15-150

Principles of Functional Programming

Slides for Lecture 1

Introduction, Philosophy, Some Basics

January 16, 2024

15-150

Principles of Functional Programming

Michael Erdmann   Dilsun Kaynar

Aileen Guo   Brandon Dong   George Zhou

Alice Tran, Alvaro Luque, Andrew Lam, Ashley Xing,
Daniel Brown, Jacky Gao, Kiera O'Flynn, Leon Lu,
Nancy Kuang, Sanjana Kuchibhotla, Stephen Nah,
Caroline Shi, Vianna Seifi, Emma Tong, Ian Martin,
Benjamin Chen, Aria Lakhmani, Anna He, Ting Chen,
Nellie Tonev, Susan Huang, Yosef Alsuhaibani,
Crystal Zhang, Michael Duncan, Chris Roberts,
Michael Perry, Deon Kouatchou, Janise Kim,
Annie Zhang, Alan Abraham, Madison Zhao,
Nia Robinson, Max Kulbida, Justin Sun,
Gursimran Panesar, Stephen Mao

How to Succeed in College

SHOW UP

How to Succeed in College

SHOW UP

Go to lecture, go to lab.

Do not expect to understand everything in real-time.

Repeated exposure is important.

How to Succeed in College

Take Notes

By writing.

The eye-hand-brain loop is magical.

How to Succeed in College

Study Your Notes

How to Succeed in College

Study Your Notes

The same day.   And again.

And again.   And again.

Figure out what you don't understand.

Repeated exposure is important.

How to Succeed in College

Keep up

Do not let work pile up.

Small steps, big achievements.

How to Succeed in College

SHOW UP

Take Notes

Study Your Notes

Keep up

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

$$\bigwedge$$

$< 1, 0, 0, 1, 1 > \quad \rightarrow \quad 3,$

$< 1, 0, 1, 1, 0 > \quad \rightarrow \quad 3,$

$< 1, 1, 1, 0, 1 > \quad \rightarrow \quad 4,$

$< 0, 1, 1, 0, 0 > \quad \rightarrow \quad 2,$

$$\bigvee$$

$$\downarrow$$

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

Three Recent Theses

- *August 2022, Efficient and Scalable Parallel Functional Programming Through Disentanglement*, by Sam Westrick, advised by Umut Acar.

- June 2022, Deductive Verification for Ordinary Differential Equations: Safety, Liveness, and Stability, by Yong Kiam Tan, advised by André Platzer.

- October 2021, First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory, by Jonathan Sterling, advised by Robert Harper.

Defining ML (Effect-Free Fragment)

- Types t

- Expressions e

- Values v (subset of expressions)

Examples:

$$(3 + 4) * 2$$

$\overset{1}{\Rightarrow}$ $7 \quad * \quad 2$

$\overset{1}{\Rightarrow}$ 14

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

$\overset{3}{\Rightarrow}$ 21

"the " ^ "walrus"

⟹ "the walrus"

The expression
"the " ^ "walrus"
reduces to the value
"the walrus".

It has type string.

"the walrus" + 1

$\Rightarrow$ **??**

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 must have if it winds up reducing to a value.

(SML makes this prediction before evaluating the expression.
Evaluation may ultimately produce a value of that type,
but could alternatively raise an exception or loop forever.)

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

(The phrase '*to type-check e*' means to decide whether *e* is well-typed.
The phrase '*e type-checks*' means *e* is well-typed.)

Important: SML **never** evaluates an ill-typed expression.

___

Given an expression **e**:

First,
   SML determines whether **e** is well-typed.

If expression **e** is well-typed,
   then SML evaluates expression **e**;
   otherwise, SML reports a type error.

Expressions

Every well-formed ML expression **e**

- has a type **t**, written as $e : t$

- may have a value **v**, written as $e \hookrightarrow v$.

- may have an effect (not for our effect-free fragment)

Example: $(3+4) * 2 : int$

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

Type int

Values $\ldots, \tilde{\ }1, 0, 1, \ldots,$

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

- $n : \texttt{int}$

- $e_1 + e_2 : \texttt{int}$

  if $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$

  similar for other operations.

Example:

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

Why?

$$3+4 : \textit{int} \quad \text{and} \quad 2 : \textit{int}$$

Why?

$$3 : \textit{int} \quad \text{and} \quad 4 : \textit{int}$$

Evaluation Rules

- $e_1 + e_2 \overset{1}{\Longrightarrow} e_1' + e_2$ if $e_1 \overset{1}{\Longrightarrow} e_1'$

- $n_1 + e_2 \overset{1}{\Longrightarrow} n_1 + e_2'$ if $e_2 \overset{1}{\Longrightarrow} e_2'$

- $n_1 + n_2 \overset{1}{\Longrightarrow} n,$

  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

$$5 \text{ div } 0 : \text{int}$$

because $5 : \text{int}$

& $0 : \text{int}$

and because **div** expects
two **int**s and returns an **int**.

However, $5 \text{ div } 0$
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: $21 + 21 \cong 42 \cong 6 * 7$
  $[2, 7, 6] \cong [1+1, 2+5, 3+3]$
  $(\text{fn } x => x + x) \cong (\text{fn } y => 2 * y)$

- 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.

Need a slightly more general definition to include function values:

- Expressions are *extensionally equivalent if they have the same type and one of the following is true:*
  both expressions reduce to equivalent values,
  or both expressions raise equivalent exceptions,
  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: $21 + 21 \cong 42 \cong 6 * 7$

  $[2, 7, 6] \cong [1+1, 2+5, 3+3]$

  $(fn\ x => x + x) \cong (fn\ y => 2 * y)$

- 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

Base 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),   #1 e,   #2 e

DO NOT USE! *

Examples: $(3 + 4, true)$

$(1.0, \sim 15.6)$

$(8, 5, false, \sim 2)$

* 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

- $$(e_1,\ e_2) \overset{1}{\Longrightarrow} (e_1',\ e_2) \quad \text{if } e_1 \overset{1}{\Longrightarrow} e_1'$$

- $$(v_1,\ e_2) \overset{1}{\Longrightarrow} (v_1,\ e_2') \quad \text{if } e_2 \overset{1}{\Longrightarrow} e_2'$$

What are the type & value of ...

(3 * 4, 1.1 + 7.2, true)

Type reasoning

$3 * 4 \quad : \text{int}$

$1.1 + 7.2 : \text{real}$

$\text{true} \qquad : \text{bool}$

So $\quad (3 * 4, \; 1.1 + 7.2, \; \text{true})$

$\qquad\qquad : \text{int} * \text{real} * \text{bool}$

Evaluation

$$(3 * 4, 1.1 + 7.2, true)$$

$$\Rightarrow (12, 1.1 + 7.2, true)$$

$$\Rightarrow (12, 8.3, true)$$

That is a value, so

$$(3 * 4, 1.1 + 7.2, true)$$

$$\hookrightarrow (12, 8.3, true)$$

What are the type & value of ...

$$(3 * 4, 1.1 + 7.2, true)$$

$$(3*4, 1.1+7.2, true) : int * real * bool$$

$$(3*4, 1.1+7.2, true) \hookrightarrow (12, 8.3, true)$$

What are the type & value of ...

$$(5 \text{ div } 0, \ 2 + 1)$$

$(5 \text{ div } 0, \ 2+1)$: int * int

$(5 \text{ div } 0, \ 2+1)$ does **not** reduce to a value, because evaluation of 5 div 0 raises an exception.

What are the type & value of ...

$$(8 + \text{"miles"}, \ \text{false})$$

This expression is ill-typed, i.e., it has no type, because the subexpression $8 + \text{"miles"}$ is ill-typed.

SML does not evaluate ill-typed expressions, so the expression has no value.

What are the type & value of ...

$$(2, (true, "a"), 3.1)$$

What are the type & value of ...

$$(2, (\text{true}, \text{"a"}), 3.1)$$

This expression has type $int * (bool * string) * real$, which is <u>different</u> from $int * bool * string * real$.

Contrast:

$$(2, (\text{true}, \text{"a"}), 3.1) : int * (bool * string) * real$$

vs. $\quad (2, \text{true}, \text{"a"}, 3.1) : int * bool * string * real$.

What are the type & value of ...

$$(2, (true, "a"), 3.1)$$

$(2, (true, "a"), 3.1) \; : \; int * (bool*string) * real$

$(2, (true, "a"), 3.1) \; \hookrightarrow \; (2, (true, "a"), 3.1)$

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.

Def: A function f is ***total***   if f reduces to a value[*]
        and f(x) reduces to a value for all values x in X.

[*] (With one unusual exception, this first condition is implied by the second.
We write it for emphasis, since f could be a general expression of type $X \to Y$.)

Functions

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

$$f \; : \; X \; \rightarrow \; Y$$

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.

Def: A function f is **total**   if f reduces to a value
and f(x) reduces to 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    function type
   REQUIRES: true
   ENSURES:  square(x) evaluates to x * x
*)

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

```
(* 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   argument   result   body of function
          name   name & type   type

(5) `(* Testcases: *)`

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

Six-Step Methodology

① ②③ (* square : int -> int function type
    REQUIRES: true
    ENSURES: square(x) evaluates to x * x
  *)

④ fun square (x:int) : int = x * x
keyword   function   argument   result   body of function
            name   name & type   type

⑤ (* Testcases: *)   ⑥

                                          Proof!

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

Declarations

Environments

Scope

Declaration

$$val \quad pi : real = 3.14$$

↑   ↑   ↑   ↑

keyword   identifier   type   value

Introduces binding of pi to 3.14
(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 the identifier **square** to a closure.

The closure consists of two parts:

- A lambda expression (code):

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

    keyword   argument name & type   body of function

- An environment (all prior bindings).

Closures

Function declarations also create value bindings:

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

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

The closure consists of two parts:

- A lambda expression (code):

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

keyword   argument name & type   body of function

CAUTION: Do NOT write return type.

- An environment (all prior bindings).

Closures

Function declarations also create value bindings:

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

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

The closure consists of two parts:

- A lambda expression (code):

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

keyword   argument name & type   body of function
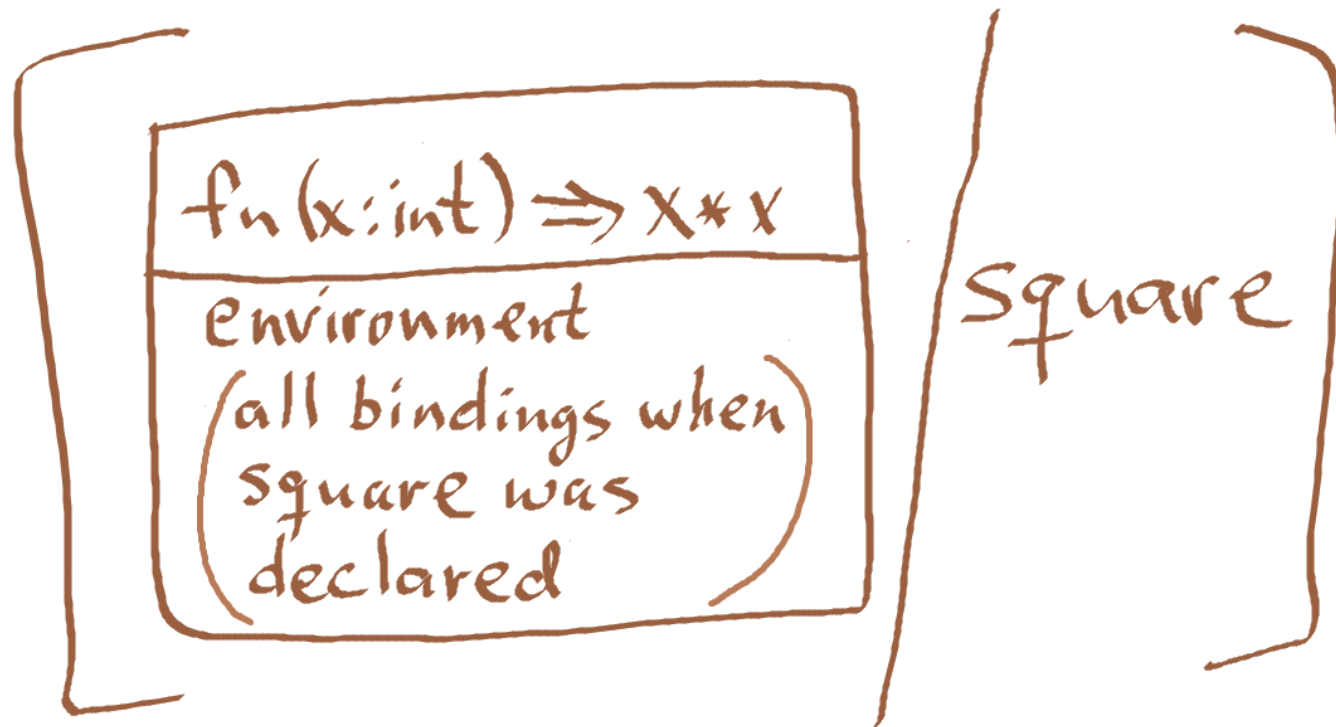
NOTE

CAUTION: Do NOT write return type.

- An environment (all prior bindings).

Closures

Function declarations also create value bindings:

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

binds the identifier **square** to a **closure**:

Course Tasks

- Assignments   45%
- Labs   10%
- Midterm 1   10%
- Midterm 2   15%
- Final   20%

Roughly one assignment per week, one lab per week.

Collaboration

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

TO DO **TONIGHT**

Go to 150's **Canvas**.

Select **Assignments**.

Do the **Setup Lab**.

(Important preparation **before** Wednesday's lab.)

(If you have questions, ask on 150's Piazza.)

That is all.

Have a good lab tomorrow.

See you Thursday.