15-150

Summer 2016

(this time, in reverse)
1. You will learn to write parallel functional programs.
2. You will learn to analyze the sequential and parallel running time of your programs.
3. You will learn to reason mathematically about the correctness of functional programs.
4. You will learn to structure functional programs using abstract types.
1. You have learned to write parallel functional programs.
2. You have learned to analyze the sequential and parallel running time of your programs.
3. You have learned to reason mathematically about the correctness of functional programs.
4. You have learned to structure functional programs using abstract types.
val unique : int comp = get >>= (fn s => (set (s+1)) >>= (fn () => get >>= (fn s => return s )))

- run unique 0
val it = 1 : int
Monads

- Imperative programs can be represented as functional programs.
- Push effects to the edge of a program.
- Example of a type-class.
fun reachable g start goal = 
  let
    val visited = ref Visited.empty

    fun dfs (cur : G.node) : bool = 
      case G.NodeSet.El.compare (cur, goal) of
        EQUAL => true 
      | _ => case Visited.member (!visited) cur of
                true => false 
              | false => (visited := (Visited.insert (! visited) cur);
                          (G.NodeSet.exists dfs (G.successors g cur)))

    in
      dfs start
    end
Benign Effects

Imperative implementation, but looks persistent to clients.

Can be faster or easier than functional code.

Hard to do correctly...
signature EPH_GAME =

sig
  type state
  type move
  val make_move : (state * move) -> unit

end
Ephemeral Data Structures

More freedom in the implementation.

Harder to backtrack or run in parallel.
## Using Effects

<table>
<thead>
<tr>
<th></th>
<th>Persistent</th>
<th>Ephemeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>benign effects</td>
<td>concurrency?</td>
</tr>
<tr>
<td>Sequential</td>
<td>benign effects</td>
<td>OK</td>
</tr>
</tbody>
</table>
fun update (f : 'a -> 'a) (r : 'a ref) : unit =  
    let val (ref cur) = r in r := f cur end
fun deposit n a = update (fn x => x + n) a
fun withdraw n a = update (fn x => x - n) a

val account = ref 100
val () = deposit 100 account
val () = withdraw 50 account
val _ = Seq.tabulate (fn x => case x of
    0 => deposit 100 account
    | 1 => withdraw 50 account) 2

(* race condition *)
Parallelism and Effects

- Total functional programming ✓
- Non-termination ✓
- Exceptions ✓
- I/O ?
- Mutation ?
fun search (d : int) (s : Game.state) : Game.est = 
  let val moves = Game.moves s 
  in 
  if Seq.null moves then score s else 
  if d = 0 then Game.estimate s else 
  case Game.player s of 
    Game.Minnie => 
      Seq.reduce1 vmin (Seq.map (fn mv => search (d - 1) 
        (Game.make_move (s, mv)))) (moves)) 
    | Game.Maxie => 
      Seq.reduce1 vmax (Seq.map (fn mv => search (d - 1) 
        (Game.make_move (s, mv)))) (moves)) 
  end 

fun next_move (s : Game.state) : Game.move = 
  let val moves = Game.moves s 
  in 
  case Game.player s of 
    Game.Minnie => 
      moveOf (Seq.reduce1 min 
        (Seq.map (fn mv => (mv, search (Settings.search_depth - 1) 
          (Game.make_move (s, mv)))))) moves)) 
    | Game.Maxie => 
      moveOf (Seq.reduce1 max 
        (Seq.map (fn mv => (mv, search (Settings.search_depth - 1) 
          (Game.make_move (s, mv)))))) moves)) 
  end
Persistent Data Structures

Naturally support parallelism and backtracking.
functor RBTDict (Key : ORDERED) : DICT =
struct
  structure Key : ORDERED = Key
  datatype color = Red | Black
  datatype 'v tree =
      Empty
    | Node of 'v tree * (color * (Key.t * 'v)) * 'v tree
  type 'v dict = 'v tree (* representation invariant: is a RBT *)
(* ... *)
end
Abstract types enable local reasoning about representation invariants.

To make a type abstract, define it to be a datatype that is not exported in the signature.
fun insert d (k, v) = 
  let 
  (* Root is Red, both RBT \rightarrow ARBT; *) 
  (* Root is Black, at most one ARBT, and the other(s) RBT \rightarrow RBT; *) 
  (* if both args have the same black--height, then so does the result. *) 
  (* if d is an RBT[Red] then ins d is an ARBT; *) 
  (* if d is an RBT[Black] then ins d is an RBT; *) 
  (* preserves the black--height. *) 
  fun balance p = 
    case p of 
    (Node(Node (a, (Red, x), b), (Red, y), c), (Black, z), d) => 
      Node(Node (a, (Black, x), b), (Red, y), Node (c, (Black, z), d)) 
    | (Node(a, (Red, x), Node (b, (Red, y), c)), (Black, z), d) => 
      Node(Node (a, (Black, x), b), (Red, y), Node (c, (Black, z), d)) 
    | (Node(Node (a, (Black, x), b), (Red, y), Node (c, (Red, z), d)), (Black, z), d) => 
      Node(Node (a, (Black, x), b), (Red, y), Node (c, (Black, z), d)) 
    | (Node(Node (a, (Black, x), b), (Red, y), Node (c, (Red, z), d))), (Black, z), d)) => 
      Node(Node (a, (Black, x), b), (Red, y), Node (c, (Black, z), d)) 
    | _ => Node p 
  in 
  fun ins d = 
    case d of 
    Empty => Node (Empty, (Red, (k, v)), Empty) 
  | Node (l, (c, (k', v')), r) => 
    case Key.compare (k,k') of 
    EQUAL => Node (l, (c, (k, v)), r) 
    | LESS => balance (ins l, (c, (k', v')), r) 
    | GREATER => balance (l, (c, (k', v')), ins r) 
  in 
  blackenRoot (ins d) 
end
functors

functor TreeDict(K : ORDERED) : DICT =
struct
  structure Key : ORDERED = K

  datatype 'v tree =
      Empty
    | Node of 'v tree * (Key.t * 'v) * 'v tree

  type 'v dict = 'v tree

  val empty = Empty

  fun lookup d k =
    case d of
      Empty => NONE
    | Node (L, (k', v'), R) =>
        case Key.compare (k,k') of
            EQUAL => SOME v'
            | LESS => lookup L k
            | GREATER => lookup R k

  fun insert d (k, v) =
    case d of
      Empty => Node (empty, (k,v), empty)
    | Node (L, (k', v'), R) =>
        case Key.compare (k,k') of
            EQUAL => Node (L, (k, v), R)
            | LESS => Node (insert L (k, v), (k', v'), R)
            | GREATER => Node (L, (k', v'), insert R (k, v))
end
Functors

Take a structure as an argument, returns a structure as a result. Both structures can contain types, values, and more structures.

(Can’t insert into a sorted tree without Key.compare.)

(Can’t Referee.go() without TWO_PLAYERS.)
Type Classes

signature ORDERED =
sig
  type t
  val compare : t * t -> order
end

structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end
Type Classes

A kind of signature—describes a non-abstract(!) type with (some) operations it supports.
Abstract Types

signature SEQUENCE =

sig
  type 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val nth : int -> 'a seq -> 'a
  (* ... *)
end
Abstract Types

Sequences provide parallelism via bulk operations on data.
Abstract Types

signature SPACE =
sig
  structure Scalar : SCALAR
  structure Seq : SEQUENCE
  type scalar = Scalar.scalar

  type point
  type vec

  val ++ : vec * vec -> vec
  val ** : vec * scalar -> vec
  val --> : point * point -> vec

(* ... *)
end
Abstract Types

Allow clients and implementations to evolve separately.

Localize reasoning. Correct once...correct forever.
Cost Graphs

Diagram:
- a
- b
- c
- d
- e
- f
- g
- h
- i
- j

The diagram represents a cost graph with nodes a, b, c, d, e, f, g, h, i, and j. The edges indicate the cost relationships between these nodes.
Cost Graphs

Separate generating the work from scheduling it on processors.

- **Work** is the number of nodes.
- **Span** is the depth of the tree.
- **Brent’s principle** relates these to $p$-processor time.
- Demonstrates that span is a meaningful measurement of parallel time.
exception NoOdd

(* returns the leftmost odd number in the tree, *
* or raises NoOdd if there is no odd number *)

fun findOdd (t : int tree) : int =
  case t of
    Empty => raise NoOdd
  | Leaf x => (case (x mod 2) = 1 of
                true => x
                false => raise NoOdd)
  | Node(l,r) => (findOdd l)
               handle NoOdd => findOdd r
Exceptions

Related to options.

Options: If you don’t check \texttt{NONE}, it’s a compile-time type error.

Exceptions: If you don’t handle them, it’s a runtime error.
  
  ▶ Act as though failures don’t happen while coding...
  ▶ Just don’t forget to handle them at the end!
Exceptions

Raising an exception is different from evaluating to a value

A raise can have any type.

Interesting control flow (similar to continuations)
fun staged_exp (e : int) : int -> int =
  case e of
    0 => (fn _ => 1)
  | _ => let val oneless = staged_exp (e-1)
    in
      fn b => b * oneless b
    end
Curried functions can do useful work before receiving all their arguments!

```ocaml
staged_exp 2
|->* let val oneless = (staged_exp (2-1)) in fn b => b * oneless b end
|->* let val oneless = (let val oneless = staged_exp (1-1) in f end) in f end
|->* let val oneless = (let val oneless = (fn _ => 1) in f end) in f end
|-> fn b => b * ((fn b => b * ((fn _ => 1) b)) b)
```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
  Zero => false
  | One => k cs
  | Char c => (case cs of
    [ ] => false
    | c' :: cs' => chareq(c,c') andalso k cs')
  | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
  | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
  | Star r =>
    let fun matchstar cs' = k cs' orelse match r cs' matchstar
    in
    matchstar cs
  end

fun accepts (r : regexp) (s : string) =
  match r (String.explode s) (fn l => case l of [ ] => true | _ => false)
Regular Expressions

It takes real mathematical sophistication to get code right!

Programming and proving—find bugs by failing to prove correctness.

Higher-order functions capture control flow as data. Complex continuations (matchstar) for complex control flow.
fun sum_cont (l : int list) (k : int -> int) : int = 
  case l of
  [] => k 0
| x :: xs => sum_cont xs (fn s => k(x + s))

fun sum l = sum_cont l (fn x => x)
Higher-Order Functions: Functions as Data

```
type array = int -> int option

fun init (i : int) (v : int) : array = fn n =>
  case n < i of
    true => SOME v
  | false => NONE

fun get (i : int) (f : array) : int option = f i
```
Higher-Order Functions: Functions as Arguments

fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
  case l of
    [] => []
  | x :: xs => f x :: map f xs
val bestGain : int list -> int =
  maxAll
  o (map (fn (buy,sells) =>
          (map (fn sell => sell - buy) sells)))
  o withSuffixes
Higher-order functions abstract \textit{patterns of computation}. Express algorithms as \textit{compositions of functions}. 
Datatypes

```
datatype 'a option = NONE | SOME of 'a

datatype order = EQUAL | LESS | GREATER

datatype 'a tree = Empty
  | Leaf of 'a
  | Node of ('a tree * 'a tree)
```
Define a type that *exactly* matches a given problem.

More precise types mean less bugs.

Make invalid states *impossible*. (The “billion dollar mistake”.)

Recursive functions for recursive data.
fun splitAt (t : tree, bound : int) : tree * tree = 
  case t of 
    Empty => (Empty , Empty) 
  | Node (l , x , r) => 
      (case bound < x of 
        true => let val (ll , lr) = splitAt (l , bound) 
          in (ll , Node (lr , x , r)) 
          end 
      | false => let val (rl , rr) = splitAt (r , bound) 
          in (Node (l , x , rl) , rr) 
          end)

fun merge (t1 : tree, t2 : tree) : tree = 
  case t1 of 
    Empty => t2 
  | Node (l1 , x , r1) => 
      let val (l2 , r2) = splitAt (t2 , x) 
        in 
          Node (merge (l1 , l2) , x, merge (r1 , r2)) 
        end 

fun mergesort (t : tree) : tree = 
  case t of 
    Empty => Empty 
  | Node (l , x , r) => 
      merge(merge (mergesort l , mergesort r), 
        Node(Empty,x,Empty))
Trees are better than lists for parallelism.

Reason abstractly about sequential and parallel complexity.
Work and Span

\[ W_{\text{mergesort}}(0) = k_0 \]

\[ W_{\text{mergesort}}(n) = k_1 + W_{\text{split}}(n) + 2 \cdot W_{\text{mergesort}}(n/2) + W_{\text{merge}}(n/2 + n/2) \]

\[ S_{\text{mergesort}}(0) = k_2 \]

\[ S_{\text{mergesort}}(n) = k_3 + \max \left( S_{\text{mergesort}} \left( \frac{n}{2} \right), S_{\text{mergesort}} \left( \frac{n}{2} \right) \right) + S_{\text{merge}}(\log n, \log n) + S_{\text{merge}}(2 \log n, 1) \]
The recurrence has the same recursive structure as the code!

Use big-O notation to describe the asymptotic behavior of the recurrence.
fun sum (l : int list) : int = 
  case l of 
    [] => 0 
  | x :: xs => x + sum xs
Once upon a time, you didn’t know what recursively-defined lists were!
fun double (n : int) : int = 
  case n of 
    0 => 0 
  | _  => 2 + (double (n - 1))
Recursion

Once upon a time, you didn’t know how to write simple recursive functions!
Correctness

Theorem
For all natural numbers $n$, $\text{double } n \approx 2n$.

Proof.
Base case: $\text{double } 0 \approx 2 \times 0$.

Inductive case: Assume $\text{double } n \approx 2n$. Show $\text{double } (n + 1) \approx 2(n + 1)$. 

The proof has the same **recursive structure** as the code!

Use **structural induction** to reason about structural recursion.

Use **equivalence** to reason about the behavior of a program.
Correctness

Don’t forget basic logic:

<table>
<thead>
<tr>
<th>Proposition</th>
<th>To Prove</th>
<th>To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \land Q$</td>
<td>Prove both</td>
<td>Assume both</td>
</tr>
<tr>
<td>$P \lor Q$</td>
<td>Prove either</td>
<td>Case on P vs. Q</td>
</tr>
<tr>
<td>$P \implies Q$</td>
<td>Assume P, show Q</td>
<td>Show P, get Q</td>
</tr>
<tr>
<td>$\exists x. P$</td>
<td>Pick x, prove P</td>
<td>Assume x, assume P</td>
</tr>
<tr>
<td>$\forall x. P$</td>
<td>Assume x, prove P</td>
<td>pick x, get P</td>
</tr>
<tr>
<td>$\forall x : t. P$</td>
<td>Induct on t</td>
<td>pick x, get P</td>
</tr>
</tbody>
</table>
Typing and Evaluation

2 : int
1 + 1 : int
(1 + 2) * (3 + 4) : int
"I am" : string
"I am" ^ " Iron Man" : string
intToString 5 : string
"Iron Man" + 1 is ill-typed : int
5 div 0 : int
Typing and Evaluation

2
1 + 1
(1 + 2) * (3 + 4)
"I am"
"I am" ^ "Iron Man"
intToString 5
"Iron Man" + 1
5 div 0

==> 2
==> 2
==> 21
==> "I am"
==> "I am Iron Man"
no value!
no value!
The Hierarchy of Programs

Arbitrary Text: e.g. ))(()(()(*)) +a’’’+’’+,a]qj

Syntactically Valid: 2 + "2 is 5"

Well-Typed: raise Div

Valuable: 2 + 2

Values: 4
Types are predictions about values

\[ e : t \] If \( e \) terminates, it will be a value for type \( t \)

Might not terminate! That’s okay!

Checked *statically*. Checked *syntactically*. 

Typing and Evaluation
Parallel Counting

\[
\text{type} \quad \text{row} = \text{int sequence} \\
\text{type} \quad \text{students} = \text{row sequence} \\
\text{fun} \quad \text{sum} \ (s : \text{int sequence}) = \text{reduce} \ (\text{op}+) \ 0 \ s \\
\text{fun} \quad \text{count} \ (s : \text{students}) : \text{int} = \text{sum} \ (\text{map sum} \ s)
\]
1. You have learned to write parallel functional programs.
2. You have learned to analyze the sequential and parallel running time of your programs.
3. You have learned to reason mathematically about the correctness of functional programs.
4. You have learned to structure functional programs using abstract types.
Where Next?

- 15–210 Parallel and Sequential Data Structures and Algorithms
- 15–312 Foundations of Programming Languages
- 15–317 Constructive Logic
- 15–417 HOT Compilation (How does SML realllly work?)
- 15–424 Foundations of Cyber-Physical Systems
  Prove things about safety-critical systems
  (because I’ll be TA’ing it in the Spring)
- 15–150 is always hiring TA’s!
Who Uses Functional Programming?

- Compilers
- Theorem provers, verification (HOL, Coq, Agda, ACL2...)
- People who:
  - want correct code
    - Finance: Jane Street
    - Consulting: Galois...
    - Telecom: Ericsson
    - Web: Netflix, Facebook, Google ITA
  - want elegant solutions to hard problems (Comp. Bio, AI)
  - want a simple language (Video Games, Engineering, Text-Editing)
  - want robust parallelism (Distributed Systems)
  - want to understand human language (Linguists!)
  - think code is art (Computer Music)
  - like theorems (Category theorists, Algebraic topologists)
Which Languages?

- OCaml (very similar to SML)
- Haskell (call-by-name; pure, plus monads)
- Scheme (untyped)
- Kappa (Comp. bio)
- Nyquist (Computer music)
- AUTOLISP, elisp, GOOL (Games, CAD, Text-Editing)
- Erlang (Ericsson, Netflix, Facebook)
Which Languages?

- F# (ML + objects, on .NET)
- Scala (ML + objects, on JVM)
- Java, C++, ... (lambdas, generics)
- ... and more (garbage collection)
Code is Math

Functional programs are mathematical transformations on data.

Functional programs are subject to mathematical analysis.
Code is Art

Code can be beautiful.

Code is for people.

Code can change the way you think.
Code is Math / Code is Art