Simple proofs of simple programs in Why3

Jean-Jacques Lévy
State Key Laboratory for Computer Science,
Institute of Software, Chinese Academy of Sciences
& Inria

Abstract

We want simple proofs for proving correctness of simple programs. We want these proofs to be checked by computer. We also want to use current interactive or automatic provers and not build new ones. Finally Hoare logic is a good framework to avoid complex definitions of the operational semantics of programming languages. In this short note, we take the example of merge-sort as expressed in Sedgewick's book about Algorithms and demonstrate how to prove its correctness in Why3, a system developed at Université de Paris-Sud, Cnrs and Inria. There are various computer systems which integrate proofs and programs, e.g. VCC, Spec#, F*, Frama-C, etc. Why3 integrates a small imperative language (Why ML) and an extension of Hoare logic with recursive data types and inductive predicates. It is interfaced with interactive systems (Coq, Isabelle/HOL, PVS) and automatic provers (Alt-Ergo, Z3, CVC3, CVC4, E-prover, Gappa, Simplify, Spass, Yices, etc). Therefore Why3 can also be considered as a fantastic back-end for other programming environments.

1 Introduction

Formal proofs of program safety are always a big challenge. Usually they comprise a large number of cases, which make them intractable on paper. Fortunately there are a few proof-assistants which guarantee the exactness of formal proofs, but less many mixing programs and proofs. Moreover we believe that these computer-checked proofs should be readable and simple when we have to prove simple programs. In this note, we take Hoare logic as the basic formal setting and we consider a simple Pascal-like imperative programming language (Why ML) with an ML-like syntax. This logic is first-order with several extensions to allow recursive data and inductively defined predicates. We hope that both

syntax of the programs and logic are self-explainable. If not, the interested reader is referred to the Why3 web site. We illustrate here the use of the Why3 system through a proof of the merge-sort program. We consider two versions of it: on lists and on arrays. The version on arrays is far more complex to prove correct.

2 Mergesort on lists

In Why ML, merge-sort on lists of integers is expressed in figure 1. Its correctness proof is easy. Pre and post-conditions follow keywords **requires** and **ensures** in the headers of functions. In post-conditions, `result` represents the value returned by the function; `sorted`, `permut`, `(++)` are predicates and functions defined in the theory of polymorphic lists in the Why3 standard library (located at URL `http://why3.lri.fr/stdlib-0.83`). Part of this theory is visible in figure 2. The verification conditions generated by Why3 can be proved automatically with Alt-Ergo, CVC3 and Eprover 1-6. The longest proof is the one for the post-conditions of `merge` (5.02 sec by CVC3) and `split` (2.05 sec by Eprover). These timings are obtained on a regular dual-core notebook. The choice of the provers and of the transformations to apply to goals (splitting conjunctions, inlining function definitions) is manual, but easy to perform thanks to the Why3 graphic interface.

Moreover the assertions are natural and look minimal. Maybe the most mysterious part is the post-condition of `merge`, which states that the result is a permutation of the concatenation ℓ_1 `++` ℓ_2 of the parameters ℓ_1 and ℓ_2 of `merge`. This property is needed to prove the `sorted` post-condition since `x1` (or `x2`) should be ranked with respect to the result of the recursive calls of `merge`. In fact it is sufficient to know that that `result` only contains elements of the lists ℓ_1 (or ℓ_2). Now the proof of `permut` in the post-condition of `merge` is totally orthogonal and is resumed in the second part of figure 1.

3 Mergesort on arrays

We now consider the program as written in Sedgewick and Wayne (2011). In this version of `mergesort` there is a trick in organizing the area to merge as a bitonic sequence, increasing first and decreasing afterwards (although not expressed in that way in the book). It thus avoids multiple loops or tests as halting conditions of loops. See the code on figure 3 in Why ML language where the second half of `a` is copied into second half of `b` in reverse order. In pre and post-conditions, new predicates or functions are used: `sorted_sub` and `permut_sub` mean `sorted` and `permut` on subarrays between bounds `lo` (included) and `hi` (excluded); (`old a`) means the array `a` before calling the function.

```
let rec split (l : list int)                          (* first part *)
 = match l with
 | Nil -> (Nil, Nil)
 | Cons x Nil -> ((Cons x Nil) , Nil)
 | Cons x (Cons y l') -> let (xs,ys) = split l' in
                                  ((Cons x xs), (Cons y ys))
 end

let rec merge l1 l2
  requires { sorted l1 /\ sorted l2}
  ensures { sorted result /\ permut result (l1 ++ l2) }
 = match l1, l2 with
   | Nil, _ -> l2
   | _, Nil -> l1
   | Cons x1 r1, Cons x2 r2 ->
       if x1 <= x2 then Cons x1 (merge r1 l2)
                   else Cons x2 (merge l1 r2)
   end

let rec mergesort l
  ensures { sorted result }
   = match l with
   | Nil | Cons _ Nil -> l
   | _ -> let l1, l2 = split l in merge (mergesort l1) (mergesort l2)
   end

                                                      (* second part *)
let rec split (l : list int)
 ensures { let (l1, l2) = result in permut l (l1 ++ l2)} = ...

let rec merge l1 l2
  ensures { permut result (l1 ++ l2)}  = ...

let rec mergesort l
  ensures { permut result l}  = ...
```

Figure 1: Mergesort on lists

The proof of `sorted` in post-condition of `mergesort` needs several add-ons to the theory of arrays in the Why3 standard library as shown in figure 4. An array is represented by a record with two fields: an integer `length` and a total map `elts` from integers to values. The functions `get` and `set` reads and writes a value from or into an element of an array (or map). Thus we define the predicates `array_eq_sub_rev_offset`, `dsorted_sub` and `bitonic_sub` both on arrays and maps. (Why3 translates predicates over arrays into predicates over maps, where the solvers are mainly acting). The first predicate is a technical abbrevia-

```
function (++) (l1 l2: list 'a) : list 'a = match l1 with
    | Nil -> l2
    | Cons x1 r1 -> Cons x1 (r1 ++ l2)
end

inductive sorted (l: list t) =
    | Sorted_Nil:
        sorted Nil
    | Sorted_One:
        forall x: t. sorted (Cons x Nil)
    | Sorted_Two:
        forall x y: t, l: list t.
        le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

function num_occ (x: 'a) (l: list 'a) : int =
    match l with
    | Nil      -> 0
    | Cons y r -> (if x = y then 1 else 0) + num_occ x r
   end

predicate permut (l1: list 'a) (l2: list 'a) =
    forall x: 'a. num_occ x l1 = num_occ x l2
```

Figure 2: Theory of lists

tion to test for equality after reversing and adding an offset to a subarray. The two last predicates mean down-sorted or bitonic on sub-arrays (and sub-maps). We finally add two lemmas about weakening the interval of a bitonic subarray.

To prove the first part (`sorted_sub`) of the post-condition of `mergesort1`, we add several assertions and invariants in its body as shown on figure 5. We use a new operator (`at a 'L`) in formulas meaning the value of array `a` at label `L`. The proof of the 161 verification conditions is fully automatic with the Alt-Ergo prover and quasi online. But this happened after many attempts and reformulations of assertions and invariants and use other provers. Retries are favourised by the incremental dependency analysis (stored in a so-called `why3session.xml` file) of Why3 which only recomputes the modified goals.

To summarize the logic of this function, there are two recursive calls on both halves of array `a`; the first half is copied into the first half of array `b`; the second half is copied in reverse order into the second half of array `b`; finally the merge of the two halves of `b` is returned in `a`. Notice that during the merge phase, the index `j` can go over the half `m` of the array `b`. Therefore the assertion `m <= !j` is not true since the index `j` can go up to `lo` when all elements are equal in `b`.

The second part (`permut_sub`) of the post-condition of `mergesort1` follows the same lines and is exhibited at URL `http://jeanjacqueslevy.net/why3/sorting/`.

Permutations on arrays are defined by counting the number of occurrences for each value. Therefore the proof demands several properties of occurrences of values in sub-arrays. The proof is very natural except for a couple of redundant assertions, which ease the behaviours of automatic provers. In fact, many provers are involved in that second part, namely Alt-Ergo, Yices, CVC4 and Z3, thoroughly chosen thanks to the graphical interface of Why3. Moreover several manual transformations were needed, such as inlining and splitting of conjunctions.

The two lemmas about weakening the interval of `map_bitonic` are proved by 30 lines of easy and readable Coq (with *ss-reflect* package). It needs 4 extra lemmas (each with 7-line long Coq proof) `sorted_sub_weakening`, `dsorted_sub_weakening`, `sorted_sub_empty` and `dsorted_sub_empty` which states the weakening of intervals for (d)sorted subarrays and the (d)sorted status of empty subarrays. In fact these lemmas could also be proved by automatic provers, but there is a trade-off between expressing abstract properties and a detailed computable presentation. For instance, `bitonic` is defined with an existential connector, which is quasi equivalent to the end of automatic first-order provers. A more precise presentation with parameterizing the index at peak of the bitonic sequence would have reactivated the automatic methods. In fact, this trade-off is a big advantage of Why3. For instance, a verification condition can also be first attempted in Coq, and later proved automatically after simplifications.

4 Conclusion

The mergesort example demonstrates the versatility of the Why3 system. One can nicely mix automatic and interactive proofs. The multiplicity of solvers (SMT solvers and theorem provers) give high confidence before attacking an interactive proof, which is then reserved for conceptual parts. It is even possible to call back automatic provers from the interactive parts (not in the mergesort example, but it did happen in a version of quicksort to avoid a long Coq proof with numerous cases), but it requires to solve several technical typing subtleties. The system demands some training since it is a bit complex to manipulate numerous solvers and interactive proof-assistants. The WhyML memory model is rather naive since variables and arrays only allow single assignments. New variables or new arrays are created after every modification of their contents. Moreover arrays are immediately expanded to maps. Therefore it would be interesting to understand how far one can go with this memory model. It did not prevent from already building a gallery of small verified programs existing in the Why3 public release. The Frama-C project uses Why3 among other analyzers to build a verification environment for C programs written for small run-times or embedded systems.

```
let rec mergesort1 (a b: array int) (lo hi: int) =
  requires {Array.length a = Array.length b /\
        0 <= lo <= (Array.length a) /\ 0 <= hi <= (Array.length a) }
    ensures { sorted_sub a lo hi /\ permut_sub (old a) a lo hi }
  if lo + 1 < hi then
  let m = div (lo+hi) 2 in
  mergesort1 a b lo m;
  mergesort1 a b m hi;
  for i = lo to m-1 do
    b[i] <- a[i]
    done;
  for j = m to hi-1 do
    b[j] <- a[m + hi - 1 - j]
    done;
  let i = ref lo in
  let j = ref hi in
  for k = lo to hi-1 do
    if b[!i] < b[!j - 1] then
      begin a[k] <- b[!i]; i := !i + 1 end
    else
      begin j := !j - 1; a[k] <- b[!j] end
  done

let mergesort (a: array int) =
  ensures { sorted a /\ permut (old a) a }
let n = Array.length a in
let b = Array.make n 0 in
  mergesort1 a b 0 n
```

Figure 3: Mergesort on arrays

Finally it is interesting to notice how robust and intuitive is Hoare logic.

5 Acknowledgements

This short note is dedicated to Luca Cardelli for his 60th anniversary. Luca has always loved mixing theoretical topics and real computing systems. I hope this short note satisfies that criterion. I also thank Chen Ran for her help in multiple Why3/Coq proofs.

References

C. Barrett and C. Tinelli. CVC4, the smt solver. New-York University - University of Iowa. URL `http://cvc4.cs.nyu.edu`.

```
use map.Map as M
clone map.MapSorted as N with type elt = int, predicate le = (<=)

predicate map_eq_sub_rev_offset (a1 a2: M.map int int) (l u: int)
  (offset: int) =
  forall i: int. l <= i < u ->
                M.get a1 i = M.get a2 (offset + l + u - 1 - i)

predicate array_eq_sub_rev_offset (a1 a2: array int) (l u: int)
                                                  (offset: int) =
  map_eq_sub_rev_offset a1.elts a2.elts l u offset

predicate map_dsorted_sub (a: M.map int int) (l u: int) =
  forall i1 i2 : int. l <= i1 <= i2 < u -> M.get a i2 <= M.get a i1

predicate dsorted_sub (a: array int) (l u: int) =
  map_dsorted_sub a.elts l u

predicate map_bitonic_sub (a: M.map int int) (l u: int) =  l < u ->
  exists i: int. l <= i <= u /\ N.sorted_sub a l i /\
                                map_dsorted_sub a i u

predicate bitonic_sub (a: array int) (l u: int) =
  map_bitonic_sub a.elts l u

lemma map_bitonic_incr : forall a: M.map int int, l u: int.
  map_bitonic_sub a l u -> map_bitonic_sub a (l+1) u

lemma map_bitonic_decr : forall a: M.map int int, l u: int.
  map_bitonic_sub a l u -> map_bitonic_sub a l (u-1)
```

Figure 4: Theory add-ons for mergesort on arrays

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008. URL `http://alt-ergo.lri.fr/`.

F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/publications/boogie11final.pdf`.

L. de Moura and N. Björner. Z3, an efficient smt solver. Microsoft Research. URL `http://z3.codeplex.com`.

B. Dutertre and L. de Moura. The Yices SMT Solver. SRI. URL `http://yices.csl.sri.com`.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008. URL `http://hal.inria.fr/inria-00258384`.

R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

A. Tafat and C. Marché. Binary heaps formally verified in Why3. Research Report 7780, INRIA, Oct. 2011. `http://hal.inria.fr/inria-00636083/en/`.

```
let rec mergesort1 (a b: array int) (lo hi: int) =
    requires {Array.length a = Array.length b /\
        0 <= lo <= (Array.length a) /\ 0 <= hi <= (Array.length a) }
    ensures { sorted_sub a lo hi /\ modified_inside (old a) a lo hi }
  if lo + 1 < hi then
  let m = div (lo+hi) 2 in
    assert{ lo < m < hi};
    mergesort1 a b lo m;
'L2: mergesort1 a b m hi;
    assert { array_eq_sub (at a 'L2) a lo m};
    for i = lo to m-1 do
      invariant { array_eq_sub b a lo i}
      b[i] <- a[i]
      done;
    assert{ array_eq_sub a b lo m};
    assert{ sorted_sub b lo m};
    for j = m to hi-1 do
      invariant { array_eq_sub_rev_offset b a m j (hi - j)}
      invariant { array_eq_sub a b lo m}
      b[j] <- a[m + hi - 1 - j]
      done;
    assert{ array_eq_sub a b lo m};
    assert{ sorted_sub b lo m};
    assert{ array_eq_sub_rev_offset b a m hi 0};
    assert{ dsorted_sub b m hi};
'L4: let i = ref lo in
    let j = ref hi in
    for k = lo to hi-1 do
      invariant{ lo <= !i < hi /\ lo <= !j <= hi}
      invariant{ k = !i + hi - !j}
      invariant{ sorted_sub a lo k }
      invariant{ forall k1 k2: int. lo <= k1 < k ->
                                    !i <= k2 < !j -> a[k1] <= b[k2] }
      invariant{ bitonic b !i !j }
      invariant{ modified_inside a (at a 'L4) lo hi }
      assert { !i < !j };
      if b[!i] < b[!j - 1] then
        begin a[k] <- b[!i]; i := !i + 1 end
      else
        begin j := !j - 1; a[k] <- b[!j] end
    done
```

Figure 5: Proof of mergesort on arrays