# intro to ocaml

## thunks, streams, lazy evaluation and monads

colin shaw
06.09.2016

1. **introduction**
    a. welcome to computer science club
    b. thank you to revunit for sponsoring
    c. what are we talking about
        i. introduction to ocaml
        ii. thunks
        iii. streams (type)
        iv. evaluation strategies
            1. strict / eager
            2. non-strict / lazy
        v. monads
        vi. being precise
            1. nil vs. null
            2. applicative vs. functional
            3. assignment vs. name binding

2. **why applicative / functional programming**
    a. new ways to think about computation
    b. new ways to think about problems
    c. provable performance

3. **tools**
    a. vim
    b. tmux
    c. pathogen
    d. vim-slime

4. **Introduction to ocaml**
    a. assignment vs. name binding
    b. scope
    c. hindley-milner can't do it all
        i. provides automatic type checking
        ii. provides parametric polymorphism
        iii. looses function overloading
    d. first class and higher order functions
    e. currying
        i. type definitions (show how relate to currying)
        ii. pass by value vs. pass by reference
    f. pattern matching

5. **vanilla lists**
    a. all lisp-like languages have the primitive list functions
    b. lists are a terminating type (e.g. empty aspect of disjunctive type)
    c. pros and cons
        i. pros
            1. familiar
            2. typical matching of disjunctive types for termination
        ii. cons
            1. clunky, inelegant feel
            2. does not represent non-terminating sequences well
    d. introduction of unit ()

6. **thunk streams**
    a. similar to lists but are non-terminating
    b. computational suspension (thunks)
    c. ability to compute infinite sequences one element at a time
        i. really important -- recursive definitions often are non-terminating
    d. functions on streams are similar to functions on lists but with thunks
    e. pros and cons
        i. pros
            1. simpler construction since non-terminating
            2. simpler reasoning of domain problem
        ii. cons
            1. more complex reasoning of function declaration
            2. slow with much redundant computation since non-memoized

7. **lazy streams**

a. similar to lists and thunk streams but more modular<br>
b. memoized computation can be much more performant<br>
c. potentially high memory use / leaks

8. **monads**

a. what are they?<br>
i. category mappings (e.g. functors)<br>
ii. monoids have specific algebraic properties<br>
1. on a set $S$<br>
2. associative binary operation $S \times S \to S$<br>
3. identity element $1 \to S$<br>
iii. monads in theory<br>
1. mapping category to itself ($T: X \to X$, e.g. endofunctor)<br>
2. two natural transformations<br>
a. binary composition ($T \times T \to T$, the join operator)<br>
b. identity ($I \to T$, the return operator)<br>
3. effectively modularization<br>
iv. monads in practice<br>
a. fmap / join (more closely parallels theory)<br>
i. fmap: $(t \to u) \to m\ t \to m\ u$<br>
ii. join: $m\ (m\ t) \to m\ t$<br>
b. bind / return (more generally useful)<br>
i. bind: $m\ t \to (t \to m\ u) \to m\ u$<br>
ii. return: $t \to m\ t$<br>
b. examples<br>
i. optionalized division<br>
1. compare structure of module with earlier demonstration<br>
2. *divopt* is similar to the more common *lift* taking two arguments<br>
ii. Identity<br>
1. avails easy morphisms on the monadic type<br>
iii. list<br>
1. typical list monad implementation<br>
2. look at internals of execution<br>
iv. printf<br>
1. classic io monad<br>
2. abstracts side effects within monadic container<br>
c. applications<br>
i. io<br>
ii. repetitive application<br>
iii. parsing