CSci 2041
Advanced Programming Principles
L16: Exam 2 Review

Eric Van Wyk

Fall 2014

L 8.0: Exam 2 Review

- Material covered.
- Logistics
- Survey of topics

Material covered - Old

- L2: Introduction to OCaml
- L3: Types and Unions
- L4: Programs as Data
- Nothing explicitly from Real World OCaml or Jason Hickey's Introduction to OCaml books.
  - But, you should understand the portions of the OCaml language needed to complete Homework 2, 3, and 4.

This material was also covered on exam 1.

Material covered - New

(Details of each section follow later.)

- L5, L6: Reasoning about Program Correctness
- L7: Reasoning about Performance
- L9, L10: Higher Order Functions
- L12: Programming Techniques
- L13, 14: Expression Evaluation

Logistics

- Wednesday, November 19.
- The full 50 minutes used for the exam.
- In class exam.
- Closed-book and closed-notes, except for a **double-sided** 8.5" $\times$ 11" page of **hand-written** notes.
- Format of exam questions will be similar to that of homework and Exam 1 questions. Some short answer questions will ask for written explanations or descriptions - these are not expected to be longer than 4 or 5 sentences (unless indicated otherwise).

L2 Intro to OCaml material: I

- Write simple OCaml functions over primitive data such as integers and strings as well as lists and tuples
  For example,
  - a function to compute the area of a circle given its radius as a value of type `float`,
  - to compute the factorial of an integer,
  - to compute the product of all numbers in a list.

L2 Intro to OCaml material: II

- Be able to determine the types (or type errors) of functions and expressions using primitive data as well as lists and tuples.
  For example
  - `3 ::  [4;5]`
  - `let x = 3 in (x, "Hello")`
  - `[4;5] @ [3.14; 6]`
  - `let add x y = x + y`
  - `let inc4 = fun x => x + 4 in inc4 (inc4 3)`

L2 Intro to OCaml material: III

- Understand the "curried" nature of functions in OCaml: understand the types, for example `int -> int -> int`, and the OCaml syntax for function application.
- Understand the difference between `let` and `let rec` in OCaml. Specifically how name binding is done in each one.
- Pattern matching:
  - Be able to write `match` expressions with appropriate patterns in the clauses.
    For example,
    - write a function to extract the first element from each pair in a list of pairs. This function must have the type `('a * 'b) list -> 'a list`
  - Be able to determine if a set of patterns is "exhaustive" or "non-exhaustive"

L2 Intro to OCaml material: IV

Go back to this

- Understand the different categorization of program errors:
  - syntax errors, detected statically
  - type errors, detected statically or dynamically, depending on the language
  - other dynamic errors reported at run time
  - "unsafe" operations that go undetected.
- Difference between static and dynamic typing. Advantages of each approach.
- Definitions of "strong" type system, "safe" language.

Expect short answer style questions for this kind of material.

L3 Types and Unions material: I

- How new types (and values) are defined in OCaml with `type` declarations.
  - Simple "enumerated" style of types and values
  - Types like `option` or our `value` (for wrapping ints and floats) types.
  - How lists and trees are implemented as recursive disjoint unions.

L3 Types and Unions material: II

For example,

- Provide three sample OCaml values that have the type `(int * string) list`
- Design a type to represent one of three kinds of animals.

  1. birds, their species name along with their wingspan in some numeric representation whether they fly (*e.g.* eagles) or not (*e.g.* emus).
  2. mammals, their species name along with the number of legs used for locomotion (*e.g.* humans use 2, dogs use 4) and their average weight in kilograms (as a floating point number).
  3. amphibians, their species name and whether they are a frog, toad, snake, lizard, or some other type of amphibian.

L3 Types and Unions material: III

- Define, identify, and write types, type constructors, values, and value constructors in type declarations.
  For example, in

  ```
  type 'a myList = Nil
                 | Cons of ('a * 'a myList)
  ```

- Be able to read and write pattern matching expressions and functions containing them over (recursive) disjoint union typed values.
  For example,
  - write a function to find the maximum element in an unordered `int btree` (see `btree.ml` in code-examples).
  - write a function determine if an `expr` can be simplified because it is the sum of 0 and some other expression.

L3 Types and Unions material: IV

- Understand the distinction between sum and product types.
  Understand why disjoint unions are sometimes referred to as "sum of product" types.
- Relation of disjoint union types to classes in OOP.

L4: Programs as Data material I

- Understand how to represent expressions using disjoint unions.
- Be able to read and write functions over this type of data.
- Understand how expressions represented this way may not be "semantically correct" in that they may have undeclared identifiers or may have expressions with type errors.
- Be able to read and write functions that check if an expression is "semantically correct"

L4: Programs as Data material II

- Determine if an OCaml declaration or expression contains type errors.
  - If it does, what is the nature of the error?
  - If it does not, what it the type?

  Similar to previous examples for L2, but over trees and such types from L3 and expressions from L4.

L5,6: Reasoning about Program Correctness

- Understand how a principle of induction can be generated from a disjoint union type.
  We did this for the list type and for a more general case.
- Be able to write short proofs for properties about functions.
- Be able to reason about imperative programs and loop invariants. This may not require a "proof", but a precise explanation why a property is (or is not) a loop invariant. Also be able to explain why a post condition would hold after a piece of imperative code has completed, if a loop invariant is provided.
- Be able to design a program from a provided loop invariant or desired property of a function.

L7: Reasoning about Program Performance

- Be able to solve simple recurrence relations like we did in class.
- Be able to determine the complexity of a function - either by solving a recurrence relation or by informal reasoning.

L9,10: Higher Order Functions I

Understand and be able to define and use

- helper functions passed into other functions (*e.g.* equality checking or comparison functions)
- what types OCaml infers for these kinds of helper functions and the functions they are passed into
- how to specify helper functions by using
  - lambda expressions
  - converting some operators into functions, *e.g.* `(+)` but not `::` and why.
  - using curried functions
- write functions that take functions as arguments, *e.g.* `take_while`

L9,10: Higher Order Functions II

- understand functions like map, filter, and folds that encapsulate a computational pattern
- be able to write code fragments using these functions
- be able to infer types or detect type errors in their use

L12: Programming Techniques

- understand proper ways to structures programs that are a series of transformations and analyses on data
- know when, and when not, to use exceptions

L13,14 Expression Evaluation I

- 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.
- understand how coroutines can be simulates using lazy evaluation