Automatic predicate testing in formal certification

You've only proven what you've said, not what you meant!

Franck Slama
fs39@st-andrews.ac.uk

University of St Andrews, United Kingdom

Abstract. The use of formal methods and proof assistants helps to increase the confidence in critical software. However, a formal proof is only a guarantee relative to a formal specification, and not necessary about the real requirements. There is always a jump when going from an informal specification to a formal specification expressed in a logical theory. Thus, proving the correctness of a piece of software always makes the implicit assumption that there is adequacy between the formalised specification –written with logical statements and predicates– and the real requirements –often written in English–. Unfortunately, a huge part of the complexity lies precisely in the specification itself, and it is far from obvious that the formal specification says exactly and completely what it should say. Why should we trust more these predicates than the code that we've first refused to trust blindly, leading to these proofs? We show in this paper that the proving activity has not replaced the testing activity but has only changed the object which requires to be tested. Instead of testing code, we now need to test predicates. We present recent ideas about how to conduct these tests inside the proof assistant on a few examples, and how to automate them as far as possible.

Keywords: Formal certification, predicate testing, proof assistant

1 Introduction

One way to increase our confidence in software is to formally prove its correctness using a proof assistant. Proofs assistants enable to write code, logical statements and proofs in the same language, and offer the guarantee that every proof will be automatically checked. Many of them are functional programming languages, like Coq [2], Idris [3] and Agda [9], and others, like the B-Method [1] belong to the imperative paradigm. These different paradigms are internally supported by different logic. Systems like Coq, Idris and Agda are based on various higher order logics (CoC, a variant of ML and UTT respectively) and are realisations of the Curry-Howard correspondence, while the B-Method is based on Hoare logic. These different foundations lead to different philosophies and different ways to implement and verify a software, but all of them greatly increase the confidence on the produced software. However, these guarantees tend to be too

often considered as perfect, when they are in fact far from it. Knuth was – certainly ironically– saying "Beware of bugs in the above code; I have only proved it correct, not tried it". The reality is precisely that a proof is not enough. When we prove the correctness of a function, we only gain the guarantee expressed by the proven lemma, and nothing more.

Say we want to implement a formally verified sorting function for list of elements of type T, where T is ordered by a relation $\leq$. We can decide to define the sorting function with a "weak" type, like $sort\ :\ List\ T\ \rightarrow\ List\ T$, and to use an external lemma to ensure the correctness of the function. Which property does this function has to respect? First, the output has to be sorted, so we need to define this notion of being sorted, here as an inductive predicate :

```
data isSorted : {T:Type} -> (Order T) -> (List T) -> Type where
    NilIsSorted : (Tord : Order T) -> isSorted Tord []
    SingletonIsSorted : (Tord : Order T) -> (x:T)
                        -> isSorted Tord [x]
    ConsSorted : {Tord : Order T} -> (h1:T) -> (h2:T) -> (t:List T)
                -> (isSorted Tord (h2::t)) -> (h1 ≤ h2)
                -> (isSorted Tord (h1::(h2::t)))
```

The first and second constructor of this predicate say that [ ] and $[x]$ are sorted according to any order, and for any x. The third one says that a list of two or more elements is sorted if $h1 \leq h2$, and if the list deprived from its head is also sorted. In order to express that the result of *sort* is sorted, we can prove the following lemma: $sort\_correct : \forall\ (T : Type)\ (Tord : Order\ T)\ (l : List\ T),\ isSorted\ Tord\ (sort\ l)$. The problem with this specification is that it does not say anything about the content of the output. The function *sort* could just return the empty list [ ] all the time, it would still be possible to prove this correctness lemma. Here, the problem is that the function is underspecified, and it is therefore possible to write a senseless implementation, which is unfortunately provably correct. Only a careful reader could realise that the lemma *sort_correct* forgets to mention that the input and output list should be in bijection, meaning that everything which was originally in the input list should still be in the output, and that nothing else should have been added.

Another bug in the specification could have been to simply forget the third constructor *ConsSorted*. But things more nasty can happen. Imagine that this constructor would have been written with a typo, and that the condition ($h1 \leq h2$) would have been incorrectly written as ($h1 \leq h1$). Any list would be seen as "sorted", just because of this single typo, and the algorithm could for example return its input unchanged. One could object that when doing the proof of correctness, we should realize that the proof is being done too easily, without having to use the essential property that the output is being built such as any element in the list is always lower or equal than its next element. The reality is quite different because many effort are going in the direction of proof automation, which aims to let the machine automatically generate the proof for some kind of goals. For example, Coq has already a Ring prover [7] and many others automations, and Idris has been recently equipped with a hierarchy of provers

for algebraics structures [6]. There are even extensions to languages, such as Ltac [4] and Mtac [10] that aim to help the automation of tactics. The problem is that the machine is never going to find a proof "too easy", and will never report that something seems weird with the specification given by the user.

Thus, if we want to trust the proven software, we're now forced to believe that there is adequacy between the formal specification and the informal requirements. A switch has occurred. We used to have to trust code, but we now have to trust logical statements and predicates. But when the specification is too often as complicated as the code, why should we blindly believe in it, when we've first refused to blindly trust the code? The primary aim of this paper is to raise awareness on the adequacy concern, and to see how heterogeneous approaches, that mixes both proofs and tests, can help to go a step forward in the certification process, in the context of proof assistants based on type theory. More precisely, we :

- Show some basic approaches to the problem of underspecification (section 2)
- Present a new way to test the predicate in the proof assistant, by automatically generating terms, and we completely automate these tests. We also show how we can go a step forward by replacing these tests about the predicate by some proofs. (section 3)
- We discuss possible directions for making dependently typed programming languages more adapted to the testing of specifications (section 4).

We use Idris, a dependently typed programming language, but all the ideas that we present here can be applied to any proof assistant based on type theory. The running example that we use in this paper can be found online at `https://github.com/FranckS/ProofsAndTests`

2 Naive and usual approaches to the adequacy problem

When confronted to this problem of adequacy between the intuitive notion and the formalised one, a first possibility is to formalise the notion multiple times, with different predicates, and to prove that they are equivalent. With our example, that means that we need to find another formalisation $isSorted'$ of being sorted, and to prove the following lemma. $pred\_equiv : \forall\ (T : Type)\ (l : List\ T), (isSorted\ l) \leftrightarrow (isSorted'\ l)$. This approach aims at increasing the confidence in our formal definitions by assuming that if we've managed to define multiple times the same notion, then we've surely succeeded to define the notion we wanted. The biggest problem with this approach is to be able to find some alternative formalisations that are sufficiently different from the original one. Obviously, if the new formalisations are too similar to the original one (and in the worst case the new ones are just syntactical variants of the first one), then we won't gain any guarantee. The ideal would be to capture the same notion by using very different points of view, and we will show in section 3 an original approach for doing so.

In order to gain confidence in the formal specifications we write, another traditional approach is to test the predicate on some values. That consists in defining a few terms, usually by hand, for which we know if the predicate should hold or not, and to prove that the predicate effectively holds when it should, and that it does not when it should not. For example, with the predicate *isSorted* defined above, we can prove that it holds on the list [3, 5, 7] that we know sorted.

```
isSorted_test1 : isSorted natIsOrdered [3, 5, 7]
isSorted_test1 =
  let p1 : (3 <= 5) = tryDec (lowerEqDec natIsOrdered 3 5) in
  let p2 : (5 <= 7) = tryDec (lowerEqDec natIsOrdered 5 7) in
    ConsSorted 3 5 [7]
      (ConsSorted 5 7 [] (SingletonIsSorted _ 7) p2) p1
```

This test is a *test done by proof* : we show that the predicate holds on some specific value, here [3, 5, 7], by doing the proof. We can go a step forward by removing the need of doing these specific proofs by hand, because in this case, the predicate isSorted can be decided : there exists an algorithm that produces a proof of (*isSorted l*) if appropriate, or a proof of (*not* (*isSorted l*)) otherwise.

```
decideIsSorted : (Tord : Order T) -> (l:List T)
                 -> Dec(isSorted Tord l)
decideIsSorted Tord [] = Yes (NilIsSorted Tord)
decideIsSorted Tord [x] = Yes (SingletonIsSorted Tord x)
decideIsSorted Tord (h1::(h2::t)) with (lowerEqDec Tord h1 h2)
  | (Yes h1_lower_h2) with (decideIsSorted Tord (h2::t))
      | (Yes h2_tail_sorted) = Yes
                (ConsSorted h1 h2 t h2_tail_sorted h1_lower_h2)
      | (No h2_tail_not_sorted) = No [...]
  | (No h1_not_lower_h2) = No [...]
```

Now, in order to do *tests by proof*, we can simply run the decision procedure.

```
isSorted_test1' : Dec (isSorted natIsOrdered [3,5,7])
isSorted_test1' = decideIsSorted natIsOrdered [3,5,7]
```

And if we evaluate $isSorted\_test1'$, the system will answer Yes and a proof of *isSorted* [3, 5, 7], which means that the predicate has passed this test. With this technique, we can run semi-automatically a few tests on the predicate *isSorted*. It is semi-automatic in the sense that we still have to define by hand some terms that we know sorted or unsorted but we can let the machine produce the proof that the predicate holds or not on these specific values. This is not too bad –and this is in fact all of what is usually done, when it is actually done– but we would like to have a stronger guarantee, and not only that the predicate will coincide with our intuitive notion on a couple of tested terms.

3 Testing the predicate by automatic generation of terms

The key idea that this paper wants to convey is that it is often easier to generate examples of a notion than it is to precisely define it. We can operate an interesting change of point of view by generating the set of terms that we precisely wanted to describe with the predicate. With our example of sorted lists, that means that we need to define the same notion of being sorted, but this time with a definition example-based. Writing a function that produces all the sorted lists might be a bit more difficult than just giving a few examples, but this complicated function will have the main advantage of being so different from the predicate that if the two notions agree, then we will have gain a great confidence on the predicate. We decide to use coinduction and the type Stream (a coinductive version of lists, potentially infinite) in order to generate –with what we call a generator– all the sorted lists of size n.

$generateSortedList : (T : Type) \to (recEnu : RecEnum\ T) \to (Tord : Order\ T) \to (n : Nat) \to Stream\ (List\ T)$.

To do so, the type T needs to be recursively enumerable, which means that there must exist a computational map $Nat \to Maybe\ c$ with the condition that this map is surjective, which means that any value of type c should be hit at least once by the map : $map\_is\_surjective : (y : c) \to (x : Nat ** (computableMap\ x = Just\ y))$.

We want to check that this function and the predicate coincide. Since the predicate *isSorted* is decidable with $decideIsSorted$, we can automatically check whether the generated sorted lists of size n are automatically sorted (according to *isSorted*) by running this decision procedure. But since there can be infinitely many generated sorted lists of size n when $n > 0$, we will only check that the generator and the predicate coincide on a finite observation[1] of the resulting stream. The key point is that this observation can be arbitrary big, and the bigger it is, the better the guarantee is about *isSorted*.

We show how the observation is made on an example. Let T be a type that only contains three constant values A, B and C. Since T is finite, it is therefore obviously recursively enumerable. We define on it the strict order $A < B < C$. Let's automatically generate the first m sorted lists of T, of size n, by unfolding m times the result of *generateSortedList*.

```
testGenerator : (m:Nat) -> (n:Nat) -> Maybe(Vect m (List T))
testGenerator m n =
        let x = generateSortedList T TisRecEnu TisOrdered n
            in unfold_n_times x m
```

We can ask for the first 8 sorted lists of size 4 by evaluating *testGenerator* 8 4:

```
Just [[A, A, A, A], [A, A, A, B], [A, A, A, C], [A, A, B, B],
      [A, A, B, C], [A, A, C, C], [A, B, B, B], [A, B, B, C]]
      : Maybe (Vect 8 (List T))
```

[1] A finite observation of a stream, also called approximation at rank m of a stream, is a vector of size m that has the same m first elements than the stream

Now, instead of simply generating the first m sorted lists, we run the decision procedure on all of these m tests in order to know if the predicate and the generator agree on this portion. The result will be a vector m booleans.

```
testSorted : (m:Nat) -> (n:Nat) -> Vect m Bool
testSorted m n =
  let x = generateSortedList T TisRecEnu TisOrdered n in
    let y = Smap (\l => let res = decideIsSorted TisOrdered l in
                            case res of
                              Yes _ => True
                              No _ => False) x in
        unfold_n_times_with_padding y m True
```

And we can inspect the result of running the first 8 tests of size 4 by evaluating *testSorted* 8 4.

```
[True, True, True, True, True, True, True, True] : Vect 8 Bool
```

When we want to test *isSorted* on a large number of tests, we might not want to inspect manually the result of each test. We can write a function $testSorted\_result : (m : Nat) \rightarrow (n : Nat) \rightarrow Bool$ that calls *testSorted* m n and does the boolean *And* on each element of the resulting vector. Now, we can for example test the predicate on the first 50 sorted lists of size 9 by running *testSorted_result* 50 9 and if we do so we get the overall result $True$ which means that the predicate agrees with the generator on all these 50 tests.

However, if the predicate *isSorted* has been incorrectly written, then the result of this test might inform us that there's something wrong with the formal specification. For example, if we've forgotten the third constructor *consSorted* in the definition of *isSorted*, then the result of (*testSorted* 8 4) will be $False$, which means that at least one of the produced list is not seen as sorted according to *isSorted*, and we will therefore know that this predicate does not capture our intuitive notion of sortedness.

In order to go a step forward, we can decide to replace the tests on the predicate by proofs. Instead of testing the predicate on a finite subset of all the generated sorted lists as we just did, we can try to prove that any of the automatically generated sorted list is provably sorted according to *isSorted*.

```
generated_implies_pred_holds : {T:Type}
   -> (recEnu:RecEnum T) -> (Tord : Order T) -> (n:Nat)
   -> (All (generateSortedList T recEnu Tord n)
           (\l => isSorted Tord l))
```

Proof. By induction on n. When n is zero, there is only one sorted list generated, which is the empty list, and we know that the empty list is sorted thanks to the constructor $NilIsSorted$. When n is some successor $(S\ pn)$, we know by using recursively the lemma on the smaller value pn that all sorted lists of size pn are sorted according to *isSorted*. Since the $Stream$ of all sorted lists of size $(S\ pn)$ has been made from the $Stream$ of all sorted lists of size pn by adding to all of

them –on the head position– an element lower or equal to their respective current heads, we know that the property has been preserved at the higher rank. □

This lemma has the advantage of not requiring the predicate to be decidable, whereas this was needed when we automatically tested the predicate on a finite observation. However, one could object that this lemma is itself built by using a predicate, *All*, and that we can't necessary trust blindly such a specification. The answer is that no guarantee is perfect, and all we can do is to add some guarantees, but there is necessarily always something to trust. Moreover, this new kind of specifications and proofs –about the predicate itself– uses more primitive components like streams and the predicate *All*, and these components can be provided once and for all. If they are part of some standard library being used intensively, there is very low risk that they do not capture the desired semantic.

4 Conclusions and future work

In this paper we've presented a new way to test a predicate based on an automatic generation of terms that should have the desired property. This adequacy between the predicate and the generator helps to gain confidence in the predicate. The technique presented on section 3 was based on a finite observation of the terms generated in endless amount, processed by the decision procedure. We have also shown on an example how these tests can be automated.

We haven't been able to find much work done in the direction of predicate testing in the environment of proof assistants, but we strongly believe that this aspect is crucial, as there is absolutely no point to prove the "correctness" of a function relatively to a bad specification. One could however question why formal certification is needed at all, if after going through all the effort of interactive theorem proving we still have to test the specification itself, and also need tools to support it. We believe that the process of formalisation helps to uncover things that were missing or weakly specified in the informal requirements. For example, the development of the formally verified compiler CompCert [8] has contributed to brought to light many under-specified behaviours in the specification of the C standard.

The machinery developed for the running example presented in this paper is extremely specific and it is not reasonable to believe that this work should and could be done for every formal specification. What it shows is that we really need to explore how proof assistants themselves could help to gain confidence about predicates and logical formulae. A possible direction could be to build execution engines for formal specifications written in dependent type theories. Such a system would take in input a predicate and would produce some of the terms that make this predicate hold. The ideal would be to have a query system where one could ask the system to try to look if some specific terms are captured by the predicate. Since the problem of finding proof is undecidable in the general case in higher-order logics (that's also the case in first-order logic), such a system

can't be complete and entirely automatic, and therefore the user would have to help the system at times.

Another possible direction is to equip proof assistants with many robust and generic concepts (like being sorted) once and for all. That would save the user from the error prone activity of writing many primitive logical properties. Equipping proof assistants with many generic and useful concepts already available, like bricks ready to be assembled, is another current challenge to make proofs assistants really usable.

References

1. Abrial, J., et al.: The b-method. In: Prehn, S., Toetenel, W.J. (eds.) VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials. Lecture Notes in Computer Science, vol. 552, pp. 398–405. Springer (1991), http://dx.doi.org/10.1007/BFb0020001
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004), http://dx.doi.org/10.1007/978-3-662-07964-5
3. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23, 552–593 (September 2013), http://journals.cambridge.org/article_S095679681300018X
4. Delahaye, D.: A proof dedicated meta-language. Electr. Notes Theor. Comput. Sci. 70(2), 96–109 (2002), http://dx.doi.org/10.1016/S1571-0661(04)80508-5
5. DeMillo, R.A., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. Commun. ACM 22(5), 271–280 (1979), http://doi.acm.org/10.1145/359104.359106
6. Franck Slama, E.B.: Automatically proving equivalence by type-safe reflection (draft under consideration). Journal of Functional Programming (2016), https://fs39.host.cs.st-andrews.ac.uk/publications/paper_Slama_Brady_JFP.pdf
7. Gregoire, B., Mahboubi, A.: Proving Equalities in a Commutative Ring Done Right in Coq. In: Theorem Proving in Higher Order Logics (TPHOLS 2005). pp. 98—-113 (2005), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.3041
8. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C semantics: CompCert and the C standard. In: ITP 2014: Fifth conference on Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 8558, pp. 543–548. Springer, Vienna, Austria (Jul 2014), https://hal.inria.fr/hal-00981212
9. Norell, U.: Dependently typed programming in agda. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Lecture Notes in Computer Science, vol. 5832, pp. 230–266. Springer (2008), http://dx.doi.org/10.1007/978-3-642-04652-0_5
10. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: a monad for typed tactic programming in coq. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 87–100. ACM (2013), http://doi.acm.org/10.1145/2500365.2500579