CS3110 Spring 2017 Lecture 6
Building on Problem Set 1

Robert Constable

1 Lecture Plan

1. Repeating schedule of remaining five problem sets and prelim.
2. Expressing PS1 related concepts in type theory.
3. More on types compared to sets.

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 Logical specifications compared to type theory specifications

We have already seen that some *simple polymorphic OCaml types resemble propositional formulas* from logic. This analogy is worth investigating further because it reveals a very strong connection between types and propositions that has been explored in computer science and logic for the past four decades, and it has been implemented in proof assistants such as Agda, Coq, and Nuprl [2, 1, 3]. It is one of the many deep connections

between computer science, mathematics, and logic. It is also having an impact in philosophy, for instance in epistemology.

3.1 numerical specifications

In PS1, we use standard logic to specify some of the programming problems. For example, Exercise 6, we asked to define the *prime numbers* among the positive integers **P**. We say that a prime number is a positive integer that has precisely two positive integer divisors, 1 and itself.[1] In the PS1 problem set you implemented an OCaml decision for primality and implemented an OCaml function to enumerate the primes. Can we specify these ideas precisely using types? The answer is a strong yes, we can specify them in a plausible way. Let us consider the features of natural numbers that lead to a simple definition.

```
7  is divisible by 1 and 7 and is prime.
8  is divisible by 1,2,4,8 and is not prime (composite).
9  is divisible by 1,3,9 and is not prime.
12 is divisible by 1,2,3,4,6,12 and is not prime.
13 is divisible by 1,13 and is prime.
15 is divisible by 1,3,5,15 and is not prime.
17 is divisible by 1,17 and is prime.
20 is divisible by 1,2,4,5,10 and is not prime.
```

Can we specify a defining property using OCaml types? We cannot specify all numerical computing tasks using only OCaml types, but we can come close in many cases. We illustrated this idea in the first problem set, and we pursue it further in this lecture.

Using the quantifiers and connectives of logic we define $prime(p)$ iff

$$(\forall x, y : \mathbb{N}.(0 < x \leq y \ \& \ (x \times y) = p)) \Rightarrow x = 1 \ \& \ y = p)$$

.

What does this say in words?

Another way to write $prime(p)$ for p an integer is

$$\forall x, y : int.((0 < x \ \& \ x \leq y) \ \& \ (x \times y = p)) \Rightarrow (x = 1 \ \& \ y = p).$$

[1] For many years 1 was considered to be a prime number, but by the 20th century there were many good reasons to disallow 1 as a prime. Sometimes the negative of a positive prime is considered a prime.

In the OCaml type theory, we can write this definition as a boolean valued function of an integer input, say $prime(n)$. We first need to define an auxiliary function to let us talk about the positive integers. It is easy to define it in OCaml. The positive integers are those integers n for which we know $n > 0$. We can specify this using a boolean valued test because the order relation is an OCaml primitive on integers. The *positive* integers can be defined as those integers for which we know $n > 0$. We can capture this with the idea of a **dependent pair** written as $x : int \star \ where \ x > 0$. We will see that this concept can be defined in OCaml. It is a basic concept in modern computational type theory. To say that a positive integer is prime in OCaml, we use the Boolean operators. In this example, we write the OCaml Boolean **and** operator as $\wedge$ rather than as the double ampersand.

$prime(p) =$

$$pr : (int \star int) \rightarrow (0 \ < \ fst(pr)) \wedge (fst(pr) \ \leq \ snd(pr)) \wedge (fst(pr) \times snd(pr)) = p$$

$$\rightarrow$$

$$(fst(pr) = 1) \wedge (snd(pr) = p).$$

In this example we are using *explicit types* not polymorphic types. We can also use explicit types directly in the code as in examples like $fun(x : int) \rightarrow x + 1$.

3.2 Deriving code for prime testing from the specification

Now that we have a specification using types, we can create the code for testing primality using the idea of step wise refinement discussed in Lecture 5. We see that the $prime(p)$ function takes as input p and pairs of integers (rather than natural numbers). So we have $pr : (int \star int)$ as the first input.

The additional inputs are the three Boolean relations $b1 : (0 \ < \ fst(pr))$, $b2 : (fst(pr) \ \leq \ snd(pr))$, $b3 : (fst(pr) \times snd(pr)) = p$. From these inputs, we must conclude that $(fst(pr) = 1)$ and $(snd(pr) = p)$. If we can draw that conclusion, then p is prime.

These inputs establish a simple programming task of checking whether for all the pairs, pr in this range, we can prove that only $(1, p)$ satisfy the conditions. This is a simple matter of testing all the possible natural number pairs in this range. That is the core of the algorithm. The step by

step refinement has not produced the algorithm until we specify how to check all of the pairs. On the other hand, the proof outline leads directly to the tests that must be carried out to finish the proof for a particular input p. So we have established essential features of the algorithm from its precise type theoretic specification.

A more complete specification would tell us when a candidate for being prime is not actually prime. In this case it would return two positive integers other than 1 and p whose product is p. It is an interesting exercise to see how to broaden the approach to cover this case as well and provide a prime testing procedure. It would not be the polynomial time procedure discovered in 2002 by Agrawal, Kayal, and Saxena of the Indian Institute of Technology (IIT) Kampur in their paper *Primes is in P*. That is a brilliant and sophisticated algorithm, too advanced for CS3110.

3.3 Comments on polymorphic type specifications

We have already seen that polymorphic types allow us to express simple programming tasks. We noticed that the specifications look close to logical expressions.

What about the type

$$(L(\alpha \to \gamma)|(R(\beta \to \gamma)) \to (L\alpha|R\beta) \to \gamma).$$

Is there a program in this type? When seen as a logical expression, is it true?

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))
```

4.1 More on types compared to sets

In Lecture 5 we started to compare types and sets. This will be a theme that we weave though out the course. Even for the very simple example of natural numbers and integers, we can see both the similarities and differences. In this subsection, we notice some superficial differences. The most obvious difference might be in the treatment of functions. To define a function from $\mathbb{N}$ to $\mathbb{N}$ in set theory, we only need to create a set of ordered pairs such that any two pairs with the same first element have the same second element. In type theory we need to produce a program to compute the function, and type $\mathbb{N} \to \mathbb{N}$ consists not of a collection of ordered pairs but of a collection of *computable functions* given as OCaml programs.

To summarize as in Lecture 4, types are a collection of *canonical values* from the computation system. There is a notion of equality defined 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.

We are starting to see how we capture this meaning in the language of the OCaml type theory. We use the *operational semantics* from the second lecture. 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. We start with much more abstract concepts. It is an interesting challenge for type theory to achieve more abstraction while preserving computational meaning.

References

[1] 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.

[2] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[3] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ, 1986.