15-150

Principles of Functional Programming

Michael Erdmann   Frank Pfenning
Miranda Lin    Helen Li    Harrison Grodin

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

<table>
<thead>
<tr>
<th>Imperative</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>Expression</td>
</tr>
<tr>
<td>• executed</td>
<td>• evaluated</td>
</tr>
<tr>
<td>• has an effect</td>
<td>• no effect</td>
</tr>
<tr>
<td>$x := 5$ (state)</td>
<td>$3 + 4$ (value)</td>
</tr>
</tbody>
</table>
Programming as Explanation

Problem statement

- high expectations to explain
- precisely and concisely

- invariants
- specifications
- proofs of correctness
- code

Analyse, Decompose & Fit, Prove
Parallelism

\[ <1, 0, 0, 1, 1 > \rightarrow 3, \]
\[ <1, 0, 1, 1, 0 > \rightarrow 3, \]
\[ <1, 1, 1, 0, 1 > \rightarrow 4, \]
\[ <0, 1, 1, 0, 0 > \rightarrow 2, \]
\[ \downarrow \]
\[ 12 \]
Parallelism

\[
\text{sum} : \text{int sequence} \rightarrow \text{int} \\
\text{type} \text{ row} = \text{int sequence} \\
\text{type} \text{ room} = \text{row sequence} \\
\]

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

\[(3 + 4) \times 2\]

\[7 \times 2\]

\[14\]

\[(3 + 4) \times (2 + 1)\]

\[21\]
"the " ^ "walrus"

1 ➞ "the walrus"

The expression
"the " ^ "walrus"

reduces to the value
"the walrus"

It has type string.
"the walrus" + 1

⇒ ??

The expression "the walrus" + 1 does not have a type and it does not reduce to a value.
Types

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

An expression is \textit{well-typed} if it has at least one type, and \textit{ill-typed} otherwise.

(We may also say that an expression \textit{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 $e : t$
- may have a value $v$, written as $e \rightarrow v$.
- may have an effect (not for our effect-free fragment)

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

$$
(3 + 4) \times 2 \quad \rightarrow 14
$$
Integers, Expressions

Type \textit{int}

Values \ldots, \overset{\sim}{1}, 0, 1, \ldots,
    that is, every integer \(n\).

Expressions \(e_1 + e_2, \ e_1 - e_2, \ e_1 \ast e_2, \ e_1 \ \text{div} \ e_2, \ e_1 \ \text{mod} \ e_2, \ \text{etc.}\)

Example: \(\overset{\sim}{4} \ast 3\)
Typing Rules

- $n : \text{int}$
- $e_1 + e_2 : \text{int}$
  
  if $e_1 : \text{int}$ and $e_2 : \text{int}$

*similar for other operations.*

Example:

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

Why?

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

Why?

$3 : \text{int}$ and $4 : \text{int}$
Integers, Evaluation

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

because \ 5 : \text{ int} \\
\_ \quad 0 : \text{ int}

and because \text{ div} \text{ expects two \text{ int}s and returns an \text{ int}}.

However, \ 5 \text{ div } 0 \\
does not reduce to a value.
Notation Recap

e : t  “e has type t”

e \to e'  “e reduces to e’”

e \to v  “e evaluates to v”
Extensional Equivalence

\[ \equiv \]

An equivalence relation on expressions (of the same type).
Extensioonal 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 $\equiv$ as shorthand for “is equivalent to”.

• Examples:
  
  $21 + 21 \equiv 42 \equiv 6 \times 7$
  
  $[2, 7, 6] \equiv [1+1, 2+5, 3+3]$
  
  $(\text{fn } x \mapsto x + x) \equiv (\text{fn } y \mapsto 2 \times y)$

• 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 \( t_1 \times t_2 \) 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 \( e \)

Examples:

\( (3 + 4, \text{true}) \)

\( (1.0, \sim 15.6) \)

\( (8, 5, \text{false}, \sim 2) \)

You will learn how to extract components using pattern matching.
Products, Typing

Typing Rules

- \((e_1, e_2) : t_1 \times t_2\)

  if \(e_1 : t_1\)

  and \(e_2 : t_2\)

Example: \((3+4, \text{true}) : \text{int} \times \text{bool}\)
Evaluation Rules

- \((e_1, e_2) \xrightarrow{1} (e'_1, e_2)\) if \(e_1 \xrightarrow{1} e'_1\)

- \((v_1, e_2) \xrightarrow{1} (v_1, e'_2)\) if \(e_2 \xrightarrow{1} e'_2\)
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.

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  function type
   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)
Five-Step Methodology

1. \((*\ square : \ int \rightarrow \ int\ function \ type\)\)
2. \((REQUIRES: \ true)\)
   \((ENSURES: \ square(x) \ evaluates \ to \ x \times x\)\)
3. \((fun\ square\ (x:\int) : \ int = x \times x\)\)
4. \((keyword\ function\ \ argument\ result\ body\ \ of\ \ function)\)
5. \((*\ Testcases: \ *)\)
   \((val\ 0 = square\ 0)\)
   \((val\ 49 = square\ 7)\)
   \((val\ 81 = square\ (\sim9))\)
Declarations

Environments

Scope
Declaration

\[
\text{val } \text{pi : real = 3.14}
\]

- Keyword
- Identifier
- Type
- Value

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

Lexically statically scoped.
val x : int = 8 - 5
val y : int = x + 1
val x : int = 10
val z : int = x + 1

[3/y]
[4/y]
[10/x]
[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

k
⇒ 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`.

[Diagram of a closure binding to `square`]
Closures

Function declarations also create value bindings:

```java
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):
  ```java
  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:

```plaintext
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.