S0.3: Final Exam Review

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Final Exam Review

- Material covered, and not covered
- Logistics
- Survey of topics

Material covered: I

- S1.1 Introduction to OCaml.
- S1.2 Higher Order Functions.
- S1.3 Expression, Values, and Evaluation.
- S1.4 Inductive Types and Values.
- S2 Reasoning About Correctness.
- S3 Programs as data.
- S4 Expression evaluation
  - S4.1 Lazy evaluation.
  - S4.2 Improving performance.
  - S4.3 Parallel evaluation.
- S5 Imperative programming
- S6 Modules
- S7 Search

Material covered: II

- S8 Purely Functional Data Structures.
- Hickey text, chapters 1-7.
- Okasaki text, chapter 1-3, 5 (sections 1-3)

Material not covered:

- S9: Application of Programming Principles
- That bit about my research obviously...

Logistics

- for Sec 01 that meets at 1:25pm,
  Tuesday, May 8 at 10:30am, regular classroom.
- for Sec 10 that meets at 3:35pm,
  Wednesday, May 9 at 10:30am, regular classroom.
- The full 2 hours will be used for the exam.
- Closed-book and closed-notes.
- Format of exam questions will be similar to that of in-class exercises and homework questions.
- You are allowed one **double-sided** 8.5 inch by 11 inch page of **hand-written** notes.
  You will turn in your cheat-sheet. Put your name on it.
- Bring photo ID.

S1.1: Introduction of OCaml

- Understand the basic notion of functional programming.
- Operations over primitive types and over lists.
- let-expressions
- These include (recursive) computations over numbers, strings, lists, and tuples.
- Pattern matching and understanding how to write non-trivial patterns does as well.

S1.2: Higher order functions

- Of special importance is the notion of functions as *first-class citizen* of the language an all that that entails.
- Lambda-expressions and curried functions play an important role.
- Understand and be able to construct the OCaml types for functional values.

S1.3: Expressions, values, evaluation

- understand the process of expression evaluation
- difference between expressions and values

S1.4: Inductive types and values

- OCaml's `type` keyword for type abbreviations, enumerated types, and generalization to inductive types.
- Definition of such types: lists, trees, options, etc.
- Use of pattern matching over these types

S2: Reasoning About Correctness

- Understand the principle of induction for natural numbers, our `nat` type, and for lists.
  But also understand how to derive a principle of induction for an inductive type definition.
- Be able to prove properties of simple functions over these kinds of types (using induction).
- The quiz may ask for specific parts of your proof as answers to different questions, be aware of this.
- Paying attention to how proofs are done in the sample-proofs document will save you time and effort on the quiz, and the homework.

S3: Programs as Data

- How to represent expressions, programs using inductive types.
- How to represent values, especially closures for functional-type expressions.
- Evaluation of expressions as in Hwk 04.

S3: Programs as Data

Sample problems:

- Define values of given expressions
  Value of `f` in
  `let x = 2 in let f y = x + y in f 5`
- Extend evaluation to handles lists, tuples, additional operators
- Write functions to process expressions to compute it type, number of division operations, evaluation that handles division by zero explicitly ...

S4.1: Expression evaluation: Lazy evaluation

- understand the evaluation techniques of call-by-name, call-by-value, and lazy evaluation
- know how they differ and how they are similar
- be able to evaluate expressions by hand using the techniques discussed in class for each of these evaluation strategies
- be able to show how laziness allows for a freer style of writing programs in some cases
- be able to read and write OCaml code that simulates lazy evaluation, specially using the `stream` type.

S4.2: Expression evaluation: Improving performance

- tail recursion, tail position
- identifying a function as tail recursive, regardless of any optimizations a compiler might do
- using accumulating parameters to increase performance
- using continuation passing style to increase performance
- be able to write these kinds of functions

S4.3: Expression evaluation: Parallel evaluation

Understand concepts

- difference between parallelism and concurrency
- determinism, non-determinism
- concepts of “work” and “depth” in parallel programs

S5: Programming with Effects

- issues of pointing vs copying
- ideas from denotational semantics
  - meaning of expression as `env -> value` function
    `eval : expr -> (env -> value)`
  - meaning of statement as `state -> state` function
    `exec : stmt -> (state -> state)`

S6: Modularity

- What is “programming in the large”
- Why do we care about modularity
- How are modules, signatures/interfaces, and functors used in OCaml and for what purposes
  - in abstract data types
  - for separate compilation
  - for code organization
- Concepts of transparent, translucent, and opaque modules
- purpose of the `with type` clause in manipulating signatures

S7: Search

- how search space can be seen as a tree or a graph, reasons for different views
- understand how OCaml `option`s can be used to control search
- understand how exceptions can be used to control search
- understand how continuations can be used to control search
- understand concepts and programming fragments

S8: Purely Functional Data Structures

- Understand invariants of binomial heaps and red black trees. Be able to understand code using these data structures.
- Understand approaches to computing amortized costs (Banker's and Physicist's methods) and be able to apply them.

Conceptual concerns

- role played by type systems, both static and dynamic
- the guarantees that these provide
- reasons for having static or dynamic type systems
- issues of operator precedence and associativity, in expressions that compute values and in type expressions
- referential transparency and its implications

Programming in OCaml

You may be asked to

- write functions over lists,
- write higher order functions,
- use functions such as `map`, `filter`, `fold_left`, and `fold_right` to solve problems,
- read OCaml code and understand the computation it may carry out,
- be able to infer the type of functions and other values,
- determine if OCaml declarations are type correct.