CSci 2041
Advanced Programming Principles
L21: State and Effects

Eric Van Wyk

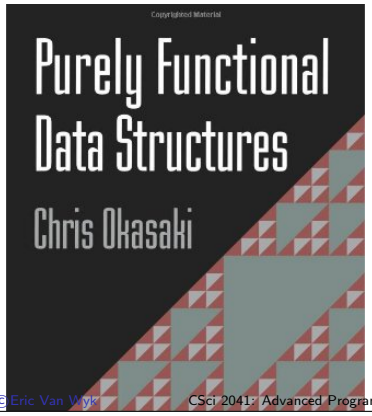
Fall 2014

Need for mutable state

- We saw need for mutable references in implementing laziness in a strict language.
- We may need it for some data structures, doubly linked lists or other circular structures.
  We can't create these *from the bottom up* like lists or trees.

Need for mutable state

But Chris Okasaki's book shows how we might need state less than we think.
A great book - check it out if you're interested in functional programming



Two points for discussion

- pointing vs. copying

  When can two references point to the same data in memory and when must we duplicate that memory?

- Denotational semantics
  We've seen how to evaluate expressions.
  What is the meaning of a statement?

Pointing vs Copying

- Consider this function

```
cons2 x y lst = x ::  y ::  lst
```

- This list

```
let l1 = all_ints_up_to 1000000
```

- And this list

```
let l2 = cons2 1 2 l1
```

- How much memory is required to store both `l1` and `l2`?
- Is there some reason that the underlying machine representation of `l2` could have a pointer to `l1`?

  Or must be copy all of `l1` to create a duplicate that is used in `l2`?

- In a language in which the value of `l1` never changes, because it might be a pure functional language, then `l2` can have a pointer to `l1`.
- In a language in which some element of `l1` might be changed by an assignment statement, then we may want to make a copy of `l1`.
- In mainstream languages the issue is phrased as making a shallow copy or a deep copy.

- The key issue is whether or not a data structure is mutable.

  Will it be changed after it is created?

- There are many libraries for Java and C++ that work over non-mutable data.
  - They don't provide operations to change a value once it has been created.
  - Thus the library implements only shallow copies and saves memory.

Denotational Semantics

- We've seen how to evaluate expressions to compute a value.
- How do we execute statements?

Meaning of expressions

- ```
  type value = IntVal of int | BoolVal of bool
  type env = (string * value) list
  eval:  expr -> env -> value
  ```
- Consider some expression
  `let e1 = Add ( Mul (Var "x", ...), ... )`
- What is its meaning?

  Does `eval` define its meaning?
- What is the type of `eval e1`?
- It is `env -> value`.
- So we can think of the meaning of an expression as a function from an environment or state to a value.
  ```

States and environments

- states and environments are more or less the same thing
- they map names to values
- but we tend to use the term "environment" in evaluating expressions or pure functional languages

  and "state" when thinking of imperative programs with statements and side effects

在functional programming較常稱做environment

Meaning of statements

- So what about statements?
- What is a statement? What is the "type" of its meaning?
- What is the meaning of `x = y + 3;` ?
- We can think of statements as state transformers.
- Their meaning has the type `state -> state`.
- So let's define the type `stmt` and the function `exec: stmt -> state -> state`.
- Find this code in `code-examples/interpreter.ml`