15-150

Principles of Functional Programming

Michael Erdmann   Frank Pfenning

Miranda Lin   Helen Li   Harrison Grodin

Aditi Gupta, Alexander Liao, Ariel Davis, Ashwin Srinivasan,
Brandon Wu, Brian Scheuermann, Disha Das, Elliot Spargo,
Emma Cohron, Ethan Rosenthal, Eunice Chen,
Gabriel Chuang, George Ralph, Henry Nelson,
Isabel Gan, Isabelle Augensen, Jacob Neumann, Julia Gu,
Kalvin Chang, Kaz Zhou, Keshav Narayan, Kevin Grosman,
Matthew McQuaid, Mia Tang, Michael Zhang, Minji Kim,
Minji Lee, Nathan Walker, Nikhita Subbiah,
Samarth Malhotra, Shyam Sai, Siddharth Girdhar, Sue Lee,
Timothy Ganger

Course Webpage

http://www.cs.cmu.edu/~15150/

Policies: http://www.cs.cmu.edu/~15150/policy.html

Lectures: http://www.cs.cmu.edu/~15150/lect.html

Course Philosophy

Computation is Functional.

Programming is an explanatory linguistic process.

Computation is Functional

values : types

expressions

Functions map values to values

Imperative vs. Functional

Command

↓

- executed
- has an effect

$x := 5$

(state)

Expression

↓

- evaluated
- no effect

$3 + 4$

(value)

Programming as Explanation

Problem statement

↓

high expectation to explain precisely & concisely

- invariants
- specifications
- proofs of correctness
- code

Analyze, Decompose & Fit, Prove

Parallelism

/\
< 1, 0, 0, 1, 1 > ➔ 3,
< 1, 0, 1, 1, 0 > ➔ 3,
< 1, 1, 1, 0, 1 > ➔ 4,
< 0, 1, 1, 0, 0 > ➔ 2,
\/
↓
12

Parallelism

```
sum : int sequence  →  int
type row  =  int sequence
type room = row sequence

fun count  (class : room) : int =
        sum (map sum class)
```

Parallelism

- Work:
  - Sequential Computation
  - Total sequential time;

    number of operations

- Span:
  - Parallel Computation
  - How long would it take if one could have as many processors as one wants;

    length of longest critical path

Defining ML (Effect-Free Fragment)

- Types t
- Expressions e
- Values v (subset of expressions)

Examples:

$$(3+4) * 2$$

$$\xRightarrow{1} \quad 7 * 2$$

$$\xRightarrow{1} \quad 14$$

$$(3+4) * (2+1)$$

$$\xRightarrow{3} \quad 21$$

`"the " ^ "walrus"`

$\xRightarrow{1}$ `"the walrus"`

The expression

`"the " ^ "walrus"`

reduces to the value

`"the walrus"` .

It has type `string`.

"the walrus" + 1

$\Longrightarrow$??

The expression

"the walrus" + 1

does not have a type and it does not reduce to a value.

Types

A ***type*** is a **prediction** about the kind of value an expression will have if it winds up reducing to a value.

An expression is ***well-typed*** if it has at least one type, and ***ill-typed*** otherwise.

(We may also say that an expression ***type-checks***, meaning that it is well-typed.)

First, type-check an expression.

If the expression is well-typed,
then evaluate the expression.

(The ML compiler does that.)

Expressions

Every well-formed ML expression `e`

- has a type `t`, written as $\underline{\texttt{e : t}}$
- may have a value `v`, written as $\underline{\texttt{e} \hookrightarrow \texttt{v}}$.
- may have an effect (not for our effect-free fragment)

Example: $(3+4)*2 \; : \; \text{int}$

$(3+4)*2 \hookrightarrow 14$

Type `int`

Values …, `~1, 0, 1`, …,

that is, every integer n.

Expressions e_1 `+` e_2, e_1 `-` e_2, e_1 `*` e_2,

e_1 `div` e_2, e_1 `mod` e_2, *etc.*

Example: ~4 * 3

Typing Rules

- n : `int`
- e_1 `+` e_2 : `int`

  if e_1 : `int` and e_2 : `int`

similar for other operations.

Example:

(3 + 4) * 2 : int

Why?

3+4 : int and 2 : int

Why?

3 : int and 4 : int

Evaluation Rules

- $e_1 + e_2 \xrightarrow{1} e_1' + e_2$ if $e_1 \xrightarrow{1} e_1'$
- $n_1 + e_2 \xrightarrow{1} n_1 + e_2'$ if $e_2 \xrightarrow{1} e_2'$
- $n_1 + n_2 \xrightarrow{1} n$,

  with n the sum of the integer values n_1 and n_2.

Example of a well-typed expression with no value

5 div 0 : int

5 div 0 : int

because 5 : int
& 0 : int

and because div expects
two ints and returns an int.

However, 5 div 0
does not reduce to a value.

Notation Recap

$e : t$ "e has type t"

$e \Rightarrow e'$ "e reduces to e'"

$e \hookrightarrow v$ "e evaluates to v"

Extensional Equivalence

$$\cong$$

An equivalence relation on expressions
(of the same type).

Extensional Equivalence

- Expressions are *extensionally equivalent* *if they have the same type and one of the following is true:*

  both expressions reduce to the same value,
  or both expressions raise the same exception,
  or both expressions loop forever.

- Functions are *extensionally equivalent* if they map equivalent arguments to equivalent results.

- In proofs, we use $\cong$ as shorthand for "is equivalent to".

- Examples: $21 + 21 \cong 42 \cong 6 * 7$
  $[2, 7, 6] \cong [1+1, 2+5, 3+3]$
  $(\text{fn } x \Rightarrow x + x) \cong (\text{fn } y \Rightarrow 2 * y)$

- Functional programs are *referentially transparent*, meaning:
  - The *value* of an expression depends only on the *values* of its sub-expressions.
  - The *type* of an expression depends only on the *types* of its sub-expressions.

Types in ML

Basic types:

int, real, bool, char, string

Constructed types:

product types
function types
user-defined types

Products, Expressions

Types $t_1 * t_2$ for any type t_1 and t_2.

Values $(v_1,\ v_2)$ for values v_1 and v_2.

Expressions $(e_1,\ e_2),\quad \underbrace{\#1\ e,\quad \#2\ e}_{\text{DO NOT USE!}^{*}}$

Examples: (3+4, true)

(1.0, ~15.6)

(8, 5, false, ~2)

* You will learn how to extract components using pattern matching

Typing Rules

- $(e_1,\ e_2)\ :\ t_1\ *\ t_2$

  if $e_1 : t_1$

  and $e_2 : t_2$

Example: $(3+4,\ \text{true}) : \text{int} * \text{bool}$

Evaluation Rules

- $(e_1,\ e_2) \xrightarrow{1} (e_1',\ e_2) \quad \text{if } e_1 \xrightarrow{1} e_1'$
- $(v_1,\ e_2) \xrightarrow{1} (v_1,\ e_2') \quad \text{if } e_2 \xrightarrow{1} e_2'$

Functions

In math, one talks about a function f mapping between spaces X and Y,

$$f : X \to Y$$

In SML, we will do the same, with X and Y being types.

Issue: Computationally, a function may not always return a value. That complicates checking equivalence.

Definition: A function f is ***total*** if f(x) returns a value for all values x in X.

(Totality is a key difference between math and computation.)

Sample Function Code

```
(* square : int -> int
   REQUIRES: true
   ENSURES:  square(x) evaluates to x * x
*)

fun square (x:int) : int = x * x



(* Testcases: *)

val 0 = square 0
val 49 = square 7
val 81 = square (~9)
```

Sample Function Code

```
(* square : int -> int
   REQUIRES: true
   ENSURES:  square(x) evaluates to x * x
*)
```

function type

```
fun square (x:int) : int = x * x
```

keyword

function name

argument name & type

result type

body of function

```
(* Testcases: *)

val 0 = square 0
val 49 = square 7
val 81 = square (~9)
```

Five-Step Methodology

① `(* square : int -> int` function type

② `REQUIRES: true`

③ `ENSURES:  square(x) evaluates to x * x`

`*)`

④

`fun`	`square`	`(x:int)`	`: int`	`= x * x`
keyword	function name	argument name & type	result type	body of function

⑤ `(* Testcases: *)`

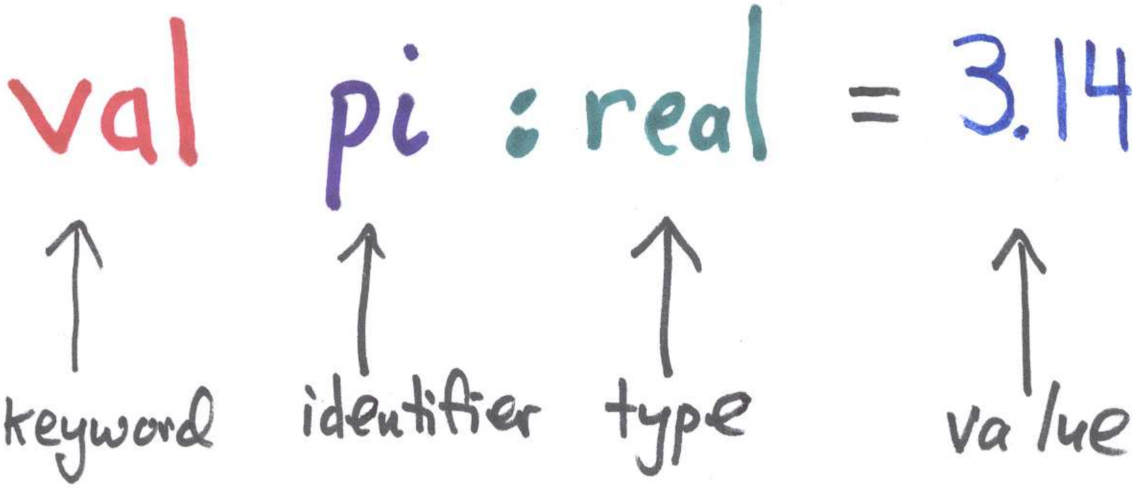
```
val 0 = square 0
val 49 = square 7
val 81 = square (~9)
```

Declarations

Environments

Scope

Declaration



```
val pi : real = 3.14
```

Introduces binding of 3.14 to pi
(sometimes written [3.14/pi])

Lexically statically scoped.

```
val x : int = 8-5        [3/x]
val y : int = x+1        [4/y]
val x : int = 10         [10/x]
val z : int = x+1        [11/z]
```

second binding of x *shadows* first binding.

First binding has been *shadowed*.

Local Declarations

`let ... in ... end`

```
let
    val m : int = 3
    val n : int = m*m
in
    m+n
end
```

This is an expression.

What type does it have? int

What value? 12

Local Declarations

```
val k : int = 4
```

```
let
    val k : real = 3.0
in
    k * k
end
```

↪ 9.0 : real

Type?

Value?

```
k
```

← Type?

Value?

↪ 4 : int

Concrete Type Def

```
type float = real

type point = float*float

val p : point = (1.0,2.6)
```

Closures

Function declarations also create value bindings:

```
fun square (x:int) : int = x * x
```

binds a closure to the identifier `square`.

Closures

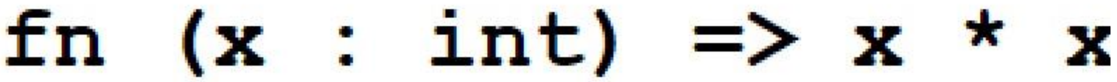
Function declarations also create value bindings:

```
fun square (x:int) : int = x * x
```

binds a closure to the identifier **`square`**.

The closure consists of two parts:

- A lambda expression (anonymous function value):



```
fn (x : int) => x * x
```

keyword

argument
name & type

body of function

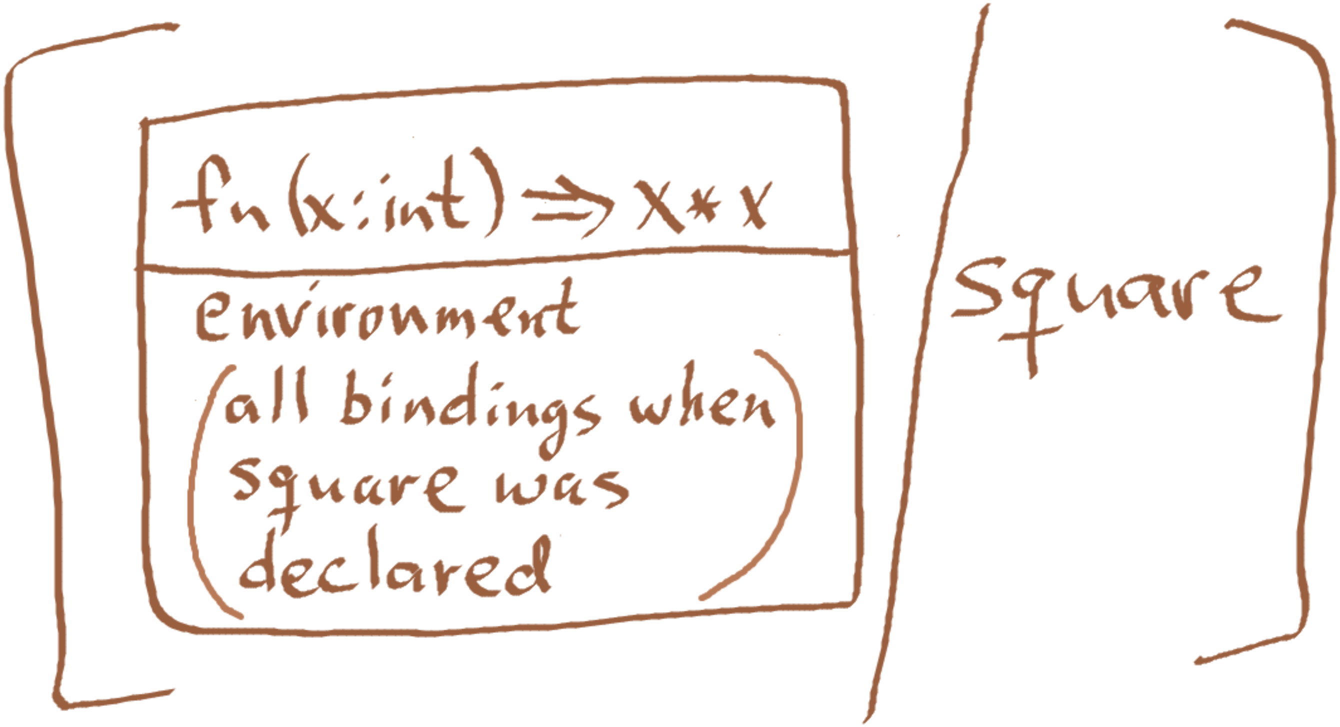
- An environment (all prior value bindings).

Closures

Function declarations also create value bindings:

```
fun square (x:int) : int = x * x
```

binds a closure to the identifier **`square`**.



Course Tasks

Assignments	35%
Labs	10%
Midterm 1	15%
Midterm 2	15%
Final	25%

Roughly one assignment per week, one lab per week.

Collaboration

Be sure to read the
course and university webpages
regarding academic integrity.