New Types

Michael P. Fourman

October 29, 2006

Aims

In this practical, you will learn to use products to represent other types of data, and use ML structures and signatures to control the specification and implementation of a new type.

Assessment

Your work will be assessed on the basis of the correctness of the two structures you implement. Part 1 carries 60% of the marks for this practical; the remaining 40% are allocated to Part 2.

Introduction

This practical continues the mathematical flavour of the first practical. However, this time, some more interesting data types are involved. The exercises cover two different kinds of useful mathematical object: rational numbers, and approximate real numbers.

Both of these are types of number. For each of them, you asked to provide an implementation matching the signature `NumberSig`.

```
infix 4 ==;
infix 6 ++ --;
infix 7 ** //;

signature NumberSig =
sig
(* Type *)
   type num;

(* Binary Operations (infix) *)
   val ++ : num * num -> num
   and -- : num * num -> num
   and // : num * num -> num
   and ** : num * num -> num

(* Unary Operations *)
   and ~~      : num -> num

(* Predicates (infix) *)
   and == : num * num -> bool
end;
```

To avoid confusion with the existing arithmetic operators, we call the operators `++`, `--`, `**`, `//`, , and `==`; we make the binary operators infix, and give them the same precedence as their real counterparts.
For this practical, you should use the command `ml prac2` to start your ML session. This provides a few general functions[1], including the functions `error` and `time` you used in Practical 1, and includes the following infix directives:

1 Arithmetic Operators for Rational Numbers

Suppose that we want to do arithmetic with rational numbers. That is, we want to be able to add, subtract, multiply and divide them and also to test whether two rational numbers are equal. The following relations define these operations in terms of the numerators and denominators of the rational

[1] To see all the functions provided, open the structure `Prelude`

numbers:

$$\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}; \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}; \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2}; \\
\frac{n_1/d_1}{n_2/d_2} &= \frac{n_1 d_2}{d_1 n_2}; \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} & \quad \text{if and only if } \; n_1 d_2 = n_2 d_1.
\end{aligned}$$

Rational numbers may be represented in SML as pairs of integers. Your code should include the following lines:

```
(* rationals -- (numerator, denominator)*)
 type num = int * int;
```

Your code for this part should be placed in a file `CS201/Prac2/Rational.ML` The exercise involves doing two things:

1. Define these arithmetic operations in SML, and place them, together with the type declaration in a structure `Rational:NumberSig`.

2. Modify your functions so that rational numbers are always reduced to lowest terms. For example, evaluating `(1,3) ++ (1,3)` should produce `(2,3)`, not `(6,9)`.

2 Interval Arithmetic

Suppose now that we wish to manipulate inexact quantities (such as those measured parameters of physical devices) with known precision so that, when computations are done with these approximate quantities, the results will be numbers of known precision. One approach is to represent a number by an *interval* that gives the range of possible values of an inexact quantity. Then the result of adding, subtracting, multiplying or dividing two intervals is itself a new interval representing the range of the result. We can represent an interval by a pair of reals:

```
(* interval of uncertainty  (lowerBound, upperBound) *)
type num = real * real;
```

To simplify the analysis, we will always assume that the first number is no larger than the second number. (Later in the course, we will see how this convention can be enforced in the SML language through the use of abstract datatypes.)
You should provide an implementation of interval arithmetic `Interval:NumberSig`.
Your code for this part should be placed in a file `CS201/Prac2/Interval.ML`
The exercise involves doing two things:

1. Define the operations `++`, `--`, `**`, `//`, , and `==` in SML. [For division, it might help to consider multiplying the dividend by the reciprocal of the divisor. Furthermore, if division is undefined on an argument, then call the function `error` to terminate the execution.] Don't forget to start SML using `ml prac2` as `error` will not be defined otherwise. Place these functions in a structure `Interval:NumberSig`.

2. The formula for calculating the combined resistance of two resistors placed in parallel can be expressed in two algebraically equivalent ways:

$$\frac{R_1 R_2}{R_1 + R_2} \qquad \text{and} \qquad \frac{1}{1/R_1 + 1/R_2}$$

   (where R_1 and R_2 are the two resistances). Investigate whether your functions always produce the same result using these two equations. If they differ, which formula produces tighter error bounds? Can you explain why?