Lecture Notes on Ints

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 2
August 26, 2010

1 Introduction

Two fundamental types in almost any programming language are booleans and integers. Booleans are comparatively straightforward: they have two possible values (`true` and `false`) and conditionals to test boolean values. We will return to their properties in a later lecture.

Integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are considerably more complex, because there are infinitely many of them. Because memory is finite, only a finite subrange of them can be represented in computers. In this lecture we discuss how integers are represented, how we can deal with the limited precision in the representation, and how various operations are defined on these representations.

2 Binary Representation of Natural Numbers

For the moment, we only consider the natural numbers $0, 1, 2, \ldots$ and we do not yet consider the problems of limited precision. Number notations have a *base* b. To write down numbers in base b we need b distinct *digits*. Each digit is multiplied by an increasing power of b, starting with 2^0 at the right end. For example, in base 10 we have the ten digits 0–9 and the string `9380` represents the number $9 * 10^3 + 3 * 10^2 + 8 * 10^1 + 0 * 10^0$. We call numbers in base 10 *decimal numbers*. Unless it is clear from context that we are talking about a certain base, we use a subscript$_{[b]}$ to indicate a number in base b.

In computer systems, two bases are of particular importance. *Binary numbers* use base 2, with digits 0 and 1, and *hexadecimal numbers* (explained more below) use base 16, with digits 0–9 and A–F. Binary numbers are so important because the basic digits, 0 and 1, can be modeled inside the computer by two different voltages, usually "off" for 0 and "on" for 1. To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of a bit sequence

$$b_{n-1} \ldots b_1 b_0\ _{[2]} = b_{n-1}2^{n-1} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{n-1} b_i 2^i$$

For example, `10011`$_{[2]}$ represents $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19$.

We can also calculate the value of a binary number in a nested way, exploiting Horner's rule for evaluating polynomials.

$$10011_{[2]} = (((1 * 2 + 0) * 2 + 0) * 2 + 1) * 2 + 1 = 19$$

In general, if we have an n-bit number with bits $b_{n-1} \ldots b_0$, we can calculate

$$(\cdots((b_{n-1} * 2 + b_{n-2}) * 2 + b_{n-3}) * 2 + \cdots + b_1) * 2 + b_0$$

For example, taking the binary number `10010110`$_{[2]}$ write the digits from most significant to least significant, calculating the cumulative value from left to right by writing it top to bottom.

$$\begin{array}{rcccccr}
 & & & & 1 & = & 1 \\
1 & * & 2 & + & 0 & = & 2 \\
2 & * & 2 & + & 0 & = & 4 \\
4 & * & 2 & + & 1 & = & 9 \\
9 & * & 2 & + & 0 & = & 18 \\
18 & * & 2 & + & 1 & = & 37 \\
37 & * & 2 & + & 1 & = & 75 \\
75 & * & 2 & + & 0 & = & 150
\end{array}$$

Reversing this process allows us to convert a number into binary form. Here we start with the number and successively divide by two, calculating the remainder. At the end, the least significant bit is at the top.

For example, converting 198 to binary form would proceed as follows:

$$\begin{array}{rcrcrcr}
198 & = & 99 & * & 2 & + & 0 \\
99 & = & 49 & * & 2 & + & 1 \\
49 & = & 24 & * & 2 & + & 1 \\
24 & = & 12 & * & 2 & + & 0 \\
12 & = & 6 & * & 2 & + & 0 \\
6 & = & 3 & * & 2 & + & 0 \\
3 & = & 1 & * & 2 & + & 1 \\
1 & = & 0 & * & 2 & + & 1
\end{array}$$

We read off the answer, from bottom to top, arriving at $\texttt{11000110}_{[2]}$.

3 Modular Arithmetic

Within a computer, there is a natural size of words that can be processed by single instructions. In early computers, the word size was typically 8 bits; now it is 32 or 64. In programming languages that are relatively close to machine instructions like C or C0, this means that the native type `int` of integers is limited to the size of machine words. In C0, we decided that the values of type `int` occupy 32 bits.

This is very easy to deal with for small numbers, because the more significant digits can simply be 0. According to the formula that yields their number value, these bits do not contribute to the overall value. But we have to decide how to deal with large numbers, when operations such as addition or multiplication would yield numbers that are too big to fit into a fixed number of bits. One possibility would be to raise overflow exceptions. This is somewhat expensive (since the overflow condition must be explicitly detected), and has other negative consequences. For example, $(n+n)-n$ is no longer equal to $n+(n-n)$ because the former can overflow while the latter always yields n and does not overflow. Another possibility is to carry out arithmetic operations *modulo* the number of representable integers, which would be 2^{32} in the case of C0. We say that the machine implements *modular arithmetic*.

In higher-level languages, one would be more inclined to think of the type of `int` to be inhabited by integers of essentially unbounded size. This means that a value of this type would consist of a whole vector of machine words whose size may vary as computation proceeds. Basic operations such as addition no longer map directly onto machine instruction, but are implemented by small programs. Whether this overhead is acceptable depends on the application.

Returning to modular arithmetic, the idea is that any operation is carried out modulo 2^p for the precision p. Even when the modulus is not a power of two, many of the usual laws of arithmetic continue to hold, which makes it possible to write programs confidently without having to worry, for example, about whether to write $x + (y + z)$ or $(x + y) + z$. We have the following properties of the abstract algebraic class of *rings* which are shared between ordinary integers and integers modulo a fixed number n.

Commutativity of addition	$x + y = y + x$
Associativity of addition	$(x + y) + z = x + (y + z)$
Additive unit	$x + 0 = x$
Additive inverse	$x + (-x) = 0$
Cancellation	$-(-x) = x$
Commutativity of multiplication	$x * y = y * x$
Associativity of multiplication	$(x * y) * z = x * (y * z)$
Multiplicative unit	$x * 1 = x$
Distributivity	$x * (y + z) = x * y + x * z$
Annihilation	$x * 0 = 0$

Some of these laws, such as associativity and distributivity, do *not* hold for so-called *floating point* numbers that approximate real numbers. This significantly complicates the task of reasoning about programs with floating point numbers which we have therefore omitted from C0.

4 An Algorithm for Binary Addition

In the examples, we use arithmetic modulo 2^4, with 4-bit numbers. Addition proceeds from right to left, adding binary digits modulo 2, and using a carry if the result is 2 or greater. For example,

$$\begin{array}{cccccccl} & 1 & 0 & 1 & 1 & = & 11 & \\ + & 1 & 0_1 & 0_1 & 1 & = & 9 & \\ \hline (1) & 0 & 1 & 0 & 0 & = & 20 & = 4 \pmod{16} \end{array}$$

where we used a subscript to indicate a carry from the right. The final carry, shown in parentheses, is ignored, yielding the answer of 4 which is correct modulo 16.

This grade-school algorithm is quite easy to implement (see Section 8) in software, but it is not suitable for a hardware implementation because it is too sequential. On 32 bit numbers the algorithm would go through 32 stages, for an operation which, ideally, we should be able to perform in one machine cycle. Modern hardware accomplishes this by using an algorithm where more of the work can be done in parallel.

5 Two's Complement Representation

So far, we have concentrated on the representation of natural numbers $0, 1, 2, \ldots$. In practice, of course, we would like to program with negative numbers. How do we define negative numbers? We define negative numbers as additive inverses: *$-x$ is the number y such that $x + y = 0$.* A crucial observation is that in modular arithmetic, additive inverses already exist! For example, $-1 = 15 \pmod{16}$ because $-1 + 16 = 15$. And $1 + 15 = 16 = 0 \pmod{16}$, so, indeed, 15 is the additive inverse of 1 modulo 16.

Similarly, $-2 = 14 \pmod{16}$, $-3 = 13 \pmod{16}$, etc. Writing out the equivalence classes of numbers modulo 16 together with their binary representation, we have

...	−16	0	16	...	0000
...	−15	1	17	...	0001
...	−14	2	18	...	0010
...	−13	3	19	...	0011
...	−12	4	20	...	0100
...	−11	5	21	...	0101
...	−10	6	22	...	0110
...	−9	7	23	...	0111
...	−8	8	24	...	1000
...	−7	9	25	...	1001
...	−6	10	26	...	1010
...	−5	11	27	...	1011
...	−4	12	28	...	1100
...	−3	13	29	...	1101
...	−2	14	30	...	1110
...	−1	15	31	...	1111

At this point we just have to decide which numbers we interpret as positive and which as negative. We would like to have an equal number of positive and negative numbers, where we include 0 among the positive ones. From this considerations we can see that $0, \ldots, 7$ should be positive and

$-8, \ldots, -1$ should be negative and that the highest bit of the 4-bit binary representation tells us if the number is positive or negative.

Just for verification, let's check that $7 + (-7) = 0 \pmod{16}$:

$$\begin{array}{ccccc} & 0 & 1 & 1 & 1 \\ + & 1_1 & 0_1 & 0_1 & 1 \\ \hline (1) & 0 & 0 & 0 & 0 \end{array}$$

It is easy to see that we can obtain $-x$ from x on the bit representation by first complementing all the bits and then adding 1. In fact, the addition of x with its bitwise complement (written $\sim x$) always consists of all 1's, because in each position we have a 0 and a 1, and no carries at all. Adding one to the number $11 \ldots 11$ will always result in $00 \ldots 00$, with a final carry of 1 that is ignored.

These considerations also show that, regardless of the number of bits, -1 is always represented as a string of 1's.

In 4-bit numbers, the maximal positive number is 7 and the minimal negative numbers is -8, thus spanning a range of $16 = 2^4$ numbers. In general, in a representation with p bits, the positive numbers go from 0 to $2^{p-1} - 1$ and the negative numbers from -2^{p-1} to -1. It is remarkable that because of the origin of this representation in modular arithmetic, the "usual" bit-level algorithms for addition and multiplication can ignore that some numbers are interpreted as positive and others as negative and still yield the correct answer modulo 2^p.

However, for comparisons, division, and modulus operations the sign does matter. We discuss division below in Section 9. For comparisons, we just have to properly take into account the highest bit because, say, $-1 = 15 \pmod{16}$, but $-1 < 0$ and $0 < 15$.

6 Hexadecimal Notation

In C0, we use 32 bit integers. Writing these numbers out in decimal notation is certainly feasible, but sometimes awkward since the bit pattern of the representation is not easy to discern. Binary notation is rather expansive (using 32 bits for one number) and therefore difficult to work with. A good compromise is found in *hexadecimal notation*, which is a representation in base 16 with the sixteen digits 0–9 and A–F. "Hexadecimal" is often abbreviated as "hex". In the concrete syntax of C0 and C, hexadecimal numbers are preceded by `0x` in order to distinguish them from decimal numbers.

binary	hex	decimal
0000	`0x0`	0
0001	`0x1`	1
0010	`0x2`	2
0011	`0x3`	3
0100	`0x4`	4
0101	`0x5`	5
0110	`0x6`	6
0111	`0x7`	7
1000	`0x8`	8
1001	`0x9`	9
1010	`0xA`	10
1011	`0xB`	11
1100	`0xC`	12
1101	`0xD`	13
1110	`0xE`	14
1111	`0xF`	15

Hexadecimal notation is convenient because most common word sizes (8 bits, 16 bits, 32 bits, and 64 bits) are multiples of 4. For example, a 32 bit number can be represent by eight hexadecimal digits. We can even do a limited amount on arithmetic on them, once we get used to calculating modulo 16. Mostly, though, we use hexadecimal notation when we use bitwise operations rather than arithmetic operations.

7 Bitwise Operations on Ints

Ints are also used to represent other kinds of data. An example, explored in the first programming assignment, is colors (see Section 11). The so-called ARGB color model divides an int into four 8-bit quantities. The highest 8 bits represent the opaqueness of the color against its background, while the lower 24 bits represent the intensity of the red, green and blue components of a color. Manipulating this representation with addition and multiplication is quite unnatural; instead we usually use bitwise operations.

The bitwise operations are defined by their action on a single bit and then applied in parallel to a whole word. The tables below define the meaning of *bitwise and* `&`, *bitwise exclusive or* `^` and *bitwise or* `|`. We also have *bitwise negation* `~` as a unary operation.

And

&	0	1
0	0	0
1	0	1

Exclusive Or

^	0	1
0	0	1
1	1	0

Or

|	0	1
0	0	1
1	1	1

Negation

~	0	1
	1	0

8 Implementing Addition

We now give an example of C0 programming with arrays of bits by implementing addition on numbers of arbitrary precision using bitwise operations.

We start by defining a new type name `bit` as an alias for the type `int`. The intent is for us to write `bit` if we know that the int will be either `0` or `1`. The syntax for this construct in general is `typedef t a;` where t is an arbitrary type and a is a new name for t that acts as an abbreviation. The type-checker always expands a as t during type-checking. Technically, the definition is called *transparent*.

```
typedef int bit;  /* bit: 0 or 1 */

bool is_bit(int n) {
  return (n == 0 || n == 1);
}
```

We also define a boolean function to test if an integer represents a bit. We intend to use this function in contracts, that is, pre- and postconditions for functions as well as loop invariants.

The main type of interest is that of arrays of bits of length n. We define the function `is_num` to return true if the given array represents a bit string of precision n.

```
/* is_num(x,n) == true iff x is a bitstring of precision n */
bool is_num (bit[] x, int n)
//@requires n == \length(x);
{
  int i;
  for (i = 0; i < n; i++)
    if (!is_bit(x[i])) return false;
  return true;
}
```

A few notes on C0. The type of arrays with element of type t is written as `t[]`, so `bit[] x` is the declaration of a variable x as an array of bits. This is synonymous with an array of ints, except that writing `bit[]` expresses our intent that all elements be 0 or 1. Because `bit[]` and `int[]` are synonymous, the compiler will not flag it as an error if you try to store a value such as -1 or 2 in an array of type `bit[]`, so your contract annotations and loop invariants should explicitly verify this, using predicates such as `is_num`.

Arrays are allocated with `alloc_array(t,e)` where t is the element type, and the expression e denotes the number of elements in the array, which must be positive.[1] We access array elements by writing $A[e]$ where A denotes an array and e is an expression that evaluates to the index i. If i is less than zero or greater or equal to the length of the array, an exception is raised and the program terminated.

In our example, to guarantee that x is of the proper length n, we stipulate that the `n == \length(x)` as a precondition. The expression `\length(e)` can only be used in contracts. The reason is that in C0 there is no way to determine the length of an array at runtime from within a program, a property inherited from C. From the modern perspective, the decision by the designers of C not to endow an array with its length is difficult to understand. The lack of concern for memory safety (for example, allowing array accesses out of bounds) has led to a myriad of problems, including so-called buffer overflow attacks which to this day plague software and operating systems. On the other hand, it is possible to program low-level operating system code, dealing with specifics of the machine and hardware, which has been difficult to achieve in higher-level languages.

In C0, we take a more modern position. In the usual operation of C0, array bounds and other potentially unsafe memory access are being checked at runtime, just as in more advanced languages such as Java or ML. But we can also emulate C by compiling C0 code in unsafe mode, which may generate a more efficient executable. For the purposes of this class, the low-level marginal speed gain is not of particular interest, so we do not use unsafe mode. We envision that in future implementations of C0, theorem proving may be used in conjunction with function contracts and loop invariant to *safely* remove bounds checks where they can be shown to be redundant, giving us the best of both worlds.

[1]In this course, when we say *positive* we include zero. Describing the numbers $0, 1, \ldots$ as being "non-negative" is like buying "extra-strength non-aspirin" in the drug store: it doesn't tell you at all what it is, only what it isn't. If we need to refer to the numbers $1, 2, 3, \ldots$ we call them *strictly positive*. Also, in this course, *natural numbers* start at 0, not 1 (as is the custom in some branches of mathematics).

Next, we develop the code for binary addition on bit arrays. We require that the two arguments x and y are of the same precision n. We guarantee that the resulting sum is also of the same precision. The program loops over the two input arrays in unison, starting with the least significant bit. We use the bit-level operation of *exclusive or* to compute the sum of three bits (the two inputs and the carry) and bitwise *and* and *or* to compute the new carry. The computation of the new carry could be slightly shortened using distributivity laws, but the one below is perhaps clearest.

```
/* Binary addition mod 2^n */
bit[] plus(bit[] x, bit[] y, int n)
//@requires is_num(x,n) && is_num(y,n);
//@ensures is_num(\result,n);
{ int i;
  bit[] result = alloc_array(bit, n);
  bit carry = 0;
  for (i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant is_bit(carry);
    //@loop_invariant is_num(result,n);
   {
      result[i] = (x[i] ^ y[i]) ^ carry;
      carry = (x[i] & y[i])
            | (x[i] & carry)
            | (y[i] & carry);
    }
  // last carry is overflow bit; ignore
  return result;
}
```

We state three loop invariants. The first,

```
    //@loop_invariant 0 <= i && i <= n;
```

guarantees that the array index is in range. It is important to remember that the loop invariant is tested *just before* the exit condition. Here, this means after the increment of i and before checking $i < n$. Therefore, the invariant must include the case $i \leq n$. If we proceed to the loop body, we know that the test must have evaluated to `true`, so $i < n$ and the array access will be in bounds.

The second `is_bit(carry)` just checks that the variable $carry$ is either 0 or 1. The third,

```
    //@loop_invariant is_num(result,n);
```

checks that the (partially computed) result is a bit array of the proper length. This holds initially because the bit array is initialized with zeroes; it continues to hold through the loop because the operations of *and* (`&`), *or* (`|`), and *exlusive or* (`^`) return 0 or 1 when applied to ints that are 0 or 1.

There is no further specification of addition. This is an example where, in a sense, the function serves as its own specification. It would be difficult to think of a more primitive specification that we could use to define the meaning of addition, except in some special cases. What we can sometimes do in more advanced language that integrate specifications (for example, Agda) is to then prove a variety of expected properties of the function. In this case, we might prove commutativity, associativity, the identity property of zero, etc.

9 Integer Division and Modulus

The division and modulus operators on integers are somewhat special. As a multiplicative inverse, division is not always defined, so we adopt a different definition. We write x/y for *integer division* of x by y and $x\%y$ for *integer modulus*. The two operations must satisfy the property

$$(x/y) * y + x\%y = x$$

so that $x\%y$ is like the remainder of division. The above is not yet sufficient to define the two operations. In addition we say $0 \leq |x\%y| < y$. Still, this leaves open the possibility that the modulus is positive or negative when y does not divide x. We fix this by stipulating that integer division truncates its result towards zero. This means that the modulus must be negative if x is negative and there is a remainder, and it must be positive if x is positive.

By contrast, the *quotient* operation always truncates down (towards $-\infty$), which means that the *remainder* is always positive. There are no primitive operators for quotient and remainder, but they can be implemented with the ones at hand.

10 Shifts

We also have some hybrid operators on ints, somewhere between bit-level and arithmetic. These are the *shift* operators. We write `x << k` for the result of shifting x by k bits to the left, and `x >> k` for the result of shifting x by

k bits to the right. In both cases, k is masked to 5 bits before the operation is applied, which can be written as `k & 0x0000001F`, so that the actual shift quantity s has the property $0 \leq s < 32$. We assume below that k is in that range.

The left shift, `x << k` (for $0 \leq k < 32$), fills the result with zeroes on the right, so that bits $0, \ldots, k-1$ will be 0. Every left shift corresponds to a multiplication by 2 so `x << k` returns $x * 2^k$ (modulo 2^{32}).

The right shift, `x >> k` (for $0 \leq k < 32$), copies the highest bit while shifting to the right, so that bits $31, \ldots, 32-k$ of the result will be equal to the highest bit of x. If viewing x a an integer, this means that the sign of the result is equal to the sign of x, and shifting x right by k bits correspond to integer division by 2^k except that it truncates towards $-\infty$. For example, `-1 >> 1 == -1`.

11 Representing Colors

As a small example of using the bitwise interpretation of ints, we consider colors. Colors are decomposed into their primary components red, green, and blue; the intensity of each uses 8 bits and therefore varies between 0 and 255 (or `0x00` and `0xFF`). We also have the so-called *α-channel* which indicates how opaque the color is when superimposed over its background. Here, `0xFF` indicates completely opaque, and `0x00` completely transparent.

For example, to extract the intensity of the red color in a given pixel p, we could compute `(p >> 16) & 0xFF`. The first shift moves the red color value into the bits 0–7; the bitwise and masks out all the other bits by setting them to 0. The result will always be in the desired range, from 0–255.

Conversely, if we want to set the intensity of green of the pixel p to the value of g (assuming we already have $0 \leq g \leq 255$), we can compute `(p & 0xFFFF00FF) | (g << 8)`. This works by first setting the green intensity to 0, while keep everything else the same, and then combining it with the value of g, shifted to the right position in the word.

For more on color values and some examples, see Assignment 1.