CS3110 Spring 2017 Lecture 10 a Module for Rational Numbers

Robert Constable

Abstract

The notes and lecture start with a brief summary of the relationship between OCaml types, Coq types and logic that we investigated in Lecture 9.

This lecture begins a discussion of *modules* in OCaml and module *interfaces*. These language constructs allow us to collect definitions of types and related functions into one syntactic object called a module. The textbook discusses modules at length as do previous lecture notes for other versions of the course. These notes for Lecture 10 will not stress the connection to files and compilation. Instead they present modules as a way to organize a related group of types and functions and treat them almost as if they were primitives in OCaml.

We will use modules to organize an implementation of rational numbers and later introduce another module to implement the constructive real numbers. Modules provide the basis for *abstract types.* In Classic ML, the original ML programming language from which OCaml evolved, there were no modules, and the ideas discussed here were presented in terms of *abstract data types.* We briefly mention the reason that Classic ML created these types. They are relevant to the theme of *problem specification* and correct programming that we have already discussed as motivation for many concepts taught in this course. OCaml was also designed to support these ideas.

Note, **the in class prelim** on Tuesday March 14 will definitely have at least one question on the relationship of OCaml polymorphic types to logic. This lecture summarizes the key points on this topic.

1 Summary of using types as logical specifications

We have seen that some programming concepts are also logical concepts. This is revealed well in the correspondence between OCaml types and the computational interpretation of propositional logic. We have seen in some detail the correspondence summarized below. This correspondence is used in Coq and is extended to include the quantifiers "for all" ($\forall$) and "there exists" ($\exists$).[1]

- True – trivial evidence, say (), corresponding to the unit type.
- False – no evidence, corresponding to the void type.
- A & B – evidence a in A and b in B collected in a pair (a,b).
- A $\rightarrow$ B – there is a function taking evidence for A into evidence for B.
- A $\vee$ B – there is either evidence La for A or Rb for B.
- $\neg$ A – for the negation of A, meaning a function from A into False.

These ideas concerning a computational interpretation of logic originated with the famous Dutch mathematician L.E.J. Brouwer [?, ?, ?]. He called his approach to mathematics *intuitionism.* It is the idea that all mathematical truths are based on primitive intuitions of the human mind about time that allows us to experience mathematical truths first hand as mental constructions, one step following another in time.

The American logician S.C. Kleene, a PhD student of Church, visited Amsterdam just before the war and learned how to formalize Brouwer's ideas. After the war he wrote a very influential article [?] that put Brouwer's concepts into the logical mainstream. Another such publication was written by Brouwer's student Heyting [?]. Kleene continued this line of research to provide an account of real numbers and analysis [?]. We will look at one of these ideas later in the course. I was a student of Kleene's and found practical uses for these ideas in computer science that led to the Nuprl proof assistant.

[1]We know that Coq uses only "for all" as its logical primitive and defines "there exists."

2 Defining rational numbers in OCaml

It is not difficult to define a concrete OCaml type corresponding to the rational numbers. It will be pairs of integers, say (n,d), for the numerator and denominator, and we require that d is non-zero. As a *dependent type*, this can have a very elegant form, say $p : (int \times int)\ where\ (snd(p)\ \neq\ 0)$. This is not an official syntax in either OCaml or Coq, yet it shows how we use dependent type expressions informally when we are writing or talking about data types informally. This kind of expression shows that dependent expressions are quite natural. We need to mention the pair $(int \times int)$ since it is the type on which the rationals are based. We also normally just say "and the denominator is not zero," i.e. $(snd(p)\ \neq\ 0)$. When we say that, the reference to the denominator is dependent on the type named by p. We say that the second component of the pair is non zero.

If we proceeded in this way, we would then define all the standard operations on rationals such as add, subtract, multiply, and divide and perhaps many other basic functions, e.g. absolute value, inverse, etc. along with basic relations on rationals such a equality, less-than,absolute value and so forth. We would like to make the rational numbers look like a built-in type where we can find the primitive operations on rational numbers. If we simply define the appropriate type of ordered pairs, we miss out on the natural convenience of treating the type in a manner similar to how we treat the built-in primitive types such as int, bool, string, list and so forth.

We also need to face the question of the normal form for rationals, i.e. that 1/2 and 5/10 and 6/12 are all equal rational numbers. We write the *equality relation* among two rational numbers as $exp_1\ =\ exp_2\ in\ \mathbb{Q}$.

OCaml provides a mechanism for collecting the operations and tests we want on the rational numbers into one structure called a module. At the core is the definition of the type and then a list of commonly used functions that come with it in the module. The module can be extended to include more functionality. In this sense the module is a construct private to a user or group of users. So to share code, it is necessary to share the module as well.

Before we look at the notion of a module for collecting information about the rationals, denoted $\mathbb{Q}$, we want to remind ourselves about the key properties of the rationals. We review these below.

We will now define the type of rational numbers and some of the basic functions for computing with them. We want to confirm that the implementation is correct by showing that the standard arithmetic laws apply. These can be expressed as equalities on the primitive operators, as we list below.

x, y, z: rational.

Commutative	$x + y \;=\; y + x$	$x \star y \;=\; y \star x$
Associative	$(x + y) + z \;=\; x + (y + z)$	$(x \star y) \star z \;=\; x \star (y \star z)$
Identity	$x + 0 = x$	$x \star 1 = x$
Distributive	$(x + y) \star z \;=\; x \star z + y \star z$	
Inverse	$x + (-x) = 0$	$x \star (y \backslash x) \;=\; y$ if $x \neq 0$

$< \ \mathbb{Q}, +, \star, -, \backslash, \ 0, \ 1 \ >$ is a "field" in the sense of abstract algebra.

Abstract 'Interface'

We now examine how to create a module that uses an interface, denoted *mli*, to provide the abstract signature that defines the module of rational numbers. These ideas are discussed and illustrated in Chapter 4 of *Real World OCaml.* The textbook explains the notion of signatures and abstract types. It also notes that in the context of OCaml, "the terms interface, signature, and module type are all used interchangeably."

There are excellent CS3110 lecture notes on modules. I list two of the here.

`http://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec09-functors/functor`

`http://www.cs.cornell.edu/courses/cs3110/2016fa/l/07-modules/notes.html`

Here is the OCaml abstract type for rational numbers, *rat.mli*, defined using a module with an interface (indicated by mli).

rat.mli (We could use *big_int* instead of int.)

```
type rat
val rat0 : rat
val rat1 : rat
val add_rat : rat -> rat -> rat
val mul_rat : rat -> rat-> rat
val inv_rat : rat -> rat
val aminus_rat: rat -> rat
val mk_rat : (int * int) -> rat
```

rat.ml

```
type rat = int * int
  let mk_rat(a,b) =
      if b = 0 then failwith "mk_rat: 0 denominator"
      else (a,b)
let rat0 = (0,1)
let rat1 = (1,1)
```

2.1 GCD algorithm

We want to present rational numbers in a canonical form that removes the common factors in the numerator and denominator. We do this by "dividing out" the greatest common divisor.

See 'An Algorithm for the Greatest Common Divisor' by Anne Trostle on the Nuprl website for a more detailed explanation, and the implementation in Nuprl.

```
http://www.nuprl.org/MathLibrary/gcd/
```

$$\frac{1}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{bc}{bd} = \frac{ad+bc}{bd}$$
$$(\frac{1}{2}) + (\frac{1}{2}) = \frac{1*2+2*1}{2*2} = \frac{4}{4}$$

We want to put $\frac{a}{b}$ in *lowest terms.*

```
let reduce_rat(a,b) =
    let g = gcd a b in
      (a/g, b/g)
```

```
let rec gcd a b =
    if b = 0 then a
    else gcd b (a mod b)

# let rec gcd a b =
    if b = 0 then a
    else gcd b (a mod b)  ;;
val gcd : int -> int -> int = <fun>

# let reduce_rat(a,b) =
    let g = gcd a b in
      (a/g, b/g) ;;
val reduce_rat : int * int -> int * int = <fun>

# reduce_rat (5,27);;
- : int * int = (5, 27)
# reduce_rat (5,25) ;;
- : int * int = (1, 5)
```

Example of the gcd algorithm:

`gcd` 70 18	$70 = 3 * 18 + 16$	$70 = 7 * 5 * 2$
`gcd` 18 16	$18 = 1 * 16 + 2$	$18 = 3 * 3 * 2$
`gcd` 16 2	$16 = 8 * 2 + 0$	
`gcd` 2 0		
2		

$g =$ `gcd` $a\ b$ is the greatest common divisor of a, b.

$g =$ `gcd` $a\ b \Rightarrow$ for some u, v: $g = ua + vb$.

Prove this by induction on $|b|$.

```
if |b| = 0
  then b = 0
    g = gcd a 0 = a
```

We need $a = \_\_\_ * a + \_\_\_ * 0$, (1,0) solves this.

```
if |b| > 0
```

```
then b != 0
   g = gcd a b = gcd b (a mod b)
```

1. $a = qb + (a \bmod b)$, when $q = a \div b$.

   $|a \bmod b| < |b|$

   So for some u, v:

2. $g = ub + v * (a \bmod b)$

   $va = vqb + v * (a \bmod b)$

3. $g - va = ub - vqb$

   $\qquad\qquad (u - vq)b$

   $g = va + (u - vq)b$

References