Unions of Intervals, a Data Structure used for Program Verification

The goal of this project is to formalize the abstract domain of unions of intervals and some of its operations. This data structure is used for program verification in abstract interpretation analyzers and constraint propagation solvers. This abstract domain is an extension of the classical domain of intervals. The set of values will be abstracted, but a possible instantiation should be the rational numbers. We also require the domain to be able to express open and closed intervals, and unbounded intervals.

This project is to be carried out using the Why3 tool (version 1.7.x or higher), in combination with automated provers (e.g. Alt-Ergo 2.5.x, CVC4 1.8, CVC5 1.0.x or higher and Z3 4.12.x or higher). You can use other automatic provers or versions 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. The installation procedure may be found on the web page of the course at URL `https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html`. You can also use Why3find[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 **Tuesday, February 11th, 2025** at 23:00 UTC+1. This e-mail should be entitled "MPRI project 2-36-1", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The solution file `union.mlw`.
- The content of the sub-directory `union` generated by Why3 or Why3find. 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 at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 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 added 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 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 prove 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 it was, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

This year's project gives you a lot of liberties, so your report should in particular discuss the choices you made.

[1] `https://git.frama-c.com/pub/why3find/-/blob/master/README.md`

1 Semantic

Let V be an ordered field subset of the reals[2], we propose an axiomatization in Why3 in Figure 1.

In why3 the module is imported using **`import`** `Intf as V` **`with axiom`** `equal_is_eq`, the type is accessible as `V.t`. A skeleton of the final why3 file is shown in Figure 2. Do not hesitate to completely modify this skeleton, in particular auxiliary types, predicates, lemmas, lemma functions, and ghost results, could be needed. A function `V.real` maps an element to the corresponding real. In Why3, since we added a coercion from `V.t` to `real` using this function, it can sometimes be omitted. In the following we will use γ for relating the implementation to the mathematical concept.

We are interested in open, closed, unbounded and bounded intervals. Formally, we define an interval of elements in V as one of the following sets:

- $\emptyset$
- $\{z \in V \mid z \lozenge x\}$ with $x \in V$, and $\lozenge \in \{<, \leq\}$
- $\{z \in V \mid x \lozenge z\}$ with $x \in V$, and $\lozenge \in \{<, \leq\}$
- $\{z \in V \mid x \lozenge z \blacklozenge y\}$ with $x, y \in V$ and $\lozenge, \blacklozenge \in \{<, \leq\}$

Let I be the set of all the intervals of V. We define U as the closure of I under finite unions:

$$U = \left\{ \bigcup_{i \in J} i \;\middle|\; J \in \mathcal{P}_{fin}(I) \right\}$$

At first, it could be easier to restrict yourself to closed intervals.

1.1 Representation

Elements of U can be represented in many ways. We are not interested in the most efficient data structure, we are interested in the representation that simplifies the most the implementations, specifications and proofs of its operations. You are free to use any representation for that purpose. If the proofs of the operations asked below are too complicated, it could be useful to look for an alternative definition.

In order to help you consider alternative definitions, consider some possible choices:

- The definition can be composed on one or multiple type definitions.
- The invariant of the type definition used can be represented using a type invariant or by using a predicate used in precondition and post-condition.
- The type definition can use records, algebraic datatypes, tuples, ...
- Some fields can be ghost.
- V can be extended, or not, with ∞.
- The intervals can be unsorted, sorted, or strictly sorted.

In the following, the type representation in Why3 of U will be noted `u`.

1. *Give the definition of* `u` *(your answer to this question might be definitive only after solving the other questions).*

We suppose that you can extend the function γ to `u` $\to U$. In the why3 specification it is easier to combine γ and the membership.

2. *Give the definition of* **`predicate`** `mem (x:real) (l:u)` *which specify for a real* `x` *that* `x` $\in \gamma(l)$.

[2] https://en.wikipedia.org/wiki/Ordered_field

```
module Intf
    use real.RealInfix     use int.Int    use real.FromInt

    type t

    function real (q:t) : real        meta coercion function real

    predicate equal (a b:t) = (a.real = b.real)

    val (=) (a:t) (b:t) : bool ensures { equal a b = result }

    axiom equal_is_eq: forall a b. equal a b -> a = b

    predicate (<=) (a:t) (b:t) = (a.real <=. b.real)
    predicate (<)  (a:t) (b:t) = (a.real <.  b.real)

    val (<=) (a:t) (b:t) : bool ensures { result = (a <= b) }
    val (<)  (a:t) (b:t) : bool ensures { result = (a <  b) }

    use real.Abs
    val abs (a:t) : t
    ensures { result.real = abs a.real }

    type ord = | Eq | Lt | Gt
    val compare (a:t) (b:t) : ord
        ensures {
          match result with
            | Eq -> equal a b  | Lt -> a < b | Gt -> b < a
          end
        }

    val function (+)   (a b:t) : t ensures { a.real +. b.real = result.real }
    val function (-)   (a b:t) : t ensures { a.real -. b.real = result.real }
    val function ( * ) (a b:t) : t ensures { a.real *. b.real = result.real }

    val function (/) (a b:t) : t
        requires { b.real <> 0. }
        ensures { a.real /. b.real = result.real }

    use real.Truncate

    val of_int (a:int) : t ensures { result.real = from_int a }

    val truncate (a:t) : int ensures { result = truncate a.real }
    val floor    (a:t) : int ensures { result = floor    a.real }
    val ceil     (a:t) : int ensures { result = ceil     a.real }
end
```

Figure 1: Axiomatization of an ordered field V subset of the reals using Why3 standard library and with additional operations

1.2 Properties

The two main properties that need to be proved for each operation are classically *soundness* and *completeness* (also referred as preciseness). Soundness expresses that no values are lost and completeness precise that no values are uselessly added.

1.3 Set operations

The main set operations. Note that the proofs of completeness are more difficult than the proofs of soundness.

3. *Implement and prove the soundness and completeness of the function* `let singleton (q:V.t) : u` *such that* $\gamma(\texttt{singleton(q)}) = \{\gamma(q)\}$

4. *Implement and prove the soundness and completeness of the function* `let gt (q:V.t) : u` *such that* $\gamma(\texttt{gt(q)}) = \{x \mid \gamma(q) < x\}$

5. *Implement and prove the soundness and completeness of the function* `let ge (q:V.t) : u` *such that* $\gamma(\texttt{ge(q)}) = \{x \mid \gamma(q) \leq x\}$

6. *Implement and prove the soundness and completeness of the function that remove an element* `let except (v:V.t) : u` *such that* $\gamma(\texttt{except(v)}) = \mathcal{R} \setminus \{\gamma(v)\}$.

7. *Implement and prove the soundness and completeness of the function* `let union (v:u) (w:u) : u` *such that* $\gamma(\texttt{union(v,w)}) = \gamma(v) \cup \gamma(w)$

8. *Implement and prove the soundness and completeness of the function* `let inter (v:u) (w:u) : u` *such that* $\gamma(\texttt{inter(v,w)}) = \gamma(v) \cap \gamma(w)$

1.4 Comparisons of Union of Intervals

In this section, we are implementing the subset and the disjoint operation.

9. *Implement and prove only the soundness of the function* `subset(a,b)`. *If the function returns true then* $\gamma(\texttt{a}) \subset \gamma(\texttt{b})$

10. *Implement and prove only the soundness of the function* `disjoint(a,b)`. *If the function returns true then* $\gamma(\texttt{a}) \cap \gamma(\texttt{b}) = \emptyset$

The completeness means proving that when the function return false there is really a counter-example.

11. *Try to prove the completeness of one of those two functions, what are the difficulties? Does it help to provide the witness by computing in ghost? Returning it?*

1.5 Strictly Monotone Unary operators

We are interested in functions that compute the image of a union of intervals by unary operators. Instead of defining the function separately for multiple unary operators, we will implement a higher level function, that compute the image by any strictly monotone unary operator. We suppose operators are pure functions and we regroup them with their concretization in the type `op`. The field `ur` is γ (`uq`).

```
type op = {
  ghost ur: real -> real;
  uq: V.t -> V.t;
}
  invariant { forall v: V.t. ur (V.real v) = V.real (uq v) }
  by { ur = (fun x -> x); uq = (fun q -> q); }
```

12. *Define the predicate* `si`*, which specify that an operator of type* `op` *is strictly increasing.*

13. *Define the function* `op_si`*, and prove its soundness and* simplified *completeness for a strictly increasing operator:* $\forall q.\, q \in \gamma(u) \Leftrightarrow \gamma(\texttt{op})(q) \in \gamma(\texttt{op\_si}(\texttt{op}, u))$ *. What is the difference with the completeness defined as* $\gamma(\texttt{op\_si}(\texttt{op}, u)) = \{\gamma(\texttt{op})(x) \mid x \in \gamma(u)\}$ *?*

14. *Define the function* `cube(u)` *which compute the cube* ($x \mapsto x^3$) *of a union of intervals, and prove its soundness and completeness.*

15. *Define the predicate* `sd`*, which specify that an operator of type* `op` *is strictly decreasing.*

16. *Define the function* `op_sd`*, and prove only its soundness: for a strictly decreasing operator* `op`*,* $\gamma(\texttt{op\_sd}(\texttt{op}, u)) \subset \{\gamma(\texttt{op})(x) \mid x \in \gamma(u)\}$

17. *Define the function* `neg(u)` *which compute the opposite* ($x \mapsto -x$) *of a union of intervals, and prove only its soundness.*

18. *Define the function* `mul_cst(c,u)` *which compute the multiplication by a constant of a union of intervals, and prove only its soundness.*

1.6 Monotone Unary operators

19. *In a similar way to the previous section implements the function* `ceil`*,* `floor`*,* `truncate`*,* `relu` [3] *functions. Only soundness is requested.*

1.7 Bottom, the empty set

The empty set is an important element since it can show that the analyzer reached an unreachable state or the solver reached an unsatisfiable state.

20. *How would you change the definitions in order to forbid this case and warn the caller when it is reached? (You do not have to modify your code)*

2 Conclusions

Do not forget to end your report with a conclusion that summarizes your achievements, explain the issues you could not solve, if any, and comment about what you learned when doing this project.

[3] `max(x,0)` used in neural networks

```
module Skel
 use int.Int
 use bool.Bool
 use real.RealInfix

 scope Make
  clone Intf as V
    with axiom equal_is_eq

  type u (** = ... *)

  (** Just some unsound placeholders *)
  predicate _fmla
  val _code () : u
  ensures { false }
  val _codeb () : bool
   ensures { false }

  predicate mem (x:real) (l:u) = _fmla

  let singleton (q:V.t) : u
    ensures { _fmla } = _code ()

  let gt (q:V.t) : u
    ensures { _fmla } = _code ()

  let ge (q:V.t) : u
    ensures { _fmla } = _code ()

  let except (x:V.t) : u
    ensures { _fmla } = _code ()

  let inter (u:u) (v:u) : u
    ensures { _fmla } = _code ()

  let union (u:u) (v:u) : u
    ensures { _fmla } = _code ()

  let subset (t1:u) (t2:u) : bool
    ensures { _fmla } = _codeb ()

  let disjoint (t1:u) (t2:u) : bool
    ensures { _fmla } = _codeb ()

  type op = {
    ghost ur: real -> real;
    uq: V.t -> V.t;
  }
  invariant { forall v: V.t.
                  ur (V.real v) = V.real (uq v) }
  by { ur = (fun x -> x); uq = (fun q -> q); }

  (** unary strictly increasing *)
  predicate si (op: op) = _fmla

  let op_si (op: op) (l:u) : u
    requires { si op }
    ensures { _fmla } = _code ()

  let cubic (l:u) : u
    ensures { _fmla } = _code ()

  (** unary strictly decreasing *)
  predicate sd (op: op) = _fmla

  let op_sd (op: op) (l:u) : u
    requires { sd op }
    ensures { _fmla } = _code ()

  let neg (l:u)
      ensures { _fmla } = _code ()

  let mult_cst (c:V.t) (l:u) : u
      ensures { _fmla } = _code ()

  (** unary non-strict increasing *)
  predicate nsi (op:op) = _fmla

  let op_nsi (op:op) (l:u) : u
    requires { nsi op }
    ensures { _fmla } = _code ()

  use real.Truncate
  use real.FromInt

  let ceil (l:u) : u
    ensures { _fmla } = _code ()

  let floor (l:u) : u
    ensures { _fmla } = _code ()

  let truncate (l:u) : u
    ensures { _fmla } = _code ()

  use real.MinMax

  let relu (l:u) : u
    ensures { _fmla } = _code ()

 end
end
```

Figure 2: A skeleton of the Why3 implementation