Implementing Bignums in μSmalltalk

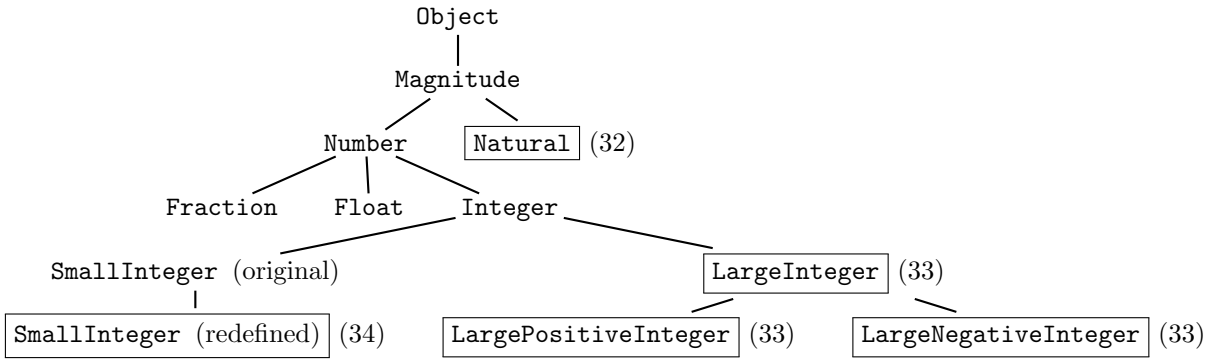
Norman Ramsey

Spring 2017

1 Approach

The pair portion of the μSmalltalk assignment is to implement arbitrary-precision arithmetic ("bignums"). You'll write a lot of methods. To help you organize them, I suggest which methods to implement in what order, and I sketch how implementations of some methods may depend on other methods.

The diagram below shows what the class hierarchy will look like once you finish. The unboxed classes are predefined μSmalltalk classes. The boxed classes are new classes you will write for this assignment. Each class that is followed by a number is from the Exercise with that number.



1.1 Big picture, part I: Natural numbers

Here is the big picture of which parts of the system do what. First, here is how you replicate the work you did on the SML assignment:

- *Comparisons* are implemented in class `Magnitude`. Or rather, four of the six comparisons are implemented in `Magnitude`. The fundamental comparisons `=` and `<` are *subclass responsibilities* (see the definition of

class `Magnitude` around page 932). You have to implement `=` and `<` on class `Natural`. The other four (`!=`, `>`, `<=`, and `>=`) are inherited from `Magnitude`.

- Addition, subtraction, multiplication, and (short) division are all implemented on class `Natural`.
  - Good news: the protocol guarantees that the argument, not just the receiver, of each of these methods has class `Natural`. No double dispatch is required.
  - Bad news: Smalltalk uses objects to hide information. Just because you know an object's class doesn't mean you have access to its representation. You will need private methods to get access to the degree and digits of the argument. The book recommends methods `degree` and `digit:`.

1.2 Big picture, part II: Large integers

As described on page 994, large integers are implemented using sign-magnitude representation. The sign is encoded in the integer's class: a large negative integer has class `LargeNegativeInteger`, and a large nonnegative integer has class `LargePositiveInteger`.[1] The magnitude, which has class `Natural` is stored in an instance variable, which I've called `magnitude`.

Both `LargeNegativeInteger` and `LargePositiveInteger` inherit from `LargeInteger`. Class `LargeInteger` is an abstract class which can define the instance variable `magnitude` and which can also define some methods that are common to both positive and negative large integers:

- Methods `=` and `<` can be implemented by subtracting the argument from `self` and examining the result to see if it is zero (or negative).
- I recommend defining a private method `isZero` which delegates to the integer's magnitude. This method will help your code work correctly with both "positive" and "negative" zero.

Most operations on large integers require double dispatch. For example, to add two large integers, you have to know the sign of both argument and receiver. A receiver knows its own sign; the sign of an argument is communicated by double dispatch. For example, method `+` on class `LargePositiveInteger` looks like this:

[1] In the perverse jargon of Smalltalk, zero is considered "positive" and positive numbers are considered "strictly positive."

```
(method + (anInteger)
     (addLargePositiveIntegerTo: anInteger self))
```

The sign of the argument is encoded in the message name `addLargePositiveIntegerTo:`, and the implementation of the method `addLargePositiveIntegerTo:` depends on the class of the receiver:

- When a *positive* large integer receives `addLargePositiveIntegerTo:`, it knows that the sum of two positive integers is positive, and it sends `withMagnitude:` to *class* `LargePositiveInteger` with the sum of the magnitudes.
- When a *negative* large integer receives `addLargePositiveIntegerTo:`, it sends `subtract:withDifference:ifNegative:` to the *argument's* magnitude, with a success continuation that produces a large positive integer and a failure continuation that produces a large negative integer.

All this is achieved without ever interrogating the class of an object, which keeps the system "open"—any object that has the right protocol will work.

Double dispatch is also used to implement multiplication, which is a little easier, because there is less case analysis—the product of two magnitudes is always a (nonnegative) magnitude.

Always write as little double dispatch as possible. For example, write the `negated` without extra dispatch—just create a new number with the same magnitude as the receiver but the opposite sign. And you can implement subtraction without writing any new dispatch at all! The default implementation, which dispatches to class `Number`, works perfectly well with large integers.

If you feel yourself not quite certain about double dispatch, you can read more about it in the book section "Inspecting multiple representations the object-oriented way: magnitudes and numbers" on page 931.

Once you have large integers working, you have a system that exemplifies the expressive power of Smalltalk: arithmetic and relational operators are implemented by messages flying around and dispatching on methods of classes `Magnitude`, `Number`, `LargeInteger`, `LargePositiveInteger`, and `LargeNegativeInteger`. Your final step is "mixed arithmetic" with large and small integers.

1.3 Big picture, part III: Mixed arithmetic

Mixed arithmetic has two goals:

- Seamlessly allow arithmetic and relational operators on a mix of small and large integers.
- Extend small-integer arithmetic so that when an intermediate result doesn't fit in a machine word, it automatically "fails over" to large-integer arithmetic.

The net result should be a system of arithmetic where you "pay as you go": if you don't need the features of large arithmetic, you're not paying extra for them, but if you do need them, they are there automatically.

You implement mixed arithmetic by applying one technique you've applied before, plus two new ones.

- The technique you've applied before is double dispatch. For example, to add a small integer to a number, you'll need a new method `addSmallIntegerTo:`, and method `+` on a small integer will dispatch to `addSmallIntegerTo:`. New method `addSmallIntegerTo:` must be defined on both large and small integers.
- The first new technique is *coercion*, which you can study in the context of classes `Integer`, `Fraction`, and `Float`. Whenever you perform an operation on mixed large and small integers, you *coerce* the small integer to a large one and repeat the operation. This form of coercion is the same regardless of the sign of the large integer, so it can go on class `LargeInteger`.
- The second new technique is to use primitives that detect overflow. For example, when adding small integers, you can no longer use primitive `+`, because it doesn't handle overflow. You'll need a different primitive that can invoke a failure continuation when addition overflows.

2 Details of class `Natural`

I suggest you implement class `Natural` in three stages, testing extensively at the end of each stage.

Stage I — Basics

1. Methods `digit:`, `digit:put:`, and `makeEmpty:` manipulate the array of digits that represent the number. Implement these methods first.

   Remember that `digit:` should work with any nonnegative argument, no matter how large.

   Arrays in μSmalltalk are 1-based, not 0-based: the first element is at position 1, not position 0. The mismatch between μSmalltalk's 1-based arrays and the 0-based abstraction you are using for polynomials is tedious. I recommend that you *hide* the 1-based nature of the underlying representation by defining and using the `digit:` and `digit:put:` methods suggested in the assignment. If you use the `digit:` method carefully, you'll have to worry about sizes only when you allocate new results.

2. Method `doDigits:` has to do with *indexes*, not with digits themselves. This way it can be used for mutation.

3. Once you have access to the digits, you can define `trim`, which removes unneeded leading zeroes.

4. Once `trim` is written, you can write `digits:`, to initialize a newly allocated bignum.

5. Now you can define the *class* method `new:`, which is your first public method—it creates a new bignum.

Stage II — Simple functions

6. Method `decimal` can be very simple if you use base $b = 10$. If you use a larger base, you will need to implement short division, just as you did on the previous assignment.

7. Once you have `decimal`, `print` is easy. Now you can debug!

8. Method `isZero` should be straightforward at this point.

9. With access to the digits, you can write `=`. You will find iteration over digits (`doDigits:`) to be helpful, and you will need to be careful when comparing bignums of different degree. (There is a simple, elegant solution to the degree problem; try to find it!)

Stage III — Arithmetic

10. The heart of your arithmetic implementation will be the two methods `set:plus:` and `set:minus:`. They depend on the digit methods above. A loop driven by `doDigits:` may be helpful.

11. You can now implement addition with method `+`. In addition to `set:plus:`, you may find it useful to use `trim` and `makeEmpty:`.

12. Subtraction is more complicated because it can fail: the difference of two natural numbers is not always a natural number. But building `subtract:withDifference:ifNegative:` on top of `set:minus:` is otherwise analogous to your construction of `+`.

13. With natural-number subtraction in hand, you can now implement the standard methods `-` and `<`. You should be able to get everything you need from `subtract:withDifference:ifNegative:`, without having to use lower-level methods of class `Natural`.

14. Multiplication is the most complicated of all. You will want to allocate a new number with `makeEmpty:` and initialize it to zero. Then, as suggested in the book, you'll need a double sum to add in all the partial products. A doubly nested `doDigits:` loop will help. To manipulate the partial products, methods `digit:` and `digit:put:` are essential. Finally, use `trim` to control the growth of your bignums.

3 Details of large integers

The book defines class `LargeInteger`, but this definition is good enough only for homogeneous arithmetic on large integers, not for mixed arithmetic on large and small integers. You will need to add methods that add to, multiply by, or compare with a small integer. Here's one example:

```
(method smallIntegerGreaterThan: (anInteger)
    (> self (asLargeInteger anInteger)))
```

You'll need similar methods for addition and multiplication.

For testing, include this `decimal` method in class `LargeInteger`:

```
(method decimal () (locals decimals)
    (set decimals (decimal magnitude))
    (ifTrue: (negative self)
       {(addFirst: decimals #-)})
    decimals)
```

You will need to have implemented the `decimal` method on class `Natural`.

Once you've gotten this far, `LargePositiveInteger` and `LargeNegativeInteger` will be relatively straightforward. The list of methods and hints given in the book should get you through. You will lean heavily on your `Natural` methods, but only the *public* methods. These are the methods of class `Magnitude`, together with the methods listed in the box on page 992.

Here are a few example methods of class `LargePositiveInteger` from my solution:

```
(method negative () false)
(method strictlyPositive () (not (isZero self)))
(method + (anInteger) (addLargePositiveIntegerTo: anInteger self))
(method addLargePositiveIntegerTo: (anInteger)
   (withMagnitude: LargePositiveInteger (+ magnitude (magnitude anInteger))))
```

You'll need a complete set of methods `negated`, `print`, `negative`, `positive`, `strictlyPositive`, `+`, `*`, `addLargePositiveIntegerTo:`, `addLargeNegativeIntegerTo:`, `multiplyByLargePositiveInteger:`, and `multiplyByLargeNegativeInteger:`. (You'll also need a `div:` method, but it can send `error`.) This design reuses the `LargeInteger` methods as much as possible.

4 Details of small integers with overflow detection

Getting mixed arithmetic to work requires a major overhaul of the `SmallInteger` class. Here are some illustrative methods:

```
(class SmallInteger SmallInteger  ; overwrite SmallInteger with new class
   ()
   (method asLargeInteger () (new: LargeInteger self))

   (method + (aNumber) (addSmallIntegerTo: aNumber self))
   (method addSmallIntegerTo: (anInteger)
      (value (addSmall:withOverflow: self anInteger
                          {(+ (asLargeInteger self) anInteger)})))
   (method addSmall:withOverflow: primitive add:withOverflow:)
   ...
```

The coercion method `asLargeInteger` enables mixed arithmetic. The three addition methods enable both mixed arithmetic (via double dispatch) and overflow detection (via primitive method, when adding two small integers).

1. You will need to replicate the addition structure for multiplication.

2. You will need to replace the primitive subtraction method with the classic "subtract from me by adding a negated argument."

3. You will need to implement `negated` using a primitive method that can detect overflow.

4. To support mixed arithmetic, you will have to implement all the methods that get dispatched when `+` or `*` is sent to a large integer: `addLargeNegativeIntegerTo:`, `addLargePositiveIntegerTo:`, `multiplyByLargeNegativeInteger:`, and `multiplyByLargePositiveInteger:`.

5. You will have to use double dispatch to implement `<`, and you will also have to replace the primitive `>`.

6. You will need to reimplement the `=` method, probably by subtracting and comparing the difference with zero. You would benefit from implementing private method `isZero` as well.