CS3110 Spring 2017 Lecture 5
Still More OCaml Types

Robert Constable

1 Lecture Plan

1. Repeating schedule of problem sets (6 of them) and prelim.
2. Polymorphic types continued.
3. Recursive types and definition of OCaml Lists.

2 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS1	Out on Tue. Feb 7	Feb 14
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 Overview – Types in Mathematics and Programming

This lecture will continue our discussion of polymorphic types and their relationship to logic. We will note a feature of OCaml that is a bit odd. It brings up an interesting issue about the role of types as specifications for programming tasks, which was the theme of the last lecture.

We will discuss the difference between types and sets. That topic is covered in the notes from Lecture 4, but we did not have time to cover that topic last time.

I also decided that it would be good to continue introducing you to various key players in the development of modern programming languages and in shaping the foundations of computer science. We have already mentioned Alonzo Church, Alan Turning, Haskell Curry, John McCarthy, and Robin Milner. Some of these pioneers, such as Church, McCarthy, and Milner are people I knew well personally. So I can give you a sense of who they were as well as what they contributed of great lasting value in computer science.

We will also examine the type of lists in more detail. The major source for that material will be the url's for previous versions of the course that provide excellent accounts of lists. These are primary resources for this lecture. I repeat those links just below.

`http://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec03-scope/scope.htm`

`http://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec04.html`

In due course we will talk more generally about what the nature of types and the development of a rigorous mathematical type theory. We already mentioned that types in OCaml are actually *partial types* in the sense that OCaml allows some expressions that may not terminate to be members of essentially all of its types. Thus the type of integers, `int` includes not only integers, but also expressions that "would be integers" if they terminated, but we do not yet know whether they terminate. Moreover, OCaml has a vast number of types that we will not study, several related to the idea of *objects*, the O part of OCaml. For example all of these concepts are implemented among the many OCaml types: objects, records, modules, first-class modules, functors, and classes. All of them are ways of grouping types and operations together. We will only study a couple of them in this course. We will study records and modules but not objects or classes.

The topic of partial types is important to a thorough understanding of modern programming and to the notion of correctness. We will discuss several topics in this category to give you a better idea of the directions in which programming languages are evolving and in which our understanding of computation is deepening. So it will be important for this course to know why `int` is a partial type. Suppose we have the

boolean valued functions `gr` to test whether an integer is greater than 0 and `le` to test whether less than 0.

```
let rec loop n:int :int if x = 0 then 0
else if gr(x) then loop(x+1) else loop(n-1).
```

We say that this program diverges on any non-zero input. So while `loop(0),loop(1)` both have type *int*, only one expression is actually an integer, namely 0. The other expression does not converge to an integer. We say it *diverges.* We gave a different but related example in Lecture 3.

As already mentioned, we write $\perp$ for a generic expression that has no canonical value because it "diverges," i.e. its computation does not terminate. We can even find $\perp$ in *bool* and *unit*! Why?

3.1 Polymorphic Types

We have already seen that polymorphic types allow us to express simple programming tasks. We noticed that the specifications look close to logical expressions. We saw examples like these. Recall that we mentioned that there is an OCaml type that is empty. We abbreviate it as *void*. It can play the role of $False$ in logic just as the unit type can play the role of $True$. Can we make sense of this type as an *implication*? $False \to True$?

unit, int, bool, char, string, exn
$\alpha \to \alpha$.
$void \to void$.
$(\alpha * \beta) \to \alpha$.
$(\alpha * \beta) \to L\alpha|R\beta$. [1]
$(\alpha * \beta) \to \beta * \alpha$.
$((L\alpha|R\beta) \to \gamma) \to L(\alpha \to \gamma)|R(\beta \to \gamma)$.
$((\alpha \to \beta) * (\beta \to \gamma)) \to (\alpha \to \gamma)$.

3.2 Additional polymorphic types and the idea of specifications and refinement

```
# fun ab -> (fun abc -> (fun x -> abc(x) ab(x) ));;
```

[1] Sometimes we will abbreviate this verbose expression as $(\alpha \mid \beta)$. To suggest the relationship to the Boolean "or" it would also make sense to write $\alpha||\beta$ when it is clear that we are talking about types.

```
- : 'a -> ('b -> 'a -> 'b -> 'c) -> 'b -> 'c = <fun>

# fun p -> fun d -> match d with |a -> fst(p) a |b -> snd(p) b ;;
- : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b = <fun>
```

Why does this code give the wrong type?

3.3 Program development by stepwise refinement

This section title comes from an excellent paper by Niklaus Wirth entitled *Program Development by Stepwise Refinement* [7]. His idea is that one should use a rich type system to specify what a program should do, and then construct the program by refining the type step by step until the program is constructed. We can do this for the above example. *That example had to work exactly backwards*! This is a problem with both OCaml and the Coq proof assistant. They both work this way – which is backwards. The Cornell proof assistant was built following Wirth's idea using the richest type system yet developed for mathematics and programming.[2] We will illustrate the refinement method from time to time in the course. We give a simple example next.

3.3.1 Program synthesis from a typed specification

Here is a very simple example of the refinement method for using a specification given as a type to guide the development of a program that has exactly that type. We use the *curry* function as an example.

Suppose we are given the type

$$((\alpha \star \beta) \to \gamma) \to (\alpha \to \beta \to \gamma).$$

How can we use it to guide the creation of the function being specified? We see that the overall structure of the specification is input function type to output function type. So it looks like this.

$$given \;\; f_1 : (\alpha \star \beta) \to \gamma) \; build - elmnt - in \; (\alpha \to \beta \to \gamma).$$

[2] That proof assistant has refinement in its name, PRL, for Proof/Program Refinement Logic. We are currently using version ν, so it is called NuPRL.

We see that to build the element of $(\alpha \rightarrow \beta \rightarrow \gamma)$, we can assume that we have inputs, say x1 of type α and x2 of type β. So this can generate the new goal below.

$$given \;\; f_1 : (\alpha \star \beta) \rightarrow \gamma), x1 : \alpha, x2 : \beta \; build - elmnt - in \; \gamma.$$

Now we see that to produce an element of γ we could use the function we named f_1, so we can say that, apply f_1. This sets up a new subgoal, to build an element of $(\alpha \star \beta)$.

$$given \;\; f_1 : (\alpha \star \beta) \rightarrow \gamma), x1 : \alpha, x2 : \beta \; build - elmnt - in \; (\alpha \star \beta).$$

We see that this goal is very easy to achieve because we have both $x1 : \alpha$ and $x2 : \beta$. So we can achieve that subgoal as follows.

$$given \;\; f_1 : (\alpha \star \beta) \rightarrow \gamma), x1 : \alpha, x2 : \beta \; build - elmnt - in \; (\alpha \star \beta) \; by \; (x1, x2).$$

Now we see that we can finish the job of building an element in γ by applying f_1.

3.3.2 Summary of the refinement method

The above procedure is a method of showing how to create the function we need to achieve our goal. We have broken the goal down into a series of steps driven by only the type information we know from the goal. This kind of step by step process can be implemented by an algorithm driven by the type information to synthesize a function that achieves exactly the stated goal. This is the process that my colleague Joseph Bates and I pursued using the framework that Robin Milner created for his LCF system [3, 1, 2]. The Nuprl system implements this method of using type specifications to drive the creation of objects of the given types. We can see the final product in this simple function definition.

```
# let curry = (fun h ->(fun x -> (fun y -> h (x,y)) ) ) ;;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>.
```

4 User defined list example

OCaml has a built in type of lists created with the template [; ; ;]. On the other hand, we can define our own version of lists as an example of a *recursive type*. We show that approach first, taking material from the 2008 version of the course using the above url.

```
type intlist = Nil | Cons of (int * intlist)

let length(lst: intlist): int =
  match lst with
    Nil -> 0
  | Cons(h,t) -> 1 + length(t)

 (* Return the last element of the list (if any) *)
let rec last(is: intlist):int =
    match is with
      Nil -> raise Fail("empty list!")
    | Cons(i,Nil) -> i
    | Cons(i,tl) -> last(tl)

 (* Return the ith element of the list *)
let rec nth (is: intlist) (i:int):int =
    match (i,is) with
      (_,Nil) -> raise Fail("empty list!")
    | (1,Cons(i,tl)) -> i
    | (n,Cons(i,tl)) ->
if (n <= 0) then raise Fail("bad index")
else ith(tl, i - 1)

(* Append two lists:  append([1,2,3],[4,5,6]) = [1,2,3,4,5,6] *)
let rec append(list1:intlist, list2:intlist):intlist =
    match list1 with
      Nil -> list2
    | Cons(i,tl) -> Cons(i,append(tl,list2))

(* Reverse a list:  reverse([1,2,3]) = [3,2,1].
 * Notice that we compute this by reversing the tail of the
 * list first (e.g., compute reverse([2,3]) = [3,2]) and then
 * append the singleton list [1] to the end to yield [3,2,1]. *)
```

```
let rec reverse(list:intlist):intlist =
    match list with
      Nil -> Nil
    | Cons(hd,tl) -> append(reverse(tl), Cons(hd,Nil))
```

5 What is a type?

This is a rewording of the corresponding section from Lecture 4 because we did not discuss it that lecture, and now we can say a bit more.

In computer science, types are a fundamental concept, analogous to the concept of sets in mathematics. However, they are different in very important ways that is worth understanding because types are in a sense co-developed with programming languages. It is important that in this course you know the difference between sets and types. You are probably a bit more familiar with sets because they are used in mathematics right from the start. There are two ways that sets are discussed: intuitive and axiomatic. This is discussed in Lecture 4.

First-Order Logic axiomatizes the logical operators and quantifiers: (&, $\vee$, $\Rightarrow$, $\sim$, $\forall$, $\exists$), used in the axioms for set theory. There are a dozen logical rules plus the set theory axioms. These are taken by many people as the foundations for mathematics. But this is not an appropriate foundation for computer science because these axioms do not capture computability as a primitive notion. Moreover, when computation is defined in set theory, say using Turing Machines or an equivalent account, the logical language has no computational primitives that allow users to *experience computational facts.* There are "no main stream programming languages" that use sets instead of types. *That is because set theory has no built in computational primitives, so even the set of functions from integers to integers is not defined in reference to computable functions.*

We will see later that there were several very eminent mathematicians who complained about using set theory as a foundation precisely because it did not allow users to experience the basic truths of mathematics. We will see that L.E.J. Brouwer believed that all of mathematics is founded on computational experience and that "there are no non-experienced

mathematical truths" [4]. We will examine this idea at various points in the course where it is especially relevant.

Types are based on a computation system defined on untyped expressions. OCaml uses small step evaluation semantics. This approach is due to another Edinburgh University professor, Gordon Plotkin [6]. Gordon was one of the computer scientists who saw that the Logic of Computable Functions (LCF) was also a nice programming language [5].

To summarize again as in Lecture 4, types are a collection of *canonical values* from the computation system with a notion of equality on them. As canonical values, the expressions stand on their own. The meaning of the expressions arises from relating the canonical and non-canonical values. For example, consider the list `[1;2;3]`. The relationship between the constructors and the destructors starts to reveal their "meaning." When we say `hd [1;2;3] = 1` then we experience the meaning of the *hd* operation. We start to "understand" what a list is by applying operations to build them and others to take them apart.

How do we capture this meaning in the language of the OCaml type theory? We use the *operational semantics* we saw in the second lecture. We will also explain later that types can be understood as *partial equivalence relations on expressions*. From this definition, we already see that before we can understand an OCaml type, we need to know the computation rules relevant to it. That is why we started by discussing evaluation and canonical values. Without those ideas, we cannot understand types in computer science. In contrast, to understand sets in mathematics, we never mention computation.

References

[1] J. L. Bates and Robert L. Constable. Definition of micro-PRL. Technical Report 82–492, Cornell University, Computer Science Department, Ithaca, NY, 1981.

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

[3] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[4] A. Heyting, editor. *L. E. J. Brouwer Collected Works*, volume 1. North-Holland, Amsterdam, 1975.

[5] Gordon D. Plotkin. LCF considered as a programming language. *Journal of Theoretical Computer Science*, 5:223–255, 1977.

[6] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[7] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.