Data Project: Worked Examples

Fall 2012

The following examples are intended to show how you might go about solving the puzzles of the Data Project.

1 Integer Example

Here is an example of an integer puzzle. In this example, we will assume that you have solved the `negate()` and `isEqual()` puzzles.

```
/*
 * fitsBits - Return 1 if x can be represented as an
 *   n-bit two's complement integer, else return 0
 *   1 <= n <= 32
 *   Examples: fitsBits(5, 3) = 0
 *             fitsBits(-4,3) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int fitsBits(int x, int n) {
    return 2;
}
```

For this puzzle, we want to tell if a number fits in n bits. What do we know about numbers which fit inside of n bits?

- For such a number x, $-2^{n-1} \leq x < 2^{n-1}$.
- Each bit from the n^{th} upward is the same. (Remember that we can extend a signed n-bit integer to an m-bit integer by copying the sign bit into the upper $m - n$ bits.)
- If we truncate the number to n bits, the result should represent the same number.

The first approach is difficult to do given the restrictions of this assignment. The second or third approach might be doable with bit level operations; let's try the third approach.

How can we truncate a 32-bit number to n bits? We need an operation which can cut off the upper bits of `x`. Fortunately, such an operation exists! A simple left shift by $32 - n$ will suffice. If `x` cannot fit within n bits, then truncating it will change the value of the represented number; if it can, then truncating it will not change the value.

Next we need a way to check that the result is the same as the original value of `x`. But since the truncated value is shifted, the bits do not align. We need to find a way to return the bits to their original position.

What if we right shift back by the same amount? Doing so will place the bits in the right place, and will sign-extend the truncated value back to 32 bits. This is exactly what we need.

Now that the bits are in the correct place, all that is needed is a way to check equality. Fortunately, we can employ the same operations used from `isEqual()`.

Putting this together, we get the following solution:

```
int fitsBits(int x, int n) {
    int shift = 32 + negate(n);
    int truncated = (x << shift) >> shift;
    return isEqual(x, truncated);
}
```

Note that any solutions you write may not make any function calls, but should instead re-use the same bit-level operations.

2 Floating Point Example

Here is an example of a floating-point puzzle.

```
/*
 * float_twice - Return bit-level equivalent of expression 2*f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned ints, but
 *   they are to be interpreted as the bit-level representation of
 *   32-bit floating point values.
 *   When argument is NaN, return argument.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_twice(unsigned uf) {
    return 2;
}
```

The object of this problem is to multiply an arbitrary floating point number by 2. How should we go about solving this?

When dealing with floating-point representations, one approach is to first extract the sign, exponent, and mantissa from the input value. We can then modify these values as appropriate, and recombine them at the end to produce the result.

To extract these components, we can use a combination of shifts and "masks", bit strings that represent the bits we are interested in. For instance, to get the exponent of a 32-bit floating-point value `uf`, we can take `(uf >> 23) & 0xFF`. The right shift removes the lower 23 bits, which we don't care about, and the `&` operation zeros all bits except the lower eight (corresponding to the

bits of the mask `0xFF`). Alternatively, we could extract the exponent as `(uf & 0x7F800000) >> 23`, first extracting the relevant bits and then shifting the result into place.

Once we have extracted the sign, exponent, and mantissa, we have three types of value to consider: NaN (including infinity), denormalized values, and normalized values. We can test for these cases by checking if the exponent is 127 (NaN), 0 (denormalized), or something else (normalized).

For this problem, we must address each case separately.

- If the input is NaN, we can return the result without modification.
- If the input is denormalized, we know that the mantissa represents the bits after the decimal point of the value, and that the implied exponent is -126. For instance, if the mantissa is $00100000111110000011111_2$, the value represented is $0.00100000111110000011111_2 \times 2^{-126}$. To multiply this value by 2, we can shift the mantissa to the left by 1. We must also make sure to handle "overflow" from denormalized to normalized representations.
- If the input is normalized, we simply need to add 1 to the exponent to multiply the represented value by 2. (*Why?*) If we overflow past the range of representable values, we should return +/- infinity by setting the mantissa to 0.

Once we have modified the value as appropriate, we can re-combine the sign, exponent, and mantissa with a sequence of shifts and bitwise-or operations.

Putting all of the above together, we get the following solution:

```
unsigned float_twice(unsigned uf) {

    // Extract sign, exponent, and mantissa
    unsigned sign = uf >> 31;
    unsigned exponent = (uf >> 23) & 0xFF;
    unsigned mantissa = uf & 0x7FFFFF;

    if(exponent == 127) {  //NaN
        // Do nothing
    } else if(exponent == 0) {  // Denormalized
        mantissa <<= 1;
        if(mantissa > 0x7FFFFF) {  // Result should be normalized
            mantissa = mantissa & 0x7FFFFF;  // Chop off leading bit
            exponent = 1;
        }
    } else {  // Normalized
        exponent++;
        if(exponent == 0xFF) {  // Infinity
            mantissa = 0;
        }
    }

    return (sign << 31) | (exponent << 23) | mantissa;
}
```