

Computational Real (v1)

This project is to be carried out using the Why3 tool, in combination with automated provers (Alt-Ergo (v2.0), CVC4 (v1.5) and Z3 (v4.4.1) (Eprover 1.9-1-001)). You can use other automatic provers or version if you want, if they are freely available and recognized by why3. You may use Coq for discharging particular proof obligations, although the project can be completed without it. To get started, you need to install the **latest version** of Why3. The installation procedure may be found on the web page of the course.¹

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to francois.bobot@cea.fr and jean-marie.Madiot@inria.fr, no later than **Friday, February 22th, 2018** at 22:00 UTC+1. This e-mail should be entitled “Project”, be signed with your name, and have as attachment an archive (zip or tar.gz) storing the following items:

- The source file `creal.mlw`.
- The content of the sub-directory `creal` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for your grade for the project.

The report must be written in French or English, and should typically consist of 2 to 4 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you add should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *“For this function, I propose the following implementation: [give a pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To proof this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain].”*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g. why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard was it, what were the major difficulties, was there any unexpected result, and any other information that you think are important to consider for the evaluation of the work you did.

0 Goal

The goal of this project is to prove a little library of computational real that contains:

- conversion from integer constant
- addition

¹<https://francois.bobot.eu/mpri2018/>

- negation
- subtraction
- square root
- inverse

It is based on the thesis of Valérie Ménessier-Morain *ARITHMÉTIQUE EXACTE: Conception, algorithmique et performances d'une implémentation informatique en précision arbitraire* chapter 3.

A term t formed with these operations is interpreted into a real with the function `interp` into a real.

For a term formed with these operation and a positive precision n , the function `compute` returns an integer `res` that verifies:

$$(\text{res} - 1) * 4^{-n} < \text{interp}(t) < (\text{res} + 1) * 4^{-n}$$

So the result is an approximation of the real interpretation of the term up to 4^{-n} .

1 Axioms

We are going to use the standard library of Why3 for all the definition of the mathematical function used. The defined properties are sufficient except for the logarithm. So we add the only axioms needed for the project:

$$\forall xy, 0 < x < y \implies \log(x) < \log(y)$$

2 Function on integers

We are going to use the function from `mach.int.Int` of the standard library for providing the basic operation on integer of arbitrary size with the additional functions:

```

use mach.int.Int
use real.RealInfix
use int.Power

use int.EuclideanDivision as ED

val power2 (l:int)
  requires { 0 ≤ l }
  ensures { result = power 2 l }

val shift_left (z: int) (l:int) : int
  requires { 0 ≤ l }
  ensures { result = z * (power 2 l) }

val ediv_mod (x:int) (y:int) : (int, int)
  requires { 0 < y }
  ensures { let d,r = result in
    d = ED.div x y ∧ r = ED.mod x y }

val shift_right (z: int) (l:int) : int
  requires { 0 ≤ l }
  ensures { result = ED.div z (power 2 l) }
```

```

use real.Square
use real.FromInt
use real.Truncate

```

```

val function isqrt (n:int) : int
  requires { 0 ≤ n }
  ensures { result = floor (sqrt (from_int n)) }

```

1. *Give an implementation of power2*
2. *Give an implementation of shift_left which uses power2*
3. *Give an implementation of ediv_mod*
4. *Give an implementation of shift_right which uses ediv_mod*
5. *Give an implementation of isqrt*

You may need to force the provers to prove that a value is the division of two numbers, for that you can use the following lemma function:

```

let lemma euclid_uniq (x y q : int) : unit
requires { y > 0 }
requires { q * y ≤ x < q * y + y }
ensures { ED.div x y = q } =
  ()

```

3 Difficulty of Non-linear Arithmetic on Reals

The correction of this program depends heavily on mathematical properties that involve multiplication or division on arbitrary terms. Provers doesn't handle them well natively. They usually only support linear arithmetic, multiplication and division with one constant argument. So we need to guide them.

You could use successive assertions and use the connective *by* and *so*.

```

assert { A by B so C so D }

```

This assertions is A if is not splitted in the *Why3* ide, otherwise it splits into four sub-goals: B, $B \rightarrow C$, $C \rightarrow D$, and $D \rightarrow A$. The advantage compared to multiple assertions is that B, C and D doesn't pollute the context for proving new assertions:

```

assert { B };
assert { C };
assert { D };
assert { A };
assert { next; it has B, C, D and A in the context }

```

Another possibility is to use lemma functions. The function `euclid_uniq` is an example. The lemmas of the next section are good candidates for lemma functions.

3.1 Power

The thesis proves the corrections of the algorithm for any base b . Here we are choosing $b = 4$. We define the logic function `_B` as 4^n using the `pow` function from `real.PowerReal`:

```

function _B (n:int) : real = pow b (from_int n)

```

We need some properties on this function:

6. *prove that $_B$ is positive*
7. *prove that $_B\ n \ *.\ _B\ m = _B\ (n+m)$*
8. *prove that $_B\ n \ *.\ _B\ (-n) = 1$.*
9. *prove that $0. \leq a \rightarrow \text{sqrt}\ (a \ *.\ _B\ (2*n)) = \text{sqrt}\ a \ *.\ _B\ n$*
10. *prove that $0 \leq y \rightarrow _B\ y = \text{from_int}\ (\text{power}\ 4\ y)$*
11. *prove that $y < 0 \rightarrow _B\ y = \text{from_int}\ (\text{power}\ 4\ (-y))$*
12. *prove that $0 \leq y \rightarrow \text{power}\ 2\ (2 \ *.\ y) = \text{power}\ 4\ y$*

4 Computational Real

The goal is to compute approximate reals by an integer, we use the second definition of the framing:

```
predicate framing (x:real) (p:int) (n:int) =
  (from_int p -. 1.) *. ( $\_B\ (-n)$ ) <. x <. (from_int p +. 1.) *. ( $\_B\ (-n)$ )
```

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

4.1 Addition

We want to implement the function that compute the framing of an addition from the framing a little more precise of the arguments:

```
let compute_add (n: int) (ghost x : real) (xp : int) (ghost y : real) (yp : int)
  requires { framing x xp (n+1) }
  requires { framing y yp (n+1) }
  ensures { framing (x+.y) result n } =
  compute_round n z (xp + yp)
```

We use the two following auxiliary functions:

```
let compute_round (n:int) (ghost z : real) (zp: int)
  requires { (from_int zp -. 2.) *. ( $\_B\ -(n+1)$ ) <. z <=. (from_int zp +. 2.) *. ( $\_B\ -(n+1)$ ) }
  ensures { framing z result n } =
  round_z_over_4 zp

let round_z_over_4 (z : int)
  ensures { ((from_int z) -. 2.) *. ( $\_B\ (-1)$ ) <. from_int result <=. ((from_int z) +. 2.) *. ( $\_B\ (-1)$ ) }
  =
  shift_right (z + 2) 2
```

14. *Prove these three functions*

4.2 Subtraction

15. *Define and prove the function `compute_neg` that computes the framing of the negation of a real using its framing at the same precision*
16. *Define `compute_sub` using `compute_neg` and `compute_add`*

4.3 Conversion of Integer Constant

The conversion from integer constant is in fact simple:

```
let compute_cst (n: int) (x : int) : int
  ensures { framing (from_int x) result n } =
  if n = 0 then
    x
  else if n < 0 then
    shift_right x (2*(-n))
  else
    shift_left x (2*n)
```

4.4 Square Root

The code is simply:

```
let compute_sqrt (n: int) (ghost x : real) (xp : int)
  requires { 0. ≤. x }
  requires { framing x xp (2*n) }
  ensures { framing (sqrt x) result n } =
  if xp ≤ 0 then
    0
  else
    isqrt xp
```

We use a proof different for the square root than the one of the thesis. In fact the last case of the proof in the thesis is applicable to all the previous case, which simplifies a lot the proof. The idea is to show that square roots of two successive numbers are close even after taking the floor or ceiling because they are in the same integer or one of them is an integer. For $n \geq 1$ an integer :

$$\lceil \sqrt{n+1} \rceil - 1 \leq \lfloor \sqrt{n} \rfloor \leq \lfloor \sqrt{n-1} \rfloor + 1$$

17. Prove these two relations

18. Prove `compute_sqrt`

4.5 Compute

We are defining terms as the following algebraic datatype:

```
type term =
| Cst int
| Add term term
| Neg term
| Sub term term
| Sqrt term
```

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

The function `compute` has the following contract:

```
let compute (t:term) (n:int) : int
  requires { wf_term t }
  ensures { framing (interp t) result n }
```

20. define *wf_term* that checks that square root is applied only to terms with non negative interpretation.
21. define and prove the *compute* function

5 Division

We are now supposing that the precision is always smaller than 1, i.e. $0 \leq n$.

The computation is done by successively reducing to simpler case, finally an argument similar to the one of square root is used. The algorithm is different from the one of the thesis.

1. negative number are handled as positive one
2. positive reals smaller than one are handled as reals bigger than one by multiplying by 4^m with a sufficiently big m
3. positive reals bigger than one are inverted.

So the inverse function is computed using the first auxiliary functions:

```

let inv_simple_simple (ghost x:real) (p:int) (n:int)
  requires { framing x p (n+1) }
  requires {  $0 \leq n$  }
  requires {  $1. \leq x$  }
  ensures { framing (inv x) result n } =
  let k = n + 1 in
  let d,r = ediv_mod (power2 (2*(n+k))) p in
  if 2*r ≤ p then d else d+1

```

The proof use the fact that if the quotient is smaller than the dividend then the quotient change of at most one when the dividend changes by one (div and mod are euclidean division **import** int.EuclideanDivision as ED):

$$0 < a \implies 0 < b \implies ED.div(a, b) < b \implies$$

$$ED.div(a, b+1) = \begin{cases} (ED.div(a, b)) - 1 & ED.mod(a, b) < ED.div(a, b) \\ (ED.div(a, b)) & otherwise \end{cases}$$

$$0 < a \implies 1 < b \implies ED.div(a, b) < b - 1 \implies$$

$$ED.div(a, b-1) = \begin{cases} (ED.div(a, b)) + 1 & b - 1 - ED.div(a, b) \leq ED.mod(a, b) \\ (ED.div(a, b)) & otherwise \end{cases}$$

The proof also use the fact that in the case of *inv_simple_simple*, $ED.div(a, b) \leq b - 1 - ED.div(a, b)$, so the two first case can't happend at the same time.

22. Prove these two properties
23. Prove the function *inv_simple_simple*

The second auxiliary function, just change the ghost part:

```

let inv_simple (ghost x) p m n
  requires {  $0 \leq m$  }
  requires {  $0 \leq n$  }
  requires {  $\_B (-m) < . x$  }
  requires { framing x p (n+1+2*m) }
  ensures { framing (inv x) result n } =
    inv_simple_simple (x *.  $\_B m$ ) p (n+m)

```

24. Prove the function *inv_simple*

The sufficiently big m is computed by *msd* which look for an approximation of the term strictly bigger than 1 by increasing the precision. By supposing that the term is non zero we are sure that such m exists and is smaller than $-\lfloor \log_2(|\text{interp}(t)|) \rfloor$. The logic function \log_2 is defined in `real.ExpLog`.

The function *msd* is mutually recursive with the function *compute* and it is simpler if it indicates the sign of the real:

```

let rec compute (t:term) (n:int) : int
  requires { wf_term t }
  requires {  $0 \leq n$  }
  ensures { framing (interp t) result n }
=
  match t with
  | Cst i  $\rightarrow$  ...
  | Add t1 t2  $\rightarrow$  ...
  | Neg t1  $\rightarrow$  ...
  | Sub t1 t2  $\rightarrow$  ...
  | Sqrt t1  $\rightarrow$  ...
  | Inv t  $\rightarrow$ 
    let m,sgn = msd t 0 (compute t 0) in
    let p = compute t (n+1+2*m) in
    if sgn
    then inv_simple x p m n
    else
      - (inv_simple (-. x) (-p) m n)
  end

with msd (t:term) (n:int) (c:int) : (int, bool)
  requires {  $0 \leq n$  }
  requires { wf_term t }
  requires { interp t  $\neq$  0. }
  requires { framing (interp t) c n }
  ensures { let m,sgn = result in
     $0 \leq m \wedge$ 
    if sgn then  $\_B (-m) < . \text{interp } t$  else  $\text{interp } t < . -. \_B (-m)$ 
  }
=
  if c = 0 || c = 1 || c = -1 then begin
    let c = compute t (n+1) in
    msd t (n+1) c
  end
  else begin
    if 1 < c then

```

```
    n, true
  else
    n, false
end
```

- 25. *extend the type term, the*
- 26. *prove both functions*
- 27. *prove the termination of the functions*

6 Bonus

The bonus and hard question is to prove the original algorithm from the thesis for the inverse (which uses ceiling or flooring according to the sign, not rounding) or find a counterexample that breaks it.

7 Extraction

You can extract all your code using:

```
why3 extract -D ocaml64 -o creal.ml creal.mlw
```

and execute it.