Semester Review

A long list of things you already know

15-150 M21

Lecture 0804
04 August 2021

0 Reasoning About Code

Reasoning About Code

- *Mathematically* articulate the structure of our code
- Deduce its properties

Evaluation

- Value:

```
fn () => 1 div 0
```

- Valuable:

```
let val x = 2+2 in fn () => x div 0 end
```

- Raises exception:

```
(fn () => 1 div 0) ()
```

- Loops forever:

```
let fun loop() = loop() in loop() end
```

Extensional Equivalence & Referential Transparency

Two well-typed expressions are said to be **extensionally equivalent** if they have the same type and either

- They both evaluate to the same value
- They both raise the same exception
- They both loop forever

Referential Transparency: If `e1` $\cong$ `e2` (and both are pure), then any instance of `e1` can be replaced with `e2` (and vice versa) without changing the overall behavior of the code.

Swap out with a better implementation

```
fun exp 0 = 1
  | exp n = 2 * exp(n-1)
```

```
fun pow 0 = 1
  | pow n =
      case (isEven n) of
        true => square(pow (n div 2))
      | false => 2*square(pow (n div 2))
```

Prop. For all $\mathtt{n} \geq 0$,

$$\mathtt{exp\ n} \quad \cong \quad \mathtt{pow\ n}$$

Totality & the Valuable Stepping Principle

A *total* function is one which is guaranteed to evaluate to a value when applied to *any* value of the input type

$$
\begin{aligned}
&\texttt{map g (f(x)::map f xs)} \\
&\cong \texttt{g(f(x)) :: map g (map f xs)} \\
&\qquad\qquad\qquad \text{(defn } \texttt{map}\text{, totality of } \texttt{f}\text{,totality of } \texttt{map f}\text{)}
\end{aligned}
$$

If a function `f` is not assumed to be total, then we need to justify this kind of steps with lengthy reasoning about why the two sides are extensionally equivalent. If we can avoid this, it's nice to. We know that we can obtain extensional equivalences by stepping through code, treating valuable expressions as values

Effects

- Exceptions are a kind of effect:

```
(fn _ => raise Fail "Unimplemented") [1,2]
```

  This expression doesn't evaluate to a value!

- We also have actual effects:

```
r := 1
```

Reasoning about effects makes the code more complicated!

Cost Semantics

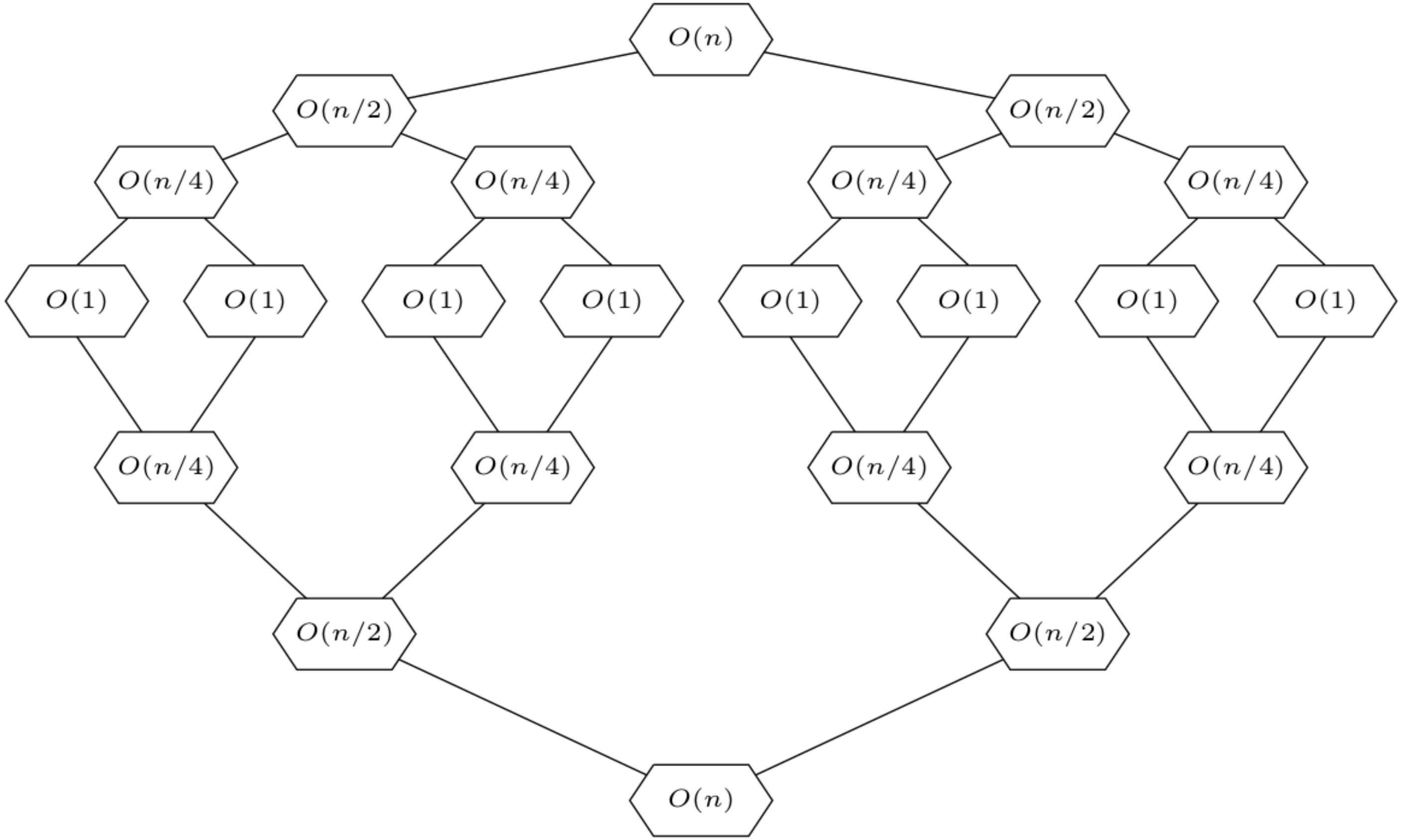
Another way we reason mathematically about code: quantifying the runtime.

$$W_{\texttt{msort}}(n) \text{ is } O(n \log n)$$

In addition to the sequential runtime (work), we had the parallel runtime (span) which assumed we took advantage of every opportunity for parallelism, and had unlimited processors.

$$S_{\texttt{msort}}(n) \text{ is } O(\log^2(n))$$

Algorithmic Design



1 Recursion

Datatypes, Pattern Matching, and Recursion

- Recursively construct data:

```
datatype 'a list =
    [] | :: of 'a * 'a list
datatype 'a tree =
    Empty | Node of 'a tree * 'a * 'a tree
```

- Pattern match to recursively deconstruct:

```
fun foo [] = ...
  | foo (x::xs) = ... foo xs ...
```

- Inductively establish correctness
- Solve for runtime by recurrence

Higher Order Functions

```
fun map f [] = []
  | map f (x::xs) =
      (f x)::map f xs
fun filter p [] = []
  | filter p (x::xs) =
      if (p x)
      then x::filter p xs
      else filter p xs
fun foldl g z [] = z
  | foldl g z (x::xs) =
      foldl g (g(x,z)) xs
```

Structural Induction

IS `T=Node(L,x,R)` for some values `L,R:tree` and `x:int`

IH1 `rev(inord L)` $\cong$ `inord(revTree L)`

IH2 `rev(inord R)` $\cong$ `inord(revTree R)`

`rev(inord (Node(L,x,R)))`

$\cong$ `rev((inord L)@(x::(inord R)))` (defn `inord`)

$\cong$ `(rev (x::inord R)) @ (rev(inord L))` (**Lemma 1,2**)

$\cong$ `((rev (inord R))@[x]) @ (rev(inord L))`

(**Lemma 2** , defn of `rev`)

$\cong$ `(rev (inord R))@(x::(rev(inord L)))` (**Lemma 2,3,4**)

1 Recurrence:

$$W(0) = k_0$$
$$W(n) = k_1 + k_2 n + W(n-1)$$

2 Work Tree

3 Measurements

Height: n Work on the i-th level: $k_1 + k_2(n-i)$

4 Sum:

$$W(n) \approx k_0 + \sum_{i=0}^{n} (k_1 + k_2(n-i)) = \dots$$

5 Big O:

$$W(n) \text{ is } O(n^2)$$

Converting to tail-recursive form

`texp : int * int -> int`

REQUIRES: $\texttt{n} \geq 0$

ENSURES: $\texttt{texp(n,acc)} \cong \texttt{acc * 2}^{\texttt{n}}$

```
fun texp (0,acc) = acc
  | texp (n,acc) = texp(n-1,2*acc)
```

Functions Are Accumulators

```
factCPS : int -> (int -> 'a) -> 'a
```

REQUIRES: `n` ≥ 0

ENSURES: `factCPS n k` $\cong$ `k(fact n)`

```
fun factCPS 0 k = k 1
  | factCPS n k =
       factCPS (n-1) (fn res => k(n*res))
```

2 Data Representation

Datatypes

- Options

```
fun hd [] = NONE
  | hd (x::_) = SOME x
```

- Order

```
case Int.compare(x,y) of
  LESS => ...
| EQUAL => ...
| GREATER => ...
```

- Extended integers

```
datatype int' = NEGINF
              | FIN of int
              | POSINF
```

Representing Regular Expressions

```
datatype ''a regexp =
    Const of ''a
  | One
  | Zero
  | Times of ''a regexp * ''a regexp
  | Plus of ''a regexp * ''a regexp
  | Star of ''a regexp
```

```
match : ''a regexp
         -> ''a list
         -> (''a list * ''a list -> 'b)
         -> 'b
```

Data Structures

```
structure LLQ :> QUEUE =
struct
  (* INVARIANT: if (f,b):'a queue, then the
     list f@(rev b) lists the elements of the
     queue in their queueing order *)
  type 'a queue = ('a list * 'a list)

  ...
```

3 Abstraction

Abstraction

Idea:

Make functions more general by "abstracting" away details: replace by variable name, and take in a value for that variable as an argument.

Polymorphism

```
fun len ([] : 'a list):int = 0
  | len (x::xs) = 1+(len xs)
```

The `'a` can be instantiated with whatever type we want!

The value `fn (x,y)=>y` can be used as a value of type `int * int -> int`, or `string*bool -> bool`, and so on.

Lambda Abstraction

```
fn f => fn x => fn y => f(x,y)
```

The `f` can be instantiated with whatever value we want (if its MGT is an instance of `'a * 'b -> 'c`)!

- Lambda abstract comparison function

  ```
  fun merge cmp (L1,L2) = ...
  ```

- Lambda abstract predicate function

  ```
  fun filter p L = ...
  ```

- Lambda abstract other function

  ```
  fun map f L = ...
  ```

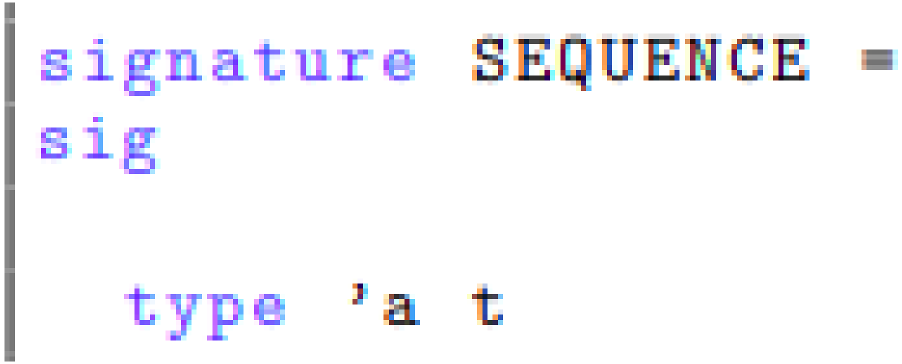
Lambda Abstract Typeclasses

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

functor OrdTreeSet(Elt : ORD) : SET =
  struct
    ...
```

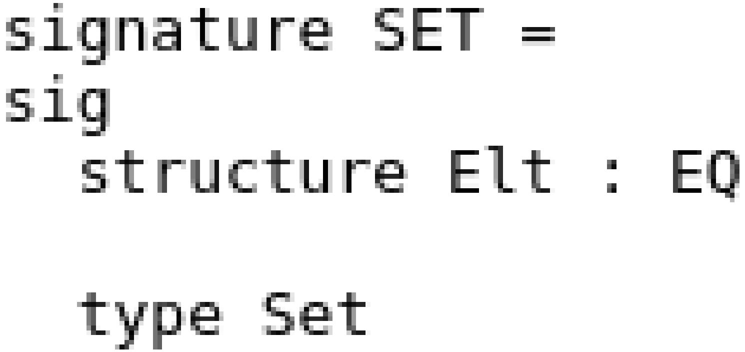
Abstract Types

We had the notion of an *abstract type*



```
signature SEQUENCE =
sig

  type 'a t
```



```
signature SET =
sig
  structure Elt : EQ

  type Set
```

Especially when opaquely ascribed, we don't know (and often don't care) what type this is implemented as. We instead just work with it based on the signature & documentation

4 Suspension and Control

Lambda Suspension

Functions are values. One of the things we mean by this statement is the fact that well-typed expressions of the form

```
fn x => e
```

are values. Therefore, `e` does not get evaluated until this function value is *applied*(the evaluation of `e` is “*suspended* behind the lambda”).

We use suspended computations for a variety of purposes.

CPS Control Flow

```
fun search p Empty sc fc = fc ()
  | search p (Node(L,x,R)) sc fc =
      if p x then sc x else
      search p L sc (fn () => search p R sc fc)
```

```
fun search p Empty sc fc = fc ()
  | search p (Node(L,x,R)) sc fc =
      if p x then sc(x,[]) else
      search p L
      (fn (res,dirs) => sc(res,Left::dirs))
      (fn () =>
        search p R
        (fn (res,dirs) =>sc(res,Right::dirs) )
```

Super CPS

```
fun iterate (check : 'a -> result)
      (L : 'a list) (combine : 'a -> 'b -> 'b)
      (base : 'b) (success : 'a -> 'c)
    (panic : string -> 'c) (return : 'b -> 'c)
        : 'c  =
let
  fun run ([] : 'a list) (k:'b -> 'c) : 'c =
        k base
    | run (x::xs) k = (case (check x) of
            Accept => success x
        |     Keep => run xs (k o (combine x))
        |  Discard => run xs k
        | (Break s)=> panic s)
```

Exceptions

```
exception NotFound
fun search p Empty sc = raise NotFound
  | search p (Node(L,x,R)) sc =
      (if p x then sc x else
       search p L sc)
       handle NotFound =>
         search p R sc
```

Laziness

```
datatype 'a stream =
       Stream of unit -> 'a front
     and 'a front =
       Nil | Cons of 'a * 'a stream
```

```
fun natsFrom k =
      Stream.delay(fn () => natsFrom' k)
and natsFrom' k =
      Stream.cons(k,natsFrom (k+1))
val nats = natsFrom 0
```

Conclusions

- *Think* about code
- Do incredible things.

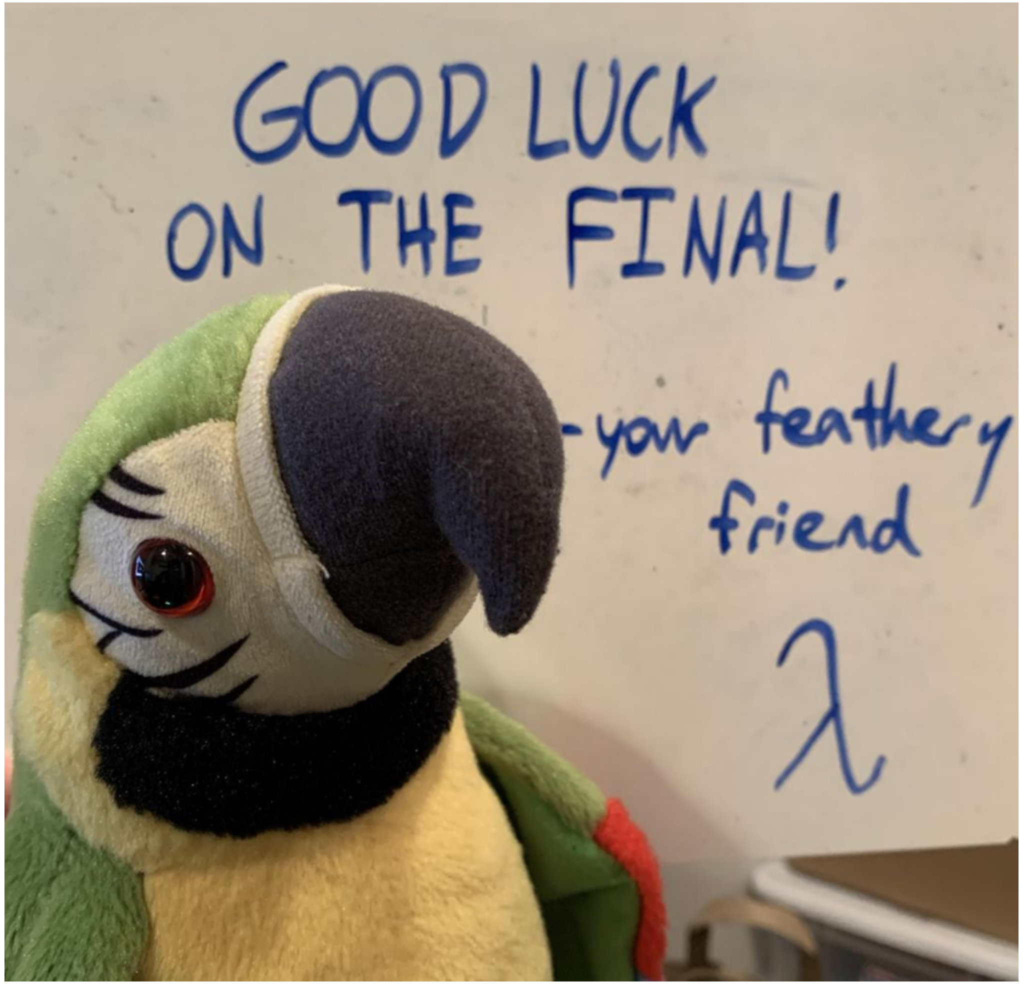
Advice for studying/taking exam

- Write your own review lecture & final (try to come up with your own examples of the phenomena we talked about)
- Ask questions in OH and on Piazza
- I'll try to put up as much up-to-date scores as possible, but time spent running numbers is time wasted.
- I believe that all of you learned functional programming and I want to give you a good grade. I just need an excuse to do so...

end

So,

- Thank you for being amazing
- Good luck on the final (you got this!)
- Come to the optional lectures
- Relax & enjoy the rest of your summer



THANK YOU!