CS3110 Spring 2017 Lecture 8
Specifications continued

Robert Constable

1 Lecture Plan

1. Repeating schedule of remaining five problem sets and prelim.
2. Examples of logical specifications and the evolution of specification methods.
3. OCaml specifications using types, limitation and comparisons.

2 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS2	Out on Feb 23	March 2
Prelim	Tue. March 14, in class	
PS3	Out on Thur. March 2	March 16
PS4	Out on March 16	March 30
PS5	Out on April 10	April 24
PS6	Out on April 24	May 8 (day of last lecture)

3 Review of role of specifications in programming

Here are some highlights from the introduction to this course that bear on the topic of specifications. We start with the very first sentence of the first lecture, bolding those statements that bear on this lecture in particular.

1.This course is about the *functional programming paradigm and the tasks for which it is appropriate.* To experience the ideas it is critical to know at least one mainstream functional programming language and use it to solve

interesting problems. That language should have a rich type system. **Types provide high value in precisely specifying programming tasks and in reasoning about whether a program accomplishes them "according to the spec."**

2.It is characteristic of Cornell CS that **the enduring intellectual themes are part of our upper level CS curriculum.** In PL we are easily able to see some of the deepest ideas in action. For example, you already know that in some sense, OCaml is a *universal programming language*. It is surprising how small the universal core of OCaml is.

3. We are studying the functional programming paradigm, algorithm design, and precise problem specification **using data structures for which functional languages work well**. Typical examples of such data are lists, trees, arbitrarily large natural numbers (Big nums), real numbers, and of course functions. We will also consider data types such as co-lists and co-trees although OCaml does not have these important types.

4.The OCaml programming language is known for its relatively clean and simple *type system* including *polymorphic types*. Thus types will be a key focus of the course. **OCaml types serve as an introduction to the wider role of *type theory* in computer science, not only in programming but in the subject called *formal methods* and in computer security.**

5.Highly precise *logical specifications* are widely used in mathematics, science, engineering, and commerce. **We are already seeing something quite interesting about OCaml, namely that its type system is rich enough to capture specifications given by propositional logic – without ever mentioning logic explicitly.** We will take advantage of this important feature of OCaml types to avoid teaching logic as a separate topic in the course. This small topic is significant as well as fun.

6.Another foundational language for mathematics is **type theory**, advocated by Sir Bertrand Russell and Albert North Whitehead [10, 7] in a three volume treatise called *Principia Mathematica*.[1] **The British computer scientist Sir Tony Hoare showed how to adapt this more expressive language to define the types emerging from the use of programming languages [5].**

It is interesting that I already heard point 6 come back to me as a question after Lecture 7. This is good, and it shows that some of the earlier notes

[1]See the Logicomix [4] story about this book, *An Epic Search for Truth*.

are relevant to your new questions. It will be important to review the lecture notes for the exams because they stress points that I find important.

3.1 Comments on polymorphic type specifications

In Lecture 7 we observed this important fact:

This is a general fact, that if a Boolean assignment shows that a logical specification is not satisfiable (true in some assignment of truth values to types), then it is not programmable.

However, it is not the case that all Boolean satisfiable specification are programmable. A simple example is $L\alpha|R(\alpha \rightarrow void)$ where *void* is the empty type. Another example of this phenomenon shows up in trying to program *Pierce's Law* using polymorphic OCaml types in its specification, e.g. $((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha$. We will look deeper into this situation once we have discussed ways for thinking about the *logical meaning of types*.

This example raises the question about whether there is a precise characterization of which logical specifications can be seen as program specifications. We will be studying this question as we go deeper into type theory. In the meanwhile, we will look at another simple programming problem and how to specify it precisely using logic.

3.2 Specifying integer square roots

Consider the simple but informative problem of finding the *integer square root* of a natural number. This example conveys the idea if we imagine that *sqrt* is the function we are typing to program and it includes these instances: $sqrt(0) = 0, sqrt(1) = 1, sqrt(2) = 1, sqrt(3) = 1, sqrt(4) = 2, sqrt(5) = 2, sqrt(6) = 2, ..., sqrt(9) = 3, sqrt(10) = 3$. From this we see that we want the general condition

$$sqrt(n) \times sqrt(n) \leq n.$$

We also want to know that having $sqrt(17) = 2$ is not a good answer. We'd expect $sqrt(17) = 4$. In general we want that $sqrt(m) = n$ implies that $n + 1$ is too large, $((n + 1) \times (n + 1)) > m$.

Here is OCaml code to solve the task specified above, followed by examples of its execution.

```
# let rec sqrt n = if n = 0 then 0
          else let r = sqrt (n-1) in
          if ((r+1) * (r+1)) <= n then (r+1) else r ;;
val sqrt : int -> int = <fun>
# sqrt 17 ;;
- : int = 4
# sqrt 256 ;;
- : int = 16
# sqrt 10000 ;;
- : int = 100
# sqrt 100000 ;;
- : int = 316
# sqrt 27878400 ;;
Stack overflow during evaluation (looping recursion?).
>answer should be 5280 since
# 5280 * 5280 ;;
- : int = 27878400
# sqrt 278784 ;;
Stack overflow during evaluation (looping recursion?).
# sqrt 27878 ;;
- : int = 166
```

3.3 Logical specification of integer square root problem and proof of solvability

```
∀n ℕ   ∃r ℕ   r² ≤ n < (r+1)²
BY  allR
  n ℕ
  ⊢ ∃r ℕ   r² ≤ n < (r+1)²
  BY  NatInd 1
      basecase
      ⊢ ∃r ℕ   r² ≤ 0 < (r+1)²
  √   BY  existsR 0 THEN Auto
      upcase
      i ℕ⁺,  r ℕ,  r² ≤ i-1 < (r+1)²
      ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      BY  Decide (r+1)² ≤ i THEN Auto
          Case 1
          i ℕ⁺,  r ℕ,  r² ≤ i-1 < (r+1)²,  (r+1)² ≤ i
          ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      √   BY  existsR r+1 THEN Auto
          Case 2
          i ℕ⁺,  r ℕ,  r² ≤ i-1 < (r+1)²,  ¬((r+1)² ≤ i)
          ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      √   BY  existsR r THEN Auto
```

Figure 1: Proof of the Specification Theorem using Standard Induction.

What is remarkable of this example is that the program can be found in the proof and extracted from it as code in various programming languages. This has been a topic of research at Cornell since the 1980s and it led to the Nuprl proof system in 1984. The idea is often called *proofs as programs* [1] and it has been adopted in Agda, and Coq and other proof assistants "in the works."

It would be an excellent feature for CCaml to have a rich enough type system to express all of the programming tasks that can be described in standard logic, such as First-Order Logic [8]. These topics are discussed well in the free on-line text book cited previously, Simon Thompson's *Type Theory and Functional Programming* [9]. It is an old textbook that is still relevant in many ways.

The Coq proof assistant uses a rich type theory. We could call it the *Coq*

type theory. It is described in the on-line textbook *Software Foundations* [6]. Since I mention Coq frequently and even discuss its programming language, CoqPL, it is worthwhile to tell you a bit more about that type theory. It illustrates the future of types in programming languages. All we need to cover to make this a very useful part of the course are the types needed to define First-Order logic. These are very easy to understand given what we have already said about OCaml types and logic. Here are some examples directly from a short introduction to Coq.

3.4 The Coq type system for logic.

There are full textbooks on Coq [2, 3, 6] and a web cite devoted to it. We will look at only a small subset of the logical rules of Coq. There is a nice short unpublished paper by Bart Jacobs on the web, entitled *The Essence of Coq as a Formal System*.

In Coq *dependent function type* is primitive. Its elements are functions, and the typing has this property. **It is the only primitive logical type Coq needs.**

```
forall (x:T), P(x) defines functions taking inputs t from
type T into vales p(x) in type P(x).
T -> P defines functions taking inputs of type T into outputs of type P.
```

The non-dependent case is the same as the OCaml function type $ty1 \rightarrow ty2$.

The logical operators are defined *inductively*. OCaml would call such types recursive if it had them. The logical operators are defined on these inductive types rather than using polymorphic types. Here is the definition of And and Or. We can see the similarity to the OCaml "or" by the use of the vertical bar to separate the cases.

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

The definition of And tells us that it is an ordered pair, similar to the OCaml type (ty1 * ty2). The elements of the type are members p and q of the types P and Q respectively.

The definition of Or is essentially an OCaml *variant* with the tags *left* and *right*. The elements of the types are denoted p and q respectively.

The next type is an example of a defined *dependent type*. This kind of type is what gives Coq the power to express logic.

```
Inductive Exists(T: Type)(P: T -> Type) := exists_(t: T)(p: P t).
```

The definition tells us that T is any Type, and P is an element of the space of functions from elements t of type T into into ordered pairs (t:T) (p: P t). This captures the essence of a *dependent type* and defines the existential quantifier, as used in the theorem about square roots. Note, P is the function from type T into Type, and t is an element of the type T. So (p:P t) says that p is an element of the type that depends on the value of the function P at element t in type T. This type construct is not available in OCaml, but Coq shows that we can compute with it and reason about it using more sophisticated methods than are available in OCaml.

```
Inductive Equals(A: Type)(a: A): A -> Type := equals_: Equals A a a.
Inductive False :=.
Inductive True := true_.
```

These definitions give of the empty type False, which can be defined in OCaml as well, and the type True which has just one element in it, true. Both of these types are available in OCaml. We used them to define the negation operator, say $\alpha \rightarrow \mathit{void}$ for the polymorphic negation.

```
Definition Not(P: Type) := P -> False.

Definition modus_ponens: forall (P Q: Type), P -> (P -> Q) -> Q :=
fun (P Q: Type) => fun (p: P) => fun (pq: P -> Q) => pq p.



Definition distr: forall (P Q R: Type), And P (Or Q R) ->
Or (And P Q) (And P R) :=
fun (P Q R: Type) =>
fun (pqr: And P (Or Q R)) =>
match pqr with
```

```
 and_ p qr =>
 match qr with
 or_left q =>
 or_left (And P Q) (And P R) (and_ P Q p q)
 | or_right r =>
 or_right (And P Q) (And P R) (and_ P R p r)
 end
end.

=======
Inductive Le: nat -> nat -> Type :=
le_refl: forall (n: nat), Le n n
| le_succ: forall (m n: nat), Le m n -> Le m (succ n).
The free variables and the premises of a rule correspond
to the parameters.

Fixpoint Le_succ_succ(m n: nat)(H: Le m n) {struct H}: Le (succ m) (succ n) :=
match H in Le M N return Le (succ M) (succ N) with
 le_refl n0 =>
 (* Goal: Le (succ n0) (succ n0) *)
 le_refl (succ n0)
 | le_succ m0 n0 H0 =>
 (* Goal: Le (succ m0) (succ (succ n0)) *)
le_succ (succ m0) (succ n0) (Le_succ_succ m0 n0 H0)
end.
```

4 User defined list examples continued from Lecture 7

The following examples are included in the Lecture 7 notes as well, and we discussed the fold operations in that lecture. Here we discuss the map examples. These examples are from the 2008 version of this course.

```
Examples of operations on lists mentioned in Lecture 7.

There are two versions of the reduce operation, based on the nesting
of the applications of the function f in creating the resulting value.
```

In OCaml there are built-in reduce functions that operate on lists are called List.fold_right and List.fold_left.

These functions produce the following values:

```
fold_right f [a; b; c] r = f a (f b (f c r))
fold_left f r [a; b; c] = f (f (f r a) b) c
```

From the forms of the two results it can be seen why the functions are called fold_right which uses a right-parenthesization of the applications of f, and fold_left which uses a left-parenthesization of the applications of f. Note that the formal parameters of the two functions are in different orders, in fold_right the accumulator is to the right of the list and in fold_left the accumulator is to the left of the list.

Again using the list_ type we can define these two functions as follows:

```
let rec fold_right (f:'a -> 'b -> 'b) (lst: 'a list_) (r:'b): 'b =
    match lst with
      Nil_ -> r
    | Cons_(hd,tl) -> f hd (fold_right f tl r)
```

and

```
let rec fold_left (f: 'a -> 'b -> 'a) (r: 'a) (lst: 'b list_): 'a =
    match lst with
      Nil_ -> r
    | Cons_(hd,tl) -> fold_left f (f r hd) tl
```

Note the type signature of fold_right which is

```
('a -> 'b -> 'b) -> 'a list_ -> 'b -> 'b
```

The parameter f is a function from the element type of the input list 'a and the type of the accumulator 'b to the type of the accumulator. The type signature is analogous for fold_left,except the order of the parameters to both f and to fold_left itself are reversed compared with fold_right.

Given these definitions, operations such as summing all of the elements

of a list of integers can naturally be defined using either fold_right or fold_left.

```
fold_right (fun x y -> x+y) il 0
fold_left (fun x y -> x+y) 0 il
```

The power of fold

Folding is a very powerful operation. We can write many other list functions in terms of fold. In fact map, while it initially sounded quite different from fold can naturally be defined using fold_right, by accumulating a result that is a list. Continuing with our List_ type,

```
let mapp f l = (fold_right (fun x y -> Cons_((f x),y)) l Nil_)
```

The accumulator function simply applies f to each element and builds up the resulting list, starting from the empty list.

The entire map-reduce paradigm can thus actually be implemented using fold_left and fold_right. However, it is often conceptually useful to think of map as producing a list and of reduce as producing a value.

What about using fold_left instead to define map? In this case we get a function that not only does a map but also produces an output that is in reverse order of the input list. Note that fold_left takes its arguments in a different order than fold_right (the order of the list and accumulator are swapped), it also requires a function f that takes its arguments in the opposite order of the f used in fold_right.

```
let maprev f l = fold_left (fun x y -> Cons_((f y),x)) Nil_ l
```

This resulting function can also be quite useful, particularly as it is tail recursive.

Another useful variation on mapping is filtering, which selects a subset of a list according to some Boolean criterion,

```
let filter f l = (fold_right (fun x y -> if (f x) then
Cons_(x,y) else y) l Nil_)
```

```
The function f takes just one argument, the predicate for
determining membership in the resulting list. Now we can easily
filter a list of integers for the even ones:

    filter (fun x -> (x / 2)*2 = x) Cons_(1,Cons_(2,Cons_(3,Nil_)))

Note that if we define a function that filters for even elements of a list:

    let evens l = filter (fun x -> (x / 2)*2 = x) l;;

then type of the parameter and result are restricted to be int list_
rather than the more general 'a list of the underlying filter, because
the anonymous function takes an integer parameter and returns an integer
value.

Determining the length of a list is another operation that can
easily be defined in terms of folding.

    let length l = fold_left (fun x _ -> 1 + x) 0 l
```

References

[1] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning (JFR)*, 3(2):1–93, 2010.

[4] A Doxiadis and C. Papadimitriou. *Logicomix: An Epic Search for Truth*. Ikaros Publications, Greece, 2008.

[5] C. A. R. Hoare. Recursive data structures. *International Comput. Inform. Sci.*, 4(2):105–32, June 1975.

[6] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. Electronic, 2013.

[7] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[8] R. M. Smullyan. *First–Order Logic*. Springer-Verlag, New York, 1968.

[9] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[10] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.