3EA3 Pellets Report

James Lee - 1318125

April 13, 2017

1 Abstract

This report deals with one of Google's foo.bar questions, where you are given a positive starting number of items or "pellets" as a string, and you are asked to return the minimum number of operations required in order to transform the number of pellets to 1. There are only three operations permitted, and they are as follows:

1. Add one item.
2. Remove one item.
3. Divide the number of items by 2 (only allowed if there is an even number of items).

The following report details my personal solution to this problem. It was originally done in Python (code can be found in the appendix), however it was translated into C in order to implement ACSL specifications and prove correctness using Frama-C. As to what Frama-C is, it is just a source code analyzer and prover of C software. More information can be found at https://frama-c.com/.

2 Structure

The structure of the report will mainly consist of program functions listed one by one with their corresponding ACSL specifications, along with a short explanation of what each function does, and how the specifications verify the correctness of the functions.

3 Implementation and ACSL

We will first begin with the two Math functions, `power()` and `intLog()`. These functions needed to be created as the standard Math library functions cannot be called in the specifications.

3.1 power()

```
/*@ axiomatic power {
  @ logic integer pow(integer x, integer y);
  @ axiom base_pow: \forall integer x;
  @     pow (x, 0) == 1;
  @ axiom unfold_pow: \forall integer x, integer y;
  @     pow (x, y) == x * pow(x, y-1);
  @ }
*/
/*@
    requires x > 0;
    requires y >= 0;
    ensures \result == x * pow(x, y-1);
```

```
*/
int power(int x, int y)
{
        if (y == 0) return 1;
        else return x*power(x, y-1);
}
```

This power function is just a simple implementation that takes in integer x and integer y, and recursively returns the value x to the power of y. It requires that `x` $\geq$ `0` and `y` $>$ `0`. y is not allowed to be 0 to avoid the case where x and y are both 0, resulting in a math error. It is relatively straightforward, just multiplying x at each level with the recursive call of `pow(x, y-1)`. At the base case `pow(x, 0)` where y is 0, it just returns 1. The correctness of this function is verified by the fact that x to the power of y can also be represented as x multiplied by x, y times.

3.2 intLog()

```
/*@ axiomatic intLog {
  @ logic integer log(integer x, integer y);
  @ axiom base_log: \forall integer x, integer y: x < y;
  @     log (x, y) == 0;
  @ axiom unfold_log: \forall integer x, integer y;
  @     log (x, y) == 1 + log(x/y, y);
  @ }
*/
/*@
    requires x >= 0;
    requires y > 0;
    ensures \result == 1 + log(x, y);
*/
int intLog (int x, int y)
{
        if (x < y) return 0;
        else return 1 + intLog(x/y, y);
}
```

This function returns the integer logarithm of x, under the base of y. For example, `intLog(8, 2)` would return the base 2 logarithm of 8. Again, this function is straightforward, as the base case returns 0 if x is less than y, and the unfolding recursively calls log while dividing x by y. Since the logarithm of a number is just finding the largest power of the base that goes into that number, this function is correct as x is divided by y at each stage, simulating division by powers of the base. For the purposes of this program, we only need the integer logarithm as the decimals after do not matter, for reasons explained in the next function.

3.3 longToBin()

This is another one of the helper functions in the program. It takes in a long number n, and the integer pointer count, and returns the number n converted into it's binary representation, as a character array. ACSL specifications ensure that n is initially greater than or equal to 0, as well as making sure that count is non negative. The ensures clause performs the following: for each index i, it goes through `\result` and multiplies the value of `\result[i]` by `pow(2, i)`, and sums all these values together. By ensuring that the resulting sum is equal to n, then that means that `longToBin()` is able to successfully represent n as a character array.

An interesting interaction we can notice here is how we can multiply `\result` which is a character with an integer and still get the correct result, without having to cast the character to an integer first. This is because of the integer values of the characters '0' and '1'. The decimal value of '0' in ASCII is 48, and '1'

is 49. When multiplying 48 by a power of 2, the resulting hexadecimal value will end up with the last two places being '00', and since `char` is 1 byte, it will only retain the last two places resulting in an integer value of 0.

```
/*@
        requires n >= 0;
        requires *count >= 0;
        ensures n == \sum(0, *count, \lambda integer i;
            \result[i] * pow(2, i));
*/
// function that converts from long to binary
char* longToBin(long n, int * count)
{
        char* array = (char*)malloc(1);

        long current = n;
        //@ assert current == n;

        // minimum number of bits needed plus an
        // extra 0 (for addition)
        // intLog returns floor, add 1 for ceil
        *count = intLog(n, 2)+1+1;
        //@ assert *count == log(n, 2)+2;

        // add the end of string character
        array[*count+1] = '\0';
        //@ assert array[*count+1] == '\0';

        // start at index count and go down
        /*@ loop invariant i > -1; */
        for (int i = *count; i > -1; i--){
                if (current - power(2, i) >= 0){
                        array[*count-i] = '1';
                        current -= power(2, i);
                        //@ assert array[*count-i] == '1';
                }
                else{
                        array[*count-i] = '0';
                        //@ assert array[*count-i] == '0';
                }
        }
        return array;
}
```

First of all, this function works by calculating the minimum number of bits required to represent the number, by using the `intLog()` function with base 2. Since intLog() returns the integer floor, we add 1 to simulate the ceiling, and another 1 because we want an extra bit in case addition is necessary later. Thus, we assert that `*count == log(n, 2) + 2`.

After adding an end of string character to the end of the character array, we can then start at the index of `count` and move backwards, calculating the binary representation of the number n. The loop keeps running as long as $i >$ -1, meaning that it will iterate through the entire character array, from the index `count` to the index 0. If the value of `current` is greater than or equal to the value returned by `power(2, i)`, then

`current` is subtracted by `power(2, i)` and the character at index i is set as a '1'. Otherwise, no subtraction is done and the character at index i is set as a '0'.

An example of this function at work would be if we input n as 30. `intLog(30, 2)` would return 4, then `*count = 4+1+1 = 6`. During the loop, i is initialized as 6. `current` is initially initialized as n, and 30 is less than `power(2, 6)`, which is 64. `array[6]` is thus set as '0', and the same is done for `array[5]`, as 30 is still less than `power(2, 5)`. Once i becomes 4, `current - power(2, 4)` $\geq$ `0`, thus `current` is subtracted by 16 and `array[4]` is set as '1'. By repeating this process, and stopping at i = -1, we end with array being '0011110', and this result is returned.

3.4 reduce()

We will first show the `reduce()` function in its entirety, and then break it down in smaller sections.

```
/*@
        requires n >= 0;
        ensures \result == 1 || 0 <= \result <= log(n, 2);
*/
// calculates the number of steps it takes to get to 1
int reduce(long n){
        long input = n;
        //@ assert input == n;
        int numOps = 0;
        //@ assert numOps == 0;
        char *pointer;
        int index = 0;
        //@ assert index == 0;
        // get the binary representation of n
        pointer = longToBin(input, &index);

        printf("Binary string is: ");
        printf("%s \n", &pointer[0]);

        // if binRep = 0 return 1 op
        if (pointer[0] == '0' && index == -1){
                //@ assert index == -1 && pointer[0] == '0';
                numOps = 1;
                return numOps;
        }

        //@ assert index >= 0;

        /*@ loop invariant index >= 0 && numOps >= 0; */
        // while we are not at the second last element
        // (we end at '01')
        while (index > 1 || numOps < 0){
                //@ assert index > 1;

                // if number is 3 then it takes 2 operations
                if (pointer[index-2] == '0'
                    && pointer[index-1] == '1'
                    && pointer[index] == '1' && index == 3){
                        numOps += 2;
                        index -= 1;
                        //@ assert numOps > 0;
```

```
}
// If it ends as 001, then it just equals 1
else if (pointer[index-2] == '0'
    && pointer[index-1] == '0'
    && pointer[index] == '1'&& index == 2){
        index -= 1;
        //@ assert numOps >= 0;
}
// if ends in 0 then cut it (dividing by 2)
else if (pointer[index] == '0'){
        index -= 1;
        numOps += 1;
        //@ assert numOps > 0;
}
// if ends in 01 then subtract then cut
else if (pointer[index-1] == '0'
    && pointer[index] == '1'){
        index -= 1;
        numOps += 2;
        //@ assert numOps > 0;
}
// if ends in 11 then add then cut
else if (pointer[index-1] == '1'
    && pointer[index] == '1'
    && numOps >= 0){
        //@ assert numOps >= 0;

        /* Addition is pretty much turning
         the closest 0 into a 1 and changing
         all the intermediate 1's into 0. Find
         closest 0, can start from 2 since we
         know it already ends in '11' */
        int closest = 2;
        //@ assert index >= closest;

        /*@ loop invariant closest >= 2
            && index >= closest
            && numOps >= 0; */
        while (index < closest
            || pointer[index-closest] != '0'){
                //@ assert index >= closest;
                if (index > closest)
                        //@ assert
                            index > closest;
                        closest ++;
                //@ assert index >= closest;
        }
        //@ assert closest >= 2
            && index >= closest && numOps >= 0;

        // change it to a 1
        pointer[index-closest] = '1';
        // can cut all of the numbers and
        // add ops equal to numbers cut
```

```
                        index -= closest;
                        numOps += 1+closest;
                        //@ assert numOps > 0;
                }
        }
        //@ assert index >= 0 && numOps >= 0;
        //free(pointer);
        return numOps;
}
```

We will first take a look at the first section, which includes the function definition as well as basic initialization of variables and print statements.

```
/*@
        requires n >= 0;
        ensures \result == 1 || 0 <= \result <= log(n, 2);
*/
int reduce(long n){
        long input = n;
        int numOps = 0;
        char *pointer;
        int index = 0;
        pointer = longToBin(input, &index);

        printf("Binary string is: ");
        printf("%s \n", &pointer[0]);

        if (pointer[0] == '0' && index == -1){
                numOps = 1;
                return numOps;
        }
```

`reduce()` takes in a long number n, and returns an integer representing the minimum number of operations required to reduce that number to 1. It requires that the number n is initially greater than or equal to 0. The result is ensured to either be 1 (in the case where $n = 0$) or in the range between 0 and `log(n, 2)`. The minimum is when $n = 1$, resulting in `numOps` being 0, and the maximum is when n is a power of 2, such as 2, 4, 8, etc. First, the variables are initialized, with `input = n`, `numOps = 0`, `index = 0`, and `*pointer = longToBin(input, &index)`. The `longToBin()` function as explained earlier takes in a number n, which is in this case `input`, and an integer pointer `*count`, which is in this case `index`. The character array `*pointer` ends up as the binary representation of the number n, and `index` is set to be the minimum number of bits necessary to represent this number.

The next two lines just print out the binary string, allowing the user to visually confirm that the value of the string is equal to the number they inputted. After this, there is just a quick check, returning the number of operations to be 1 if the number inputted was 0.

```
        /*@ loop invariant index >= 0 && numOps >= 0; */
        while (index > 1 || numOps < 0){
                if (pointer[index-2] == '0'
                    && pointer[index-1] == '1'
                    && pointer[index] == '1' && index == 3){
                        numOps += 2;
                        index -= 1;
```

```
}
else if (pointer[index-2] == '0'
    && pointer[index-1] == '0'
    && pointer[index] == '1'&& index == 2){
        index -= 1;
}
else if (pointer[index] == '0'){
        index -= 1;
        numOps += 1;
}
```

Next we look at the first half of the while loop. This loop has invariants that `index` $\geq$ `0` and `numOps` $\geq$ `0`. This is correct as the loop should continue to run as long as we are not at the second last element (or as long as `index` $>$ `1`), as '01' means that the number at 1 and that is the number that we want to arrive at.

The first if is a special case. If the number is 3, then we decrement `index` by 1 and increase `numOps` by 2. In binary, the representation would be '11', so decreasing `index` by 1 is the same as subtracting by 2 in this case. We also need to check that `index` is currently at 3, so we avoid situations where numbers like '1111' from falling into this case.

The second part of the if occurs very rarely, but when it does happen, we just decrease `index` by 1 and don't do anything to `numOps`. Decrementing `index` for '001' would just result in '01' and is 1. It may seem that this function does not serve any purpose, but remember that we need to decrement `index` or else the loop will never end.

The last else if in this first segment is one of the main cases that will occur frequently. We take advantage of the fact that binary numbers have the trait that a right shift is essentially dividing the number by 2. Examples include '100' = 4 becoming '10' = 2 after a right shift, and '1100' = 12 becoming '110' = 6. Therefore, if the character array ends in '0', we decrement `index` by 1 and increase the number of operations by 1 as well.

```
else if (pointer[index-1] == '0'
    && pointer[index] == '1'){
        index -= 1;
        numOps += 2;
}
else if (pointer[index-1] == '1'
    && pointer[index] == '1'
    && numOps >= 0){
        int closest = 2;

        /*@ loop invariant closest >= 2
            && index >= closest
            && numOps >= 0; */
        while (index < closest
            || pointer[index-closest] != '0'){
                if (index > closest)
                        closest ++;
        }

        pointer[index-closest] = '1';
        index -= closest;
```

```
                        numOps += 1+closest;
                }
        }
        //free(pointer);
        return numOps;
}
```

If the character array ends in '01', then we can first subtract 1, turning it into '00', and then cut the last element off, making it '0'. We do this by decreasing `index` by 1 and increasing `numOps` by 2. For example, if the number is '101' = 5, we subtract 1 making it '100' = 4, then "divide by 2" making it '10' = 2.

The next case is efficient but also requires a `while` loop of its own. It handles the case where the character array ends in '11'. In this case, we want to add 1 instead of subtracting, as it is in fact more efficient to add as it generates more consecutive '0's. For example, if we had '01111' = 15, instead of subtracting and then cutting the 0 each time, which would result in `numOps` being 6 (subtract, then divide, each repeated 3 times), if we add 1 first, it would become '10000' = 16, allowing us to get a final result of `numOps` being 5 (add, then divide 4 times).

After adding 1, we know that there are at least 2 '0's in a row. We look for the maximum number of consecutive 0's so we can cut them all at once. This `while` loop has invariants such that `closest` $\geq$ `2`, `index` $\geq$ `closest` and `numOps` $\geq$ `0` at all times. As stated earlier, we want to find the largest number of consecutive '0's, so while the current character at the index is '0', we continue iterating through the loop. Notice that we don't actually perform any addition, we merely change the closest non '0' into a '1', which simulated addition. After this, we cut off all the consecutive '0's by subtracting `index` by `closest`, and add operations equal to `1 + closest` (one addition operation plus 'closest' number of division operations).

The end of the `reduce()` function just has 2 lines, a commented out `free(pointer)` line, and the `return` line. To my knowledge, Frama-C does not have any specifications for freeing memory after use, so this line was commented out as it could not be proven.

3.5 main()

```
int main()
{
        long n;
        while(1){
                printf("Enter a positive long integer: ");
                scanf("%ld",&n);
                if (n >= 0){
                        printf("Number of Operations: %i\n", reduce(n));
                }
        }
        return 0;
}
```

This is just the `main()` function that runs upon program start through the command line, and takes in a long integer as input, passes it to the function `reduce()`, and returns the number of operations required to transform the given number to 1. This is run in a never ending `while` loop. There are no ACSL specifications for this function.

4 Conclusions

4.1 Challenges

As with most projects, there were a few hurdles that had to be overcome in order to complete this assignment. The first hurdle was deciding how to represent and convert the data. In Python, through the use of built in functions that converted between binary and decimal, the total implementation was very short (only about 20 lines!). However, in C, I had to make my own function to do this. Another hurdle was with memory allocation and freeing of this memory. I stored the number as a binary string in a character array, and Frama-C to my knowledge does not have nice specifications towards the allocating and freeing of memory. In the end, the free memory line was just commented out as it would not turn green to indicate proven specifications. One more challenge was creating my own math functions. The Math functions in the Math library could not be used in the ACSL specifications themselves, so in order to use math operations (such as power or log) in the ACSL specifications I actually had to write my own functions and their corresponding axioms.

4.2 Future Work

Further improvements could include adding better defined specifications for the `reduce()` function, and cleaning up some specifications for other functions such as `intLog()`. There are some fringe cases where Frama-C returns a green or correct result, even though it should not be correct. One example is `intLog(0, 1)`, which would return 0 from the code, even though the ensures clause would expect a return value of 1. Another example is multiplying a character by an integer in the function `longToBin()`. Up to the value 256, the cutoff would be correct and it would return the correct value, but for numbers larger than 256 it should result in the wrong cutoff and therefore a wrong value. As with the previous example, Frama-C also returns a green, or correct specification.

In the future when Frama-C receives some additional functionality maybe the `reduce()` function can see some stronger specifications. Wrong fringe cases might also be fixed, forcing the user to analyze their code in more depth as well. For now however, Frama-C was still very helpful in generating correct specifications for this program.

4.3 Final Remarks

Adding ACSL specifications to functions and statements caused me to realize that my guards or invariants were not strong enough to cover branch cases where my program would fail or be incorrect. In conclusion, I can safely say that this project was very worthwhile, and will most likely help me create programs that have proper specifications and are correct by construction.

5 Appendix

The source code for the Python implementation.

```
def answer(n):
    """ Convert the string to binary """
    binRep = bin(int(n))
    binRep = binRep[2:]
    numOps = 0;
    if (binRep == "0"):
        return 1
    while(binRep != "1"):
        if (binRep == "11"):
            return numOps+2
        binRep = str(binRep)
```

```
    if (binRep[-1:] == "0"):
        binRep = binRep[:-1]
        numOps+=1
    else:
        """If there are more 1's following then add instead"""
        if (binRep[-2:-1]=="1"):
            binRep=bin(int(binRep,2)+1)
            binRep=binRep[2:]
        else:
            binRep=bin(int(binRep,2)-1)
            binRep=binRep[2:]
        numOps+=1
return numOps
```