CS3110 Final Review

12/3/2018

Streams

- How we read infinite data structures (i.e. user/file input, mathematical sequences, infinite game trees)
- Need a buffer (unit) so we don't get a memory overflow
- Use pattern matching

```
type ‘a stream =
| Cons of ‘a * (unit -> ‘a stream)
```

Laziness

- When we need the next stream value, we don’t want to recompute it every time
- Laziness lets us cache the value in memory to save time
- i.e. Fibonacci sequence - with laziness we save the prior values, so finding the next in the sequence is fast

Binary Search Trees

- Ordered tree s.t. a node's left child is smaller than it, and it's right child is greater
- Recursively make your way down the tree with time O(log n)
- Include a balancing operation to keep operations efficient

BST

- Each node has a value and a left and right child

```
type ‘a tree = Node of ‘a * ‘a tree * ‘a tree | Leaf

(* Insert operation *)
let rec insert x = function
| Leaf -> Node (x, Leaf, Leaf)
| Node (y, l, r) as t ->
  if x = y then t
  else if x < y then Node (y, insert x l, r)
  Else Node (y, l, insert x r)
```

Red-Black Trees

- Like BST, but have rules about node coloring:
  - No two adjacent red nodes on a path
  - Each path from root to leaf has the same black height

RBT

```
type color = Red | Black
type ‘a rbtree =
  Node of color * ‘a * ‘a rbtree * ‘a rbtree | Leaf

let let rec mem x = function
| Leaf -> false
| Node (_, y, left, right) ->
    x = y
    || (x < y  && mem x left)
    || (x > y && mem x right)
```

References

- Like a pointer to memory: lets us work with mutable values!
- OCaml `ref` keyword (`let x = ref 3110`)
- Dereferencing: `!x` returns `3110`
- Assignment: `x := 2110` changes the contents of x
- Aliasing: `let y = x` now x and y point to the same memory location
- Use `;` to end operations with side effects
- Use `==` to check if pointers share a memory location (physical equality)

Mutable Records

- You can update records in place with `mutable` keyword

```
type date = {mutable d:int, mutable m:int, mutable y:int}
let today = {d=3, m=12, y=2018}
today.d <- 4  (* update day *)
```

- Refs are defined using `mutable`:

```
type ‘a ref = {mutable contents : ‘a; }
```

Concurrency

- We can have many operations happening at once (interleaving or parallelism)
- Threads: sequential computations to be carried out at the same time
- Promises are used for concurrency: “promise” to complete a computation in the future
  - `Async` and `Lwt`

Promises (Lwt)

- Promises can only mutate once
- 2 states: pending vs. resolved/rejected
- Resolver associated with each promise (hidden from client)
  - invoked to resolve the promise

```
#require “lwt”
let (p : int Lwt.t), r = Lwt.wait()
          (* makes promise of an int and a resolver *)
Lwt.state p  (* Currently Sleepl *)
Lwt.wakeup r 3110
Lwt.state p (* Currently Return 3110 *)
```

- The promise has been resolved, so we can’t change it any further
- Alternatively, could reject using `Lwt.wakeup_exn (Failure “<exception goes here>”)`

Monads

- Structures that bind inputs into “boxes”
- Must have the following signature:

```
module type Monad = sig
  type ‘a t
  val return : ‘a -> ‘a t
  val (>>=) : ‘a t -> (‘a -> ‘b t) -> ‘b t
end
```

- `Options` are an example of monads
- Values must be binded the same way to be used in the same operation
- i.e. `Some(10) + 11` will not work

Loggable Functions

- We can use monads to let us log faults and pass them through computations
- This is our bind operation

```
let log name f = fun x -> (f x, Printf.sprintf
  “Called %s on %i; “, name x)

let loggable name f  =
  fun (x, s1) ->
    let (y, s2) = log name f x in (y, s1 ^ s2)

let inc = loggable “inc” (let inc x = x + 1)
```

Compilers

- Translate a higher-level to lower-level language
- Phases:
  - Lexing (source code to tokens)
  - Parsing (tokens to abstract syntax tree (AST))
  - Semantic analysis (like type checking)
  - Translation (AST to intermediate representation (IR))
  - Target Code Generation (IR to low-level language)

Interpreters, Substitution Model

- Directly execute higher-level programs
- Does the lexing, parsing, semantic analysis,translation, and execution

Example of AST from dis-19:

```
type exp =
| Int of int
| Var of var
| Add of exp * exp

Type stm =
| Assign of string * exp
| Seq of stm * stm
| Print of exp
| Scope of stm
```

Substitution Model

- Expression takes one step `e → e’`
- Expression takes 0 or more steps `e →* e’`
- Expression can’t take a step (i.e. fully evaluated) `e -/->`
- Evaluate expression to value `e ⇒ v`
- Binary operations `bop` evaluate both sides in some order until done

Problem: evaluates everything, even if we don’t need it!

Environment Model

- More efficient than substitution - it is lazy
- We now keep track of the variables bound in the environment (machine config):

```
<env, e> → e’ or <env, e> ⇒ v
```

- Evaluate the environment following scope rules
  - dynamic: functions evaluated in dynamic environment when function is applied
  - lexical: functions evaluated in environment where they are defined

Example:

```
# let x = 1;;
# let f = fun y -> x;;
# let x = 2;;
# f 0;;
```

Dynamic result: 2
Lexical result: 1

Type Inference

Idea:

1. Infer types in the order they are written
2. Use previous types to infer later ones
3. Use context (constraints) to help infer types

Solve constraints like a set of linear equations

```
let inc y = y + 1
```

Equivalently:

```
let inc = fun y -> ((+) 1) y
```

Expression	Type
fun y -> ((+) 1) y	R
y	U
((+) 1) y	T
(+)	int -> (int -> int)
1	int
y	V

Constraints:

1. U = V
2. R = U -> S
3. T = V -> S
4. int -> (int -> int) = int -> T

Simplify:
T = (int -> int) from (4)
(int -> int) = V -> S from (3)
V = S = int from (3)
U = V = S = int from (1)
Solved! y is an int