Why3: Computational Real Numbers

MPRI Project Report

Younesse Kaddar

2019-13-02

Throughout this project, I installed and used the following solvers:

Solver	Version
Alt-Ergo	2.2.0
CVC4	1.6
Z3	4.8.4
CVC3	2.4.1
Eprover	2.2
Spass	3.7

Most of the assertions were proved with Alt-Ergo and CVC4 (less often with Z3, and even more rarely with CVC3, Eprover and Spass). As a macOS user, the installation of Z3 was problematic (its "counterexample" counterpart was the only one to be recognized by the Why3 IDE), so much so that I had choice but to modify my `.why3.conf` file by explicitely adding a block enforcing the use of Z3:

```
[prover]
command = "z3 -smt2 -T:%t sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "z3 -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42
   memory_max_alloc_count=%S %f"
driver = "z3_440"
editor = ""
in_place = false
interactive = false
name = "Z3"
shortcut = "z3"
version = "4.8.4"
```

2. Functions on Integers

Q1-4. Give an implementation of

`power2`, `shift_left` using `power2`

- `power2` and `shift_left` are straightforward: the only notable point is the **for** loop invariant in `power2`:

  ```
  let res = ref 1 in
  for i=0 to l-1 do invariant { !res = power 2 i }
    res *= 2
  done;
  !res
  ```

  which expresses the fact that the reference variable `res` stores the suitable power of 2 at each iteration, and trivially ensures that the postcondition holds:

  - at the last iteration:

    * `!res` contains 2^{l-1} at the beginning of the body loop
    * its value is then doubled, which results in `!res` being equal to 2^l
  - one exits the loop, and `!res` yielded at the end, whence satisfying the postcondition `result = power 2 l` of `power2`

`ediv_mod`, and `shift_right` using `ediv_mod`.

- given `ediv_mod` and `power2`, `shift_right` is easily defined as `let d, _ = ediv_mod z (power2 l)in d` and poses no difficulty.
- `ediv_mod` is slightly more tricky, but nothing to be afraid of: `d` and `r` are respectively the quotient and the rest of the well-known euclidean division of `x` by `y > 0`.

  1. we first tackle the case where $x = \overbrace{|x|}^{\text{denoted by x\_abs}} \geq 0$: as it happens,

```
let x_abs = if x >= 0 then x else -x in
let d = ref 0 in
let r = ref x_abs in
while !r >= y do
  invariant { !r >= 0 && x_abs = !d * y + !r}
  variant { !r }
  incr d;
  r -= y
done;
```

     - the invariant $r \geq 0 \quad \wedge \quad \texttt{x\_abs} = dy + r$ is initially true, and remains so at each iteration of the loop as d (resp. r) is incremented (resp. decremented) by 1 (resp. y).
     - the **`while`** loop condition $r \geq y$ and the fact that $y > 0$ (precondition requirement of `ediv_mod`) justify the decreasing and well-founded variant `!r`
     - at the end the **`while`** loop:
         * $0 \leq \ r < y$
         * $\texttt{x\_abs} = dy + r$

       which provides a trivially correct implementation of the euclidean division, provided $x \geq \ 0$

  2. otherwise, if $x < 0$, we reduce this to the previous case, by computing the corresponding d_abs and r_abs for $\texttt{x\_abs} = |x| = -x$

     - if $\texttt{r\_abs} = 0$: then $\texttt{x\_abs} = \texttt{d\_abs} \times y$, and $x = (-\texttt{d\_abs}) \times y$.

       One yields $d \ \stackrel{\text{def}}{=} \ -\texttt{d\_abs}, \quad r \ \stackrel{\text{def}}{=} \ 0$. This is easily discharged by CVC4 (we can even go as far as to add the extra assertion `assert { x = - !d * y }` to help the provers, but it shouldn't be necessary).

     - else if $\texttt{r\_abs} > 0$: then

$$\begin{cases} 0 \leq y - \texttt{r\_abs} < y \\ x = -\texttt{x\_abs} = -\texttt{d\_abs}\, y - \texttt{r\_abs} = (-\texttt{d\_abs} - 1)\, y + (y - \texttt{r\_abs}) \end{cases}$$

       Therefore, one yields $d \ \stackrel{\text{def}}{=} \ -\texttt{d\_abs} - 1, \quad r \ \stackrel{\text{def}}{=} \ y - \texttt{r\_abs}$.

This is discharged by CVC4 too, but we can add the assertion `assert { x = (- !d - 1)* y + y - !r && 0 <= y - !r < y }` to convince the provers.

Q5. Give an implementation of `isqrt`

When it comes to the sheer body of the function, as seen in class:

```
let function isqrt (n:int) : int
  requires { 0 <= n }
  ensures { result = floor (sqrt (from_int n)) }
  =
    let count = ref 0 in
    let sum = ref 1 in
    while !sum <= n do
      incr count;
      sum += 2 * !count + 1
    done;
    !count
```

However, proving the postcondition `result = floor (sqrt (from_int n))` turns out to be trickier than the one we saw in class (i.e. `sqr !count <= !n < sqr (!count + 1)`), in so far as all the specification pertaining to `floor` in the standard library is:

```
  function floor real : int

  axiom Floor_int :
    forall i:int. floor (from_int i) = i

  axiom Floor_down:
    forall x:real. from_int (floor x) <= x < from_int (Int.(+) (floor x) 1)

  axiom Floor_monotonic:
    forall x y:real. x <= y -> Int.(<=) (floor x) (floor y)
```

That is, the standard-library properties related to $\lfloor \bullet \rfloor$ on which the provers can rely are:

- $\lfloor \bullet \rfloor$ is increasing and left inverse of `from_int`
- and more importantly:

$$\forall n \in \mathbb{Z}, n = \lfloor x \rfloor \implies n \leq x < n+1 \qquad \circledast$$

On top of that, `sqrt` is only assumed to be increasing, and not strictly increasing.

As a result, we:

- *neither* have the converse of $\circledast$ (which is exactly the direction needed to prove the postcondition!)
- *nor* do we have the fact that $\sqrt{\bullet}$ is strictly increasing (which is problematic when dealing with strict inequalities).

So, which assertions where added to prove `isqrt`?

- concerning the **`while`** loop: nothing special, we proceed exactly as seen in class, apart from the extra variant: `variant {n - !sum}` which is easily seen to be strictly decreasing and well-founded.

- at the end of the loop:

$$0 \le \texttt{count} \qquad \text{and} \qquad \texttt{count}^2 \le n < \texttt{sum} = (\texttt{count} + 1)^2$$

therefore, due to $\sqrt{\bullet}$ being strictly increasing and $\texttt{count} \geq 0$:

$$\texttt{count} \le \sqrt{n} < \texttt{count} + 1$$

and the converse of ⊛ would yield the expected postcondition.

But to convince the provers, based solely on the standard-library specification, we proceed as follows:

  - we first show that $\texttt{count} \leq \lfloor\sqrt{n}\rfloor$, which only resorts to $\lfloor\bullet\rfloor$ and $\sqrt{\bullet}$ being increasing and $\sqrt{\bullet}$ being a left inverse of $\bullet^2$ on $\mathbb{R}^+$ (axiom `Square_sqrt` of the standard library).
  - we then show the reverse inequality, that is: $\lfloor\sqrt{n}\rfloor < \texttt{count} + 1$ in a similar fashion. Except that this one is a bit trickier, as $\sqrt{\bullet}$ is not assumed to be strictly increasing, but we can get away with it by treating strict inequalities as being equivalent to non-strict ones *and* non-equalities.

3. Difficulty with Non-linear Arithmetic on Real Numbers

3.1 Power Function

Q6-12. Prove that

1. _B is positive
2. $\texttt{\_B}\, n \times \texttt{\_B}\, m = \texttt{\_B}(n+m)$
3. $\texttt{\_B}\, n \times \texttt{\_B}(-n) = 1$
4. $0 \le a \quad \Longrightarrow \quad \sqrt{a \times \texttt{\_B}(2n)} = \sqrt{a} \times \texttt{\_B}\, n$
5. $0 \le y \quad \Longrightarrow \quad \texttt{\_B}\, y = \texttt{from\_int}\ \ 4^y$
6. $y < 0 \quad \Longrightarrow \quad \texttt{\_B}\, y = \frac{1}{\texttt{from\_int}\ \ 4^{-y}}$
7. $0 \le y \quad \Longrightarrow \quad 2^{2y} = 4^y$

All theses lemmas but the 5th and the 6th ones are straightforwardly discharged:

- for the 5th one (`_B_spec_pos`): we lend a hand to the provers with the command `assert (pow (from_int 4) (from_int n)= from_int (power 4 n))`:

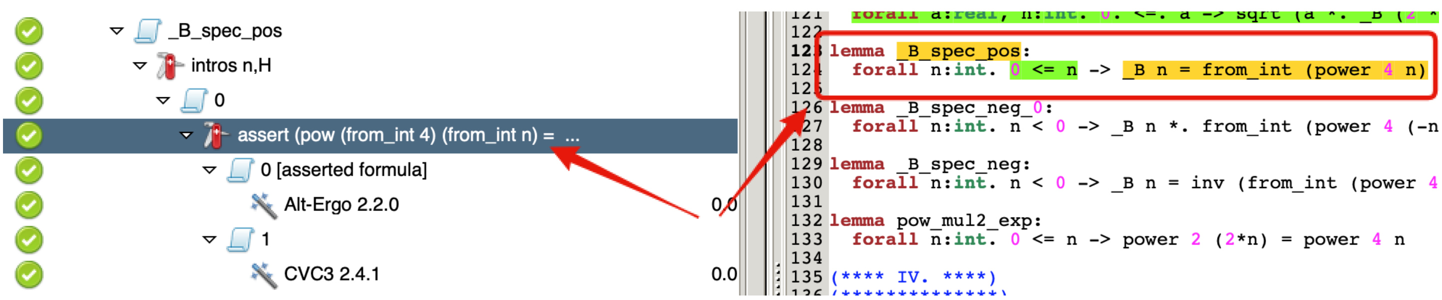

Figure 1: Why3 IDE: use of the `assert` command to prove `_B_spec_pos`

- for the 6th one (_B_spec_neg), we first prove an easily discharged (by Alt-Ergo) lemma:

  ```
  lemma _B_spec_neg_0:
    forall n:int. n < 0 -> _B n *. from_int (power 4 (-n)) = 1.
  ```

  from which _B_spec_neg immediately ensues.

4. Computational Real Numbers

Q13. Could you find a reason why this definition is better than the other for automatic provers?

- When it comes to using to two inequalities rather the terser (and prehaps more elegant)

  $$|x - p\,4^{-n}| < 4^{-n}$$

  the two-inequalities version has the advantage of not involving the absolute value abs, which would just be a burden when proving framing-related postconditions. Indeed, almost every time we would want to show a non-trivial framing (first needing to unfold abs), provers would eventually have to resort to the Abs_le lemma of the standard library, leading to unnecessary proof clutter.

- As for using _B: this fosters the use of the relevant lemmas proved in section **3.6** by the provers, bringing about more efficient proofs.

Q14. Prove these three functions

`round_z_over_4`

By dint of assertions, we show the two postconditions inequalities separately:

- $$\texttt{from\_int}\ \big(\underbrace{\texttt{shift\_right}\ \ (z+2)\,2}_{=(z+2)/\!/2^2}\big) \leq (\texttt{from\_int}\ \ z+2) \times \texttt{\_B}(-1)$$

  where // stands for the euclidean division quotient, which directly stems from

  $$4((z+2)\ /\!/\ 2^2) \leq z+2 \qquad \text{(euclidean division)}$$

- Similarly (the `from_int` 's will be omitted from now on):

  $$z-2 < 4 \times \underbrace{\texttt{shift\_right}\ \ (z+2)\,2}_{=(z+2)/\!/2^2}$$

  due to

  $$z-2 < z+2 - \underbrace{((z+2) \mod 2^2)}_{<4} = 4((z+2)\ /\!/\ 2^2)$$

`compute_round` and `compute_add`

- For `compute_round`, assuming

$$(z_p - 2) \times \text{\_B}(-(n+1)) < z \leq (z_p + 2) \times \text{\_B}(-(n+1))$$

  we show that

$$(\underbrace{\texttt{shift\_right}\ \ (z_p+2)\,2}_{=(z_p+2)\,/\!/\,2^2} - 1) \times \text{\_B}(-n) < z < ((z_p+2)\,/\!/\,2^2 + 1) \times \text{\_B}(-n)$$

  by means of two assertions (one for each inequality). Indeed:

$$\begin{aligned}
((z_p+2)\,/\!/\,2^2 - 1) \times \text{_B}(-n) &\leq \Big(\underbrace{\frac{z_p+2}{4} - 1}_{=\frac{z_p}{4}-\frac{1}{2}}\Big) \times \text{_B}(-n) && \text{since } 4((z_p+2)\,/\!/\,2^2) \leq z_p + 2 \\
&= \frac{z_p - 2}{4} \times \text{_B}(-n) \\
&= (z_p - 2) \times \text{_B}(-(n+1)) \\
&< z \\
&\leq \frac{z_p + 2}{4} \times \text{_B}(-n) \\
&= \Big(\frac{z_p - 2}{4} + 1\Big) \times \text{_B}(-n) \\
&< ((z_p+2)\,/\!/\,2^2 + 1) \times \text{_B}(-n) && \text{since } z_p - 2 < 4((z_p+2)\,/\!/\,2^2) \text{ as seen before}
\end{aligned}$$

- Given `compute_round`'s contract, `compute_add n x xp y yp` is straightforwardly defined as `compute_round n (x +. y)(xp + yp)`

4.2 Subtraction

Q15-16. Define and prove the functions `compute_neg`, `compute_sub` using `compute_neg` and `compute_add`

Those pose no difficulty:

- `compute_neg n x xp` is nothing more than `-xp`, by multiplying the framing of `x` by -1
- `compute_sub n x xp y yp` compute_adds `x` and the compute_neg'ed approximation of `y`, owing to `x` and `y` being provided at approximation $n+1$. A little help for the provers: asserting `assert { framing (-.y)yp' (n+1)}` just before yielding the result.

4.3 Conversion of Integer Constants

4.4 Square Root

Q17. Prove these two relations

Q18. Prove `compute_sqrt`

4.5 Compute

Q19. define a logic function `interp` that gives real interpretation of a term with the usual semantic for each operation

Q20. define `wf_term` that checks that square root is applied only to terms with non negative interpretation.

Q21. define and prove the `compute` function

5 Division

Q22. Prove these two properties

Q23. Prove the function `inv_simple_simple`

Q24. Prove the function `inv_simple`

Q25. extend the type `term`

Q26. prove both functions

Q27. prove the termination of the functions