval (e : expr) (env : vfv) : int =
  case e of
  ...
  | If(e1, e2, e3) =>
    let val b = eval e1 env
    in
      if b <> 0 then eval e2 env
      else eval e3 env
    end
  | Letfun(f, x, fbody, ebody) =>
    let val env1 = bind1 env (f, RClo(f, x, fbody, env))
    in eval ebody env1 end
  | Call(Var f, earg) =>
    (case lookup env f of
      fclosure as RClo (f, x, fbody, fenv) =>
        let val argv = Int(eval earg env)
            val env2 = bind1 fenv (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 $f \ x = \text{ebody}$ 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
     fclosure 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: not a function")
  | Call _ => 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 addc x y = x + y
val addSeventeen = addc 17;
val res3 = addSeventeen 25;
```

Informal type checking rules for the functional language

- An integer constant $(0, 1, -1, \dots)$ has type `int`.
- A boolean constant $(\text{true}, \text{false})$ 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 $\text{let } x = e_r \text{ in } e_b \text{ end}$ 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 $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ has type t provided e_1 has type `bool` and e_2 has type t and e_3 has type t .
- A function declaration $\text{let } f(x : t_x) = e_r : t_r \text{ in } e_b \text{ end}$ 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_r provided f has type $t_x \rightarrow t_r$ and e has type t_x .

The language so far is untyped

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

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

But evaluation of this expression fails (why?) 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.

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 ρ , the expression e has type t .

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

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

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

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

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

Modelling types in SML

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

```
datatype typ =
  TypI                (* int          *)
  | TypB              (* bool        *)
  | TypF of typ * typ (* typ -> type *)
```

Example types:

TypI represents type `int`

TypF(TypI, TypB) 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.env`.

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 z = (1 < 2) in if z 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.

Is there is a type checking tree for `1 + x`?

Is there is a type checking tree for `1 < true`?

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

Abstract syntax for an explicitly typed functional language

The abstract syntax for function declarations is decorated with types (file `fun/tychk.sml`):

```
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, xty, fbody, rty, ebody *)
  | Call of tyexpr * tyexpr
```

The abstract syntax `Letfun (f, x, xty, fbody, rty, ebody)` represents this concrete syntax:

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

Evaluation is almost as before, see function `eval` in 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 (file fun/tychk.sml)

```
fun typ (env : tyenv) (e : tyexpr) : typ =
  case e of
  CstI i => TypI
  | CstB b => TypB
  | Var x => lookup env x
  | Prim(ope, [e1, e2]) =>
    let fun chk ta tb tr =
        if typ env e1=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, erhs, ebody) =>
      let val xty = typ env erhs
          val env1 = bind1 env (x, xty)
        in
          typ env1 ebody
        end
    | ...
```

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 (versus concrete interpretation).

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

For instance, 'TypI + TypI gives TypI' instead of 'Int 3 + Int 5 gives 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.

```
fun typ (env : tyenv) (e : tyexpr) : typ =
  ...
  | If(e1, e2, e3) =>
    (case typ env e1 of
     TypB => let val e2ty = typ env e2
                val e3ty = typ env e3
              in
                if e2ty = e3ty then e2ty
                else raise Type "If: e2 and e3 different types"
              end
     | _ => raise Type "If: e1 not boolean")
  | Letfun(f, x, xty, fbody, rty, ebody) =>
    let val env1 = bind1 env (f, TypF(xty, rty))
        val env2 = bind1 env1 (x, xty)
    in
      if typ env2 fbody = rty then typ env1 ebody
      else raise Type ("Letfun: wrong type given " ^ f)
    end
  | Call(e, earg) =>
    (case typ env e of
     TypF(xty, fty) =>
       if typ env earg = xty then fty
       else raise Type "Call: wrong argument type"
     | _ => raise Type "Call: attempt to apply non-function")
```

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: `(if #f (+ 56 #t) 45)` evaluates to 45.

Postscript is dynamically typed: `false { 56 true add } { 45 } ifelse =` evaluates to 45.

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

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

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.

Array element assignment in Java and C# is dynamically typed

Class Integer and class Double are subclasses of class Number.

When `arr` is an Integer array, we expect `arr[0]` to be an Integer object.

It seems OK to assign `arr` to a variable `arrn` of type `Number[]`.

But the assignment `arrn = arr` makes `arrn` an alias for `arr`; they point to the same array.

So it is unsafe to store a Double into `arrn[0]` and hence `arr[0]`.

Therefore Java performs a runtime check at every assignment to an array element of reference type:

```
Integer[] arr = new Integer[16];
Number[] arrn = arr;           // Compiles OK
arrn[0] = new Double(3.14);    // Compiles OK, but fails at runtime
... arr[0] is expected to be an Integer ...
```

So Java array element assignment is dynamically typed.

The same is true of C#.

Collection classes in Java and C# are dynamically typed

A `LinkedList` must be able to hold any kind of objects.

Therefore the argument type of `add` is `Object`.

And the return type of `get` is `Object`.

Hence, no complaint when we store Integer into `names`.

Hence, need a type cast when extracting a person from `names`.

```
LinkedList names = new LinkedList();
names.add(new Person("Kristen"));
names.add(new Person("Bjarne"));
names.add(new Integer(1998));    // Wrong, but no compiletime check
names.add(new Person("Anders"));
...
Person p = (Person)names.get(2); // Compiles OK, but fails at runtime
```

So Java collection classes are dynamically typed.

The same is true of C#.