Abstract

This lecture begins our 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. These notes for Lecture 5 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 apply them to defining the abstract type of rational numbers.

We will use modules to organize our implementation of rational numbers and later introduce another module to implement the computational 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 only 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 built in large part to support these ideas.

1 Lecture Topics

1. History of ML: Classic ML, Standard ML (SML), and OCaml
2. LCF - logic of computable functions
3. ML - meta-language for LCF
   - programming proofs for LCF
   - guarantee that proofs use only the rules of LCF
4. Type system for ML
5. Module for rational numbers
6. Normalizing rational numbers using the gcd algorithm

Top down proof style

In the 1970’s computer scientists were exploring how to make programming much more reliable. Several of the most cited and honored research was devoted to this area. Two British computer scientists, C.A.R. Hoare (now Sir Hoare) and Robin Milner were leaders in these efforts as was the Stanford computer scientist John McCarthy (the creator of Lisp) and Dana Scott who had moved to Oxford University. There are many other very eminent researchers in this area that are not mentioned in this lecture. We will mention others as the course progresses. Robin Milner was visiting Stanford and designing a functional programming with features that supported reasoning about programs. He and his colleagues were implementing a logic for reasoning about computable functions based on ideas proposed by Dana Scott [10]. In the 1970’s Robin moved to Edinburgh University in Scotland and continued developing his approach. It was called _Edinburgh LCF_ [4] because Milner and his team built the system at the University of Edinburgh in Scotland. It became a very influential system.\footnote{Robin and the second author on these notes became close friends and colleagues, so I can tell stories about his personal role in computer science. He won the Turing Award for this and other fundamental contributions – see the url. He was advising the two authors of these notes on our logic of events when he died in 2010 and the CS community mourned.}

http://amturing.acm.org/award_winners/milner_1569367.cfm

Edinburgh LCF provided a functional programming language for implementing the LCF logic and writing formal proofs in it. This functional language is often called _Classical ML_, the letters ML abbreviate Meta-Language, a central concept in logic. The ML type system includes types for logical formulas and a precise definition of _formal proofs_. ML implemented the concept of an _abstract type_ to enforce that proofs can be built only using the formal rules of the logic as expressed in ML. There is no other way for normal users to create a proof than to build it in the abstract type of LCF proofs. Every such object will be a legal proof. The Classical ML type system makes precise the concept of an LCF proof. This
implementation of proofs can be shown faithful to the notion of proof as defined by logicians studying the foundations of mathematics, such as Hilbert, Gödel, Church, and Turing among others.

The Edinburgh LCF approach is very general and is now used by other projects to implement more expressive logics and support the creation of formal proofs in them. Systems that support the creation of formal proofs are called proof assistants, and they have been built for various logics. Edinburgh LCF was one of the first; Cambridge LCF followed, creating the HOL proof assistant \[8, 9, 7\]. The Nuprl proof assistant for type theory uses Classic ML \[2\]. The Coq proof assistant \[3, 6, 1\] uses a related metalanguage and is implemented in OCaml as is the Meta-PRL \[5\] proof assistant.

Here is a diagram that shows the structure of a proof as a tree of objects called sequents.\(^2\) The only way to build a proof tree is by applying the constructors for the abstract Classical ML type of LCF proofs.

\[
\vdash \text{for all } L : t \text{ list } A(L) \text{ and } B(L) \\
\mid \\
L : t \text{ list } \vdash A(L) \text{ and } B(L) \\
/ \quad \backslash \\
\vdash A([ ]) \quad L : t \text{ list } A(L), x : t \vdash A(x.L) \\
/ \quad \backslash \\
\vdash
\]

The value of an abstract type is especially clear in the case of proofs. The LCF designers required a mechanism to ensure that \textit{that only logically correct methods could be used to build formal proofs}. They reduced that idea to applying the constructors of the abstract type of LCF proofs. The Classic ML programming language was designed to write programs called tactics that are safe programming tools for building formal proofs. Nuprl tactics are Classic ML programs that build proof objects in the ML abstract type of Nuprl proofs.

\(^2\)These trees are frequently drawn top down, with the root at the top and leaves at the bottom.
Algebraic properties of rational numbers

We now apply the notion of an abstract type to define the type of rational numbers and the basic functions to compute 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.

\[ x, y, z : \text{rational.} \]

- **Commutative** \[ x + y = y + x \]
- **Associative** \[ (x + y) + z = x + (y + z) \]
- **Identity** \[ x + 0 = x \]
- **Distributive** \[ (x + y) * z = x * z + y * z \]
- **Inverse** \[ x + (-x) = 0 \]

\(<\text{rational, } +, *, - , \frac{1}{z}, 0, 1>\) is a “field”.

Abstract ‘Interface’

We now examine how to create a module that uses an interface, denoted \textit{mli}, to provide the abstract signature that defines the module of rational numbers. These ideas are discussed and illustrated in Chapter 4 of \textit{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.” Here is the OCaml abstract type for rational numbers, \textit{rat.mli}, defined using a module with an interface (indicated by mli).

\texttt{rat.mli (You should really use \texttt{big_int} instead of \texttt{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

\texttt{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)

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/

\[ \frac{1}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \]
\[ \left( \frac{1}{2} \right) + \left( \frac{1}{2} \right) = \frac{1 \times 2 + 2 \times 1}{2 \times 2} = \frac{4}{4} \]

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)
Example of the gcd algorithm:
\[
gcd \ 70 \ 18 \quad 70 = 3 \times 18 + 16 \quad 70 = 7 \times 5 + 2
\]
\[
gcd \ 18 \ 16 \quad 18 = 1 \times 16 + 2 \quad 18 = 3 \times 3 + 2
\]
\[
gcd \ 16 \ 2 \quad 16 = 8 \times 2 + 0
\]
\[
gcd \ 2 \ 0
\]
\[
g = \gcd \ a \ b \text{ is the greatest common divisor of } a, b.
\]
\[
g = \gcd \ a \ b \Rightarrow \text{ for some } u, \ v: \ g = ua + vb.
\]
Prove this by induction on \(|b|\).

if \(|b| = 0\)
then \(b = 0\)
\[
g = \gcd \ a \ 0 = a
\]

We need \(a = __ \times a + __ \times 0\), \((1,0)\) solves this.

if \(|b| > 0\)
then \(b \neq 0\)
\[
g = \gcd \ a \ b = \gcd \ b \ (a \mod b)
\]

1. \(a = qb + (a \mod b)\), when \(q = a \div b\).
\[
|a \mod b| < |b|
\]
So for some \(u, v\):

2. \(g = ub + v \times (a \mod b)\)
\[
va = vqb + v \times (a \mod b)
\]

3. \(g - va = ub - vqb\)
\[
(u - vq)b
\]
\[
g = va + (u - vq)b
\]
References


