Assignment 2
Verification at Every Tern

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Thursday, February 25, 2021
90 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at http://www.cs.cmu.edu/~15414/assignments.html.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst2.zip` to Assignment 2 (Code). You can generate this file by running `make handin`. This will include your solution `ternary.mlw`, the proof session in `ternary/`, and `bonus.mlw` if you decide to work on this question.
- Submit a PDF containing your answers to the written questions to Assignment 2 (Written). You may use the file `asst2-sol.tex` as a template and submit `asst2-sol.pdf`.

  Make sure your session directories and your PDF solution files are up to date before you create the handin file.

Using LaTeX

We prefer the answer to your written questions to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `asst2.tex` and a solution template `asst2-sol.tex` in the handout to get you started on this.

1 Leave No Tern Unstoned (60 pts)

Balanced ternary numbers are a representation of integers with some remarkable properties. This representation has three digits with values -1, 0, and 1. It represents any integer uniquely (assuming no leading 0s) and has some nice symmetry properties. For example, a number is negated just be negating every digit. An early computer built in Moscow in 1958 actually used balanced ternary numbers and ternary logic, instead of the binary system we are now used to. The Wikipedia article on balanced ternary provides an introduction and more details.

In this problem you are asked to implement and verify some simple functions over ternary numbers. This is partly an exercise is specification suitable for verification, and partly and exercise in working with data types. It may be helpful to review regular expressions (Lecture 4 and live code regexp.mlw) and how we wrote the axioms specifying the interpretation of regular expressions.

> **Each function you write should be verified against contracts expressing the correctness of your implementation.**

The digits d should be either $\overline{1}$, 0, or 1 with values $f(\overline{1}) = -1$, $f(0) = 0$ and $f(1) = 1$. The value of a ternary number $d_n \ldots d_0$ is determined by

$$v(d_n \ldots d_0) = \sum_{i=0}^{n} f(d_i)\, 3^i$$

From a verification perspective, this is difficult to work with due to its use of exponentials. More helpful is the following recurrence:

$$\begin{array}{lcl} v(d_n \ldots d_0) & = & f(d_0) + 3\, v(d_n \ldots d_1) \\ v(\,) & = & 0 \end{array}$$

This suggest representing ternary numbers as a list of digits, *with the least significant bit first*. Note that the representation of a number is not unique, because one can add arbitrarily many leading zeros without changing its value.

For concreteness, we suggest the following representation (which you can find in the file `ternary.mlw`, although you are free to choose a different one. If you choose a different representation, please briefly explain it in a comment in the file.

```
type digit = Z0 | P1 | M1
let function f (d:digit) : int =
match d with Z0 -> 0 | P1 -> 1 | M1 -> -1 end

type tern = list digit
(* least significant digit first *)
(* trailing Z0 digits are allowed *)
```

Note that we defined `let function f` which means that f can be used logically, in contracts, but also computationally. Here are several examples:

Integer	Ternary	WhyML
6	$1\,\overline{1}\,0$	Cons Z0 (Cons M1 (Cons P1 Nil))
-2	$\overline{1}\,1$	Cons P1 (Cons M1 Nil)

Task 1 (10 pts). Specify a predicate `value (t:tern) (a:int)` that relates a ternary number to its integer value by a set of axioms.

Task 2 (5 pts). Define a function `to_int (t:tern) : int` converting a ternary number t to the integer it represents.

Task 3 (10 pts). Define a function `from_int (a:int) : tern` converting an integer a to a ternary number. The module `int.EuclideanDivision` that defines `div` and `mod` functions may be helpful.

> *You may not use the functions* `to_int` *and* `from_int` *in the remaining tasks. Those functions should be defined directly on the ternary representation.*

Task 4 (10 pts). Define functions `inc (t:tern) : tern` and `dec (t:tern) : term` that increment and decrement t, respectively.

Task 5 (5 pts). Define a function `neg (t:tern) : tern` that negates t.

Task 6 (15 pts). Define a function `plus (s:tern) (t:tern) : tern` that computes the sum of s and t.

Task 7 (5 pts). Define a function `is0 (t:tern) : bool` that tests if t has value zero.

2 It's a Question of Semantics (30 pts)

In this collection of problems we work with the simple while language from Lecture 5.

Task 8 (10 pts). Conjecture the semantics of the following program (let's call it α_0):

```
?(n >= 0) ;
x <- n ;
y <- 1 ;
while (x > y)
( x <- div (x+y) 2 ;
  y <- div n x )
```

where div is integer division. You should describe your conjectured semantics in terms of the relation between ω and ν in

$$\omega[\![\alpha_0]\!]\nu$$

For the while loop, describe ω_{init} at the beginning of the loop and ω_{done} at the end, but you do not need to describe the intermediate states.

Task 9 (Bonus Task, not for credit). Describe the intermediate states in the above while loop and prove the correctness of the implementation (possibly with reference to the literature). Can you coax Why3 into verifying the correctness of a suitably translated program? Modifications that retain the algorithmic essence are fair game.

Task 10 (5 pts). Define the semantics of a for-loop

$$\mathsf{for}\ x\ e_1\ e_2\ \alpha$$

which goes through the values for x between the values of e_1 and e_2. It starts at the value of e_1 and counts up or down to the value of e_2, inclusively, executing α each time.

Task 11 (10 pts). Define the semantics of a constructs

$$\mathsf{let}\ x\ e\ \alpha$$

which locally binds x to the value of e while executing α. At the end of α, the value of x should revert to what it was before the let.

Task 12 (5 pts). Define an alternative semantics of the for-loop by showing how to translate it into the while language, including the let construct from the preceding task. You do not have to prove that the two definitions are equivalent.