Mastering Multiprecision Arithmetic

Norman Ramsey

Spring 2019

Introduction

Uncivilized programming languages, including Impcore and μScheme, provide crappy integer arithmetic. You better hope your results fit in a machine word—if they don't, your program might silently produce wrong answers.

Civilized languages provide integer arithmetic that works on any integer up to the size you can fit in RAM. By this standard, Standard ML is not quite civilized: arbitrary-precision arithmetic is in the specification as module `IntInf`, but implementations don't have to provide it. Full Scheme, on the other hand, is super-civilized—if you're interested in arithmetic, it's worth studying.

Arithmetic plays a prominent role in our study of abstraction mechanisms, for these reasons:

- Civilized arithmetic is a topic that every educated computer scientist should know about—if only so that you can insist on it for your projects. Don't tolerate uncivilized arithmetic!
- Arithmetic brilliantly illustrates the relative strengths and weaknesses of the two major ways of abstracting over data: abstract data and versus objects.

Basics

A natural number is represented by the sum $\sum_{0 \leq i < n} x_i \cdot b^i$.

- b is called the *base* of natural numbers.
- Each x_i is called a *digit*. It satisfies the invariant $0 \leq x_i < b$.

Hardware integer arithmetic universally uses base $b = 2$. But software arithmetic, which you will implement, typically uses a base $b = 2^k$, where k is chosen as large as possible, subject to the constraint that $(b-1)^2$ can be computed without overflow. Be cautious! Moscow ML provides only 31 bits of integer precision.

To help you get started, it is reasonable to begin with

```
val base = 7
```

Once your code is working, you'll want to replace 7 with a larger number. Look at the section "Choosing a base of natural numbers" in the book.

Performance matters. Larger b gives better performance.

Choice of representation

A natural number should be represented by a sequence of digits. But this still leaves you with interesting design choices:

- Do you prefer an array, a list, or the algebraic data type used on the first ML assignment?
- If you prefer an array, do you want a mutable array (type `'a array`) or an immutable array (type `'a vector`)?
- If you prefer a list, do you prefer to store the digits of a number in big-endian order (most significant digit first) or little-endian order (least significant digit first)?

  (If you prefer an array, the only order that makes sense is to store the least significant digit x_0 in slot 0, and in general, to store digit x_i in slot i. Anything else would be confusing.)
- If you prefer an array, how will be you be sure it contains enough digits to hold the results?
- If you prefer a list, how will you make sure you produce a list with the right number of digits?

Some implementations of arithmetic operations, especially multiplication, can leave leading zeroes in a representation. The number of leading zeroes must be controlled so that it doesn't grow without bound.

- Do you prefer a representation invariant that eliminates leading zeroes?
- If not, how will you track the number of leading zeroes in each representation? Will you store it in the representation itself, or will you compute it dynamically?

The beauty of data abstraction is that *the interface isolates these decisions, so you can easily change them.*

Representing individual digits

You can represent a digit by a value of type `int`. But if you like arrays, I recommend choosing a different representation: a value of type `int` is too easily confused with an array index. Instead, I recommend that you define

```
datatype digit = D of int
```

and use `digit array` or `digit vector` in your representation. You will have to do a little extra pattern matching, but in order to avoid confusing an index with the digit at that index, the extra pattern matching is well worth it.

Mutable representation? Really?

If you scrutinize the types and contracts of the functions in the `NATURAL` interface, you'll see that the `nat` abstraction is *immutable*: nothing in the interface allows a client to change the value of a natural number. So why would it be OK to choose a *mutable* representation, like a `digit array`? Simple: you're allowed to pick any representation you like, because the details are sealed behind the interface. As long as you don't make a silly mistake like allow `x /+/ y` to mutate `x` or `y`, no client program can tell that your representation is mutable. In industrial implementations of multiprecision numbers, mutable representations are quite popular: they minimize allocation and they use memory efficiently. It's fine to pick a mutable representation, as long as no client code can tell the difference.

Tackling arithmetic in Standard ML

If you study the `NATURAL` signature, you should observe that the first step in creating any natural number is always to call `of_int`, and the only way to get information out of a natural number is to call `decimal`. But you will have an easier time if you start with two auxiliary functions, which are closely related. If your representation is immutable, here are their specifications:

```
val addInt : nat * int -> nat
  (* addInt (x, p) returns a natural number
     that represents x + p,
     provided the following condition holds:

       for every digit x_i in x, (p + x_i)
       can be computed without overflow
   *)

val shiftAdd : int * int * nat -> nat
   (* shiftAdd (p, n, x) = p * power (base, n) + x,
      provided the same condition above holds *)
```

These two functions are related by this property:

```
shiftAdd (p, 0, x) = addInt (x, p)
```

This property can be used in either direction to implement one of the two functions: you can use `shiftAdd` to implement `addInt`, or you can use `addInt` to implement `shiftAdd`. Which direction is easier depends on your choice of representation.

If your representation is mutable, the types and contracts will be slightly different:

```
val addInt : nat * int -> unit
  (* addInt (x, p) overwrites x with x + p,
     provided p is not too large as above *)

val shiftAdd : int * int * nat -> unit
   (* shiftAdd (p, n, x) overwrites x with
      p * power (base, n) + x,
      provided p is not too large as above *)
```

Implementing function `of_int`

Once you have `addInt`, you can use it to implement `of_int`. Depending on the mutability of your chosen representation, the details vary. Here's the immutable version:

```
fun of_int n =    (* immutable representation *)
  if n < 0 then raise Negative
  else
    let val zero = ... something suitable ...
    in  addInt (zero, n)
    end
```

The mutable version is the same, except you call `addInt` for side effect and then return `zero`.

Implementing short division

In short division, a multi-digit number is divided by a single digit. The algorithm is one you learned in primary school. The algorithm is explained in *Programming Languages: Build, Prove, and Compare*, in the section on arbitrary-precision integer arithmetic.

If you prefer arrays, a full implementation is presented in David R. Hanson's *C Interfaces and Implementations*, on pages 311 and 312 (the `XP_quotient` function), which you are welcome to adapt.[1] What Ramsey calls a "remainder" r_i is called `carry` by Hanson.

If you are using an array representation, you can write the main loop using higher-order function `Array.foldri`. Try `help "Array";` in Moscow ML. The accumulating parameter holds the remainder r_{in}. The final remainder out is returned with the quotient.

If you prefer lists, the equations in the book imply the following algebraic laws:

$$\begin{aligned}
0 \text{ div } v &= 0 \\
0 \text{ mod } v &= 0 \\
(m \cdot b + d) \text{ div } v &= (m \text{ div } v) \cdot b + ((m \text{ mod } v) \cdot b + d) \text{ div } v \\
(m \cdot b + d) \text{ mod } v &= \qquad\qquad\qquad ((m \text{ mod } v) \cdot b + d) \text{ mod } v
\end{aligned}$$

These laws *must not be used directly for implementation*. Why not? Because in a direct implementation, the inductive case calls *both* `div` and `mod` recursively. That's an exponential number of calls. For implementation, you need to compute both m mod v and m div v in a *single* call. That's why the `sdiv` function returns a record containing both quotient and remainder. (This is also how integer-division hardware works.)

Using short division for base conversion

To convert a number from a large base to a smaller base, you continually short-divide by the small base until you run out of digits. Each short division produces a remainder, which is a digit of the result. The least-significant digit is produced first.

[1] If you took COMP 40 at Tufts, you may have this book. If not, it is part of the university's Safari subscription, so your access to it is already paid for.

Here's an example using repeated division by 2, converting decimal 11 to binary:

$$\begin{aligned} 11 \div 2 &= 5 \text{ remainder } 1 \\ 5 \div 2 &= 2 \text{ remainder } 1 \\ 2 \div 2 &= 1 \text{ remainder } 0 \\ 1 \div 2 &= 0 \text{ remainder } 1 \end{aligned}$$

Reading off the digits in order says that decimal 11 is binary 1011.

There is a similar problem on the homework, where you complete this table:

$$\begin{aligned} 13 \div 2 &= q_0 \text{ remainder } r_0 \\ q_0 \div 2 &= q_1 \text{ remainder } r_1 \\ q_1 \div 2 &= q_2 \text{ remainder } r_2 \\ q_2 \div 2 &= q_3 \text{ remainder } r_3 \end{aligned}$$

The converted numeral is the sequence of digits $r_3 r_2 r_1 r_0$ (in traditional big-endian order).

Implementing `decimals`

Short division is the way you implement `decimals`. *You have to get this right* or your code won't pass any test cases. The key idea is that the least significant decimal digit is "the argument mod 10," and the remaining digits are the digits of "the argument div 10." Here is part of a definition of `decimals`, using an immutable representation:

```
fun reverse_decimals n =
  if ... n is zero ... then
    []
  else
    let val { quotient = q, remainder = r } =
          sdiv (n, 10)
    in  r :: reverse_decimals q
    end
```

Implementing addition and subtraction

If you are using an array to represent a sum or product, you will need to provide it with enough digits in advance:

- For addition, you may need as many digits as are in the larger addend, plus one more.
- For subtraction, you may need as many digits as are in the minuend (left-hand operand).

Subtraction may leave you with a representation that contains leading zeroes. If leading zeroes violate your representation invariant, you'll have to do something about them.

Both addition and subtraction work from the least significant digit to the most significant digit. Array users can use `Array.foldli` or `Vector.foldli`.

The "borrowing" needed to subtract is slightly more fiddly than the "carrying" needed to add. I found it useful to define a helper function that subtracts a "borrow bit" from a single natural number.

Addition

For addition, the accumulating parameter is a carry in. The key step is to produce a sum of digits from the two addends, plus the carry in. Part of that some becomes a digit of the result, and part becomes the carry out. Code might look something like this:

```
let val s         = x_i + y_i + carry_in
    val carry_out = s div base
    val sum_i     = s mod base
in  ... do something with sum_i and carry_out ...
end
```

What happens in the ... depends on your representation, but `sum_i` becomes a digit in the output, and `carry_out` becomes the `carry_in` on the next cycle.

The final carry out should be zero. If not, you didn't allocate enough digits, and there is a bug in your code.

Subtraction

Subtraction uses the same looping structure as addition, but the computation is a little more complicated. If the difference goes negative, we have to borrow a from the next most significant digit (the "`borrow_out`").

```
let val initial_diff = x_i - y_i - borrow_in
    val (borrow_out, diff_i) =
       if initial_diff < 0 then
         (1, initial_diff + base)
       else
         (0, initial_diff)
in  ... do something with diff_i and borrow_out ...
end
```

If your final `borrow_out` is nonzero, the result of subtraction is negative, and you need to raise the `Negative` exception.

Implementing multiplication

Done carefully, multiplication is the easiest operation to implement. The equation for a product is

$$x \cdot y = \sum_{0 \le i < n} \sum_{0 \le j < m} (x_i \cdot y_j) \cdot b^{i+j}.$$

The number of digits needed to hold the double sum is potentially the total number of digits in x and y together: $m + n$. And the algorithm couldn't be simpler: it's a doubly nested loop in which each iteration adds the partial product $(x_i \cdot y_j) \cdot b^{i+j}$ to a running total. Call `shiftAdd`$(x_i \cdot y_j, i + j, \mathit{total})$. I recommend using `Array.foldli` or `Vector.foldli` on an array representation, and `foldli` or `foldri` on a list representation. (You will have to define `foldli` or `foldri` on lists.) The accumulating parameter passed to the fold is the running total—or if you are using a mutable representation, the best accumulating parameter is the empty tuple (value `()` of type `unit`).

Implementing comparison

The final operation you meet to implement compares two natural numbers and returns `EQUAL`, `LESS`, or `GREATER`. You can approach the problem the direct way or the indirect way.

- The direct approach looks at the digits. If $x = x' \cdot b + x_0$ and $y = y' \cdot b + y_0$ then you first compare the more significant digits x' and y'. If they are `EQUAL`, return `Int.compare (`x_0`, `y_0`)`. Otherwise, x and y compare the same as x' and y'.
- In the indirect approach you implement `<` by using `/-/` and seeing if it raises `Negative`. Then you use `<` to implement `compare`.

The problem with the direct approach is that depending on your representation, comparing a long number with a short number could be tricky—especially if your representation permits leading zeroes. The problem with the indirect approach is that it allocates at least one natural number, then throws it away.[2]

[2] While I find the indirect approach intellectually unsatisfying, I'm silenced by anyone who says, "the garbage collector is good at reclaiming short-lived objects; shut up and let it do its job."