CS3110 Spring 2017 Lecture 25: Course Review and Final Exam Coverage

Robert Constable

Note: Final Exam Sunday May 21, 2017 2-4 pm Kimball B 11

1 Brief Summary of Lectures

Here is a list of the lecture topics with comments.

Lecture 1: Introduction to Functional Programming with Types

This lecture is 25 pages long and it establishes the context and themes for this version of the course. It would be a good idea to read it again since there will be at least one final exam question about the themes and context. We also introduced terminology from the literature on programming languages such as a discussion of *eager* versus *lazy* evaluation.

Lecture 2: OCaml Syntax and Semantics

This lecture is 10 pages long and provides the framework for *operational semantics*. It defines the concepts of *canonical* and *non-canonical* expressions which are basic, and it provides examples of the evaluation rules. These ideas are common to all versions of the course. Dr. Clarkson covers this material in more detail in lectures 2 through 5 in the fall version of the course.

Lecture 3: More reduction rules, the notion of Currying and Uncurrying, and typing rules for some of the constructs. A key signature idea of the ML family of languages is introduced, the *polymorphic types* which in this version of the course are written with both the standard OCaml syntax ‘a, ‘b, ‘c, ... and with Greek letters as in the original articles on ML, e.g. α, β, γ, ... It would be good to know how to curry and uncurry functions.

Lecture 4: More types, especially *variant types*, and a discussion of the Church/Turing thesis and other formalisms that are universal such as *com-*

binators, e.g. **S**,**K**,**I**,**X**, and **Y**. We will not cover these combinators on the final exam. It is a remarkable curiosity that all computing can be done with only the **X** combinator. The combinators were not an important topic in the course, they were for *cultural enrichment*.

Lecture 5: More types, including more polymorphic types and the key idea of using *types as specifications* of programming tasks. After looking at lists in detail, we pose the question "What is a type?"

Lecture 6: Logical specifications and their use in deriving code. Comparing types to sets. The important idea of *dependent types* is mentioned in this lecture to express the specification that an integer input is greater than 0. The type $x : int\star \; where \; x > 0$ is mentioned. The intuitive idea is clear, and the lecture mentions that established proof assistants such as Agda, Coq, and Nuprl use dependent types. It is claimed that these types will gradually make their way into mainstream programming languages. At Microsoft the are used in the $F^\star$ language. It will be important to be able to use normal logic to specify programming tasks, e.g. to use the universal and existential quantifiers, as in $\forall\ x\ : T.\exists\ y : T.P(x, y)$.

Lecture 7: We discussed in more detail how to *specify programming tasks* using *types*, illustrated first with polymorphic types. A key point about polymorphic types is discussed, that when the type corresponds to an expression in the propositional calculus that is not a tautology, then the task described is not *programmable. An example of this phenomenon will almost surely appear on the final exam.*

Lecture 8: We studied specifications in more detail. The task of finding the integer square root of a positive integer is discussed in detail. It is shown how to create a program from a constructive proof that there is an integer square root. By popular demand, we introduced a dependent type from Coq, namely the *universal quantifier*. It is the only dependent type needed in Coq. It has the form

```
forall (x:T),P(x).
```

Other useful dependent type constructors defined in Coq were also examined, namely

```
And(P,Q:Type) = and_(p:P)(q:Q),
```

and

```
Or(P Q: Type) = or_left(p:P) | or_right(q:Q).
```

The fold operation is discussed in more detail.

Lecture 9: Inductive proofs of specifications. The lecture provided details of Prof. Kreitz's integer square root algorithm. The lecture repeated the Coq definitions of dependent types and used logical notation in the specification.

Lecture 10: Questions about Lecture 9 are answered, then a module for rational numbers is given and the algebraic properties of the rational numbers are specified in the module. An algorithm for the *greatest common divisor* (gcd) is given and discussed.

Lecture 11: The *computable real numbers* are defined as in the Bishop and Bridges book *Constructive Analysis* [1]. *Chapter 2 of this book is a resource for the course.* The importance of having a precise definition of the unbounded precision computable real numbers is discussed in terms of knowing the *ground truth* about real number computations in science and engineering. This is a salient point of the course. L.E.J. Brouwer's insights about the continuum of real numbers are discussed. It will be important to know the definition of a real number, to know how to add and multiply them and to prove that a number is positive and non-negative.

Lecture 12: Properties of the constructive real numbers and a prelim review.

Prelim in class Tue March 14.

Lecture 13: Discussion of the prelim and more theorems about the constructive reals. Mention that we will start *synthetic computational* geometry.

Lecture 14: Computational geometry and convex hull task. Cantor's theorem for the real numbers $\mathbb{R}$, *synthetic* convex hull algorithm.

Lecture 15: A Constructive synthetic Convex Hull algorithm.

Lecture 16: Implementing a Convex Hull algorithm and comments on the *Atlantic* article on the nature and origins of computer science.

Lecture 17: Constructive Euclidean Geometry, Prop 9 by Ariel Kellison and Mark Bickford.

Lecture 18: Binary Search Trees, basic definitions, discuss delete operations in detail. It is *quite likely* that the exam will include a question of how to delete from a BST and why the delete operation is correct.

Lecture 19: Konig's Lemma and Brouwer's Fan Theorem. Statement of the theorem and informal constructive proof of the Fan Theorem. Note that Konig's Lemma is not constructively true.

Lecture 20: Streams, choice sequences, and continuity. Informal proof of the Fan Theorem, using Konig's Lemma to prove that code Fan(P,n,s) for Brouwer's Fan Theorem converges, an example of Russian constructive mathematics. It will be important to know the statement of the Fan Theorem and its relationship to Konig's Lemma.

Lecture 21: Distributed computing with functional processes. Discuss the Leader Election in a Ring protocol. Discuss the general consensus problem and give imperative pseudo code.

Lecture 22: The General Process Model for distributed asynchronous computing. This is a simple topic and probably good for a simple question on distributed protocols and how to reason about them.

Lecture 23: Fixed point operators, fix and efix. Defining recursive functions in OCaml using fixed point operators, *fix* for lazy evaluation and *efix* for *eager-fixed point*. Students should know the recursive definitions of these two operators and to know why they are important. We used efix extensively and there is plenty of code using it in the lectures. The fix operator is much simpler, namely

$$fix\ f\ =\ f(fix\ f).$$

The polymorphic type of fix is $(\alpha\ \rightarrow\ \alpha)\ \rightarrow\ \alpha$.

Lecture 24: Computability theory in OCaml, stressing that all OCaml types are *partial types*, that is expressions that the type checker says are of type T might diverge. Know the very simple proof that the halting problem is not solvable by an OCaml program even for the trivial type unit and even for lazy evaluation. It is an interesting fact that all types in Coq are total types not partial types.

Lecture 25: This lecture – self reference but not non-terminating.

2 More on Unsolvable Halting Problems in Type Theory

Every non-empty OCaml type T is *partial* in that it includes terms t of type T that fail to terminate when they are evaluated. For example consider the *unit* type. Its only canonical value is (). We can disguise this element in a recursive function from *int* into *unit* as follows. We let *perp* be the diverging term $\perp$. [1]

```
let rec loop n = if n = 0 then ( ) else loop n-1.

loop(0) = ( ), so loop(0) belongs to unit.

loop(1) = ( ), so loop(1) belongs to unit.

But loop(-1) diverges, yet it also belongs to the unit type.

We say that this diverging element is perp.
```

This behavior demonstrates that all OCaml types are *partial types* because they include as members terms that diverge when evaluated.

We now prove that there is no OCaml function to decide halting on the *unit* type.

Suppose there is an OCaml function $h : unit \ \rightarrow \ bool$ such that $h(e) \ = \ true$ if and only if e converges to the element of unit. For this result we will assume that we use *lazy evaluation* in applying functions. This allows for a very simple proof, and we could apply this to OCaml by exploring its lazy evaluation option. It is less clear how to get a very simple halting problem result like this for eager evaluation.

First define the function $funx \rightarrow \ if \ h(x) \ then \ \perp \ else \ ()$. Next we apply the lazy fix operator to this function and call the result d.

$$let\ d \ = \ fix(fun\ x \ \rightarrow \ if \ h(x) \ then \ \perp \ else \ ()).$$

By the typing rule for *fix* applied to $(unit - > \ unit) \rightarrow (unit - > \ unit)$, we know that d belongs to *unit*.

What is the value of $h(d)$? If this expression converges to a truth value as we assume it does, then we can derive a contradiction as follows. If $h(d)$

[1]This is also affectionately called "Scott's bottom" because Dana Scott used it to denote the diverging element in his *domain theory* [3].

returns *true* then the result is the diverging element, $\perp$, contrary to the definition of h. If $h(d)$ returns $false$, then the result is () the element of *unit*, and that contradicts the definition of h as well. So there can be no such OCaml computable function to detect halting. Further undecidability results about partial types are studied in the article [2].

It might seem that this simple result is "cheating" because the function h does not have access to the syntax of the input expression even though for lazy evaluation the input is substituted without evaluating it. But the halting detector h for the elements of unit must analyze all expressions exp that the OCaml type checker says are of the type unit. This is an unbounded collection of OCaml expressions. Some of them will diverge, and h must "figure this out." So the halting task involves analyzing an unbounded collection of expressions to attempt to determine whether or not they halt. This suggests that the function h will be quite complex – indeed impossibly so as our simple proof that there is no such h shows.

References

[1] E. Bishop and D. Bridges. *Constructive Analysis.* Springer, New York, 1985.

[2] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Journal of Theoretical Computer Science*, 121(1&2):89–112, December 1993.

[3] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.