CIS 670 Final Report

Generic Programming for Generating Generators

Kenny Foner Leonidas Lampropoulos

December 19, 2016

1 Introduction

Property-based random testing (PBT) is a well-known technique for gaining a probabilistic confidence in software correctness. However, its use does not always come for free. For each data type which occurs at the interface of a program to be tested, someone must write code to generate random elements of that type, print elements of that type as a string, and "shrink" elements of that type into structurally smaller components. The entirety of these functions is determined solely by the structure of the data type in question; that is to say, all this code is boilerplate. We should like to write a *deriver* for each function—show, generate, shrink—only once, and apply that deriver to any data type we please. This is possible, but without the appropriate abstractions for generic programming, it becomes untenably gnarly.

In this work, we provide precisely such abstractions, in the form of a combinator library. We address this issue in the context of assisting PBT development within the Coq proof assistant: our library is in OCaml, interfacing with Coq via its OCaml plugin interface. We demonstrate the application of our library to create generic derivers of the PBT type classes for all polynomial recursive data types in Coq. Additionally, we highlight the work of one of our users, who used our library to generically derive Coq-checkable proofs of correctness for those same PBT type classes.

2 Property based testing (in Coq)

Our work is toward the betterment of the QuickChick [2] PBT framework. QuickChick is a clone for Coq of Haskell's QuickCheck [1] under development as a collaboration between Penn and Inria. Before proceeding any further, it's important to clarify precisely how property-based random testing works on Coq—its "look and feel" as well as its functionality.

Property based testing receives as input an executable specification of a program, *generates* random inputs and tests their validity against the given spec. If a falsifying input is found, it is *shrunk* to a minimal counterexample and *shown* to the user. For example, consider a standard implementation of binary trees in Gallina, accompanied by recursive equality predicate.

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.

Arguments Leaf {A}.
Arguments Node {A} _ _ _.

Fixpoint eq_tree (t1 t2 : Tree nat) : bool :=
  match t1, t2 with
    | Leaf, Leaf => true
    | Node x1 l1 r1, Node x2 l2 r2 =>
      beq_nat x1 x2 && eq_tree l1 l2 && eq_tree r1 r2
    | _, _ => false
  end.
```

Suppose we have the following function, designed to mirror a tree, which recursively exchanges each left subtree with the corresponding right subtree. We expect that mirroring a tree twice returns the original tree, but note that the following definition contains an (artificially injected) bug; a typo in the recursive call to mirror.

```
Definition mirrorSpec (t : Tree nat) :=
              eq_tree (mirror (mirror t)) t.

Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
  match t with
    | Leaf => Leaf
    | Node x l r => Node x (mirror l) (mirror l)
  end.
```

Ideally, a Coq user should be able to invoke QuickChick directly at this point, to discover a counterexample to the desired property, before embarking in a fruitless proof effort:

```
QuickChick mirrorSpec.
```

This should report a counterexample as in the following:

```
Node (0) (Node (0) (Leaf) (Leaf)) (Leaf)

*** Failed! After 2 tests and 1 shrinks
```

Sadly, life is not that pretty. In order for the QuickChick command to work for a predicate `A -> Bool`, like `mirrorSpec`, it requires a generator for elements of type `A` (to generate the random inputs), a shrinker (to reduce a failing input to a minimal counterexample like above), as well as a printer (to report that minimal counterexample). While QuickChick provides instances for many built-in datatypes of the typeclasses that correspond to these operations, users must write such instances manually for user defined datatypes.

However, most ADTs admit a straightforward generator/shrinker/printer that is good enough for most intents and purposes with a similar generic structure. For example, a simple `Show` instance for `Tree`s in Gallina could be the following:

```
Instance showTree {A} `{_ : Show A} : Show (Tree A) :=
  {| show :=
       let fix aux t :=
         match t with
           | Leaf => "Leaf"
           | Node x l r =>
             "Node (" ++ show x ++ ") ("
             ++ aux l ++ ") (" ++ aux r ++ ")"
         end
       in aux
  |}.
```

Given a `Leaf`, our printer only outputs the string "Leaf". Given a `Node` it would output "Node", accompanied by parenthesized printings of its children. It is easy to see, that such boilerplate could be generated automatically, as it is entirely determined by the structure of the type in question.

Currently, QuickChick provides a Coq plugin to derive such instances by manipulating Coq terms at the OCaml level. Naturally, implementing such a thing directly against the Coq internals API was an exercise in frustration, as the internal representation of Coq terms in OCaml is fraught with extraneous information and hidden complication (as one of the authors—having written this code himself—can testify with great fervor). Figure 1 shows from 10,000 feet the OCaml code necessary to generically derive the `Show` instances for polynomial recursive data types. What it doesn't show is how this code is unstructured, utterly unmaintainable and completely inextensible— out of necessity, due to the lack of abstraction afforded by working against the raw Coq internals API.

3 Our API

3.1 Datatype Representation

We start by describing our (relatively standard) representation for simple coq inductive datatypes.

```
type dt_rep = ty_ctr * ty_param list * ctr_rep list
```

A datatype representation is simple a type constructor, a list of its type parameters and a list of representation of its constructors. Here, a type construct (`ty_ctr`) and a type parameter (`ty_param`) is simply an opaque wrapper around corresponding coq internals.

```
type ctr_rep = constructor * coq_type
```

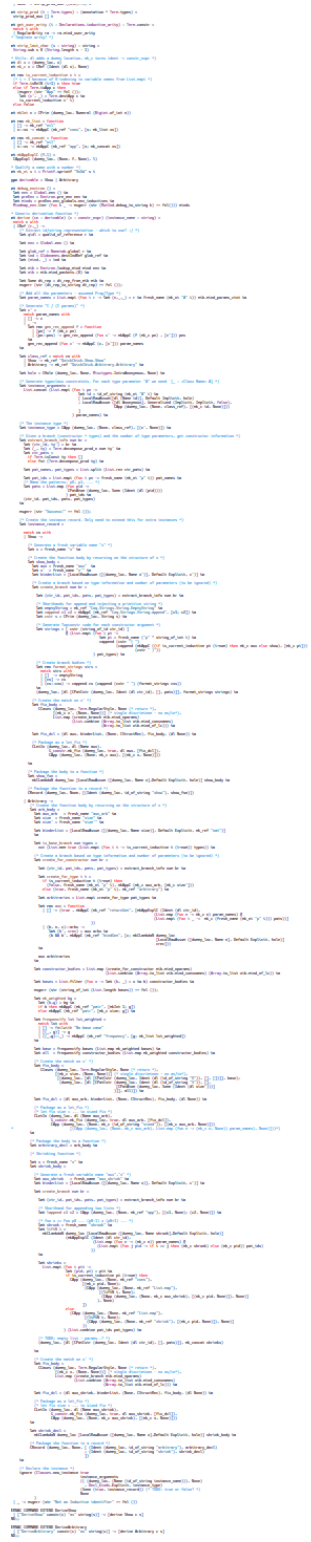

Figure 1: Previous OCaml code for deriving QuickChick typeclass instances, prior to the introduction of our library to the QuickChick codebase

A constructor representation contains an opaque reference to the constructor name as well as a representation of its corresponding type.

```
type coq_type =
  | Arrow of coq_type * coq_type
  | TyCtr of ty_ctr * coq_type list
  | TyParam of ty_param
```

Finally, we represent our simple subset of Coq types as either arrows between Coq types, a type constructor applied to a list of types or a type parameter. All of the type parameters appearing in a type are universally quantified at the top level, which is captured in the type parameter list in the datatype representation. Although this representation could be collapsed down to a more succinct view, we found that pragmatically, separating the `Arrow` type constructor from all others was much more desirable to end users. For most applications we considered, we wanted to give special treatment to function types, and so it made sense to represent it distinctly, rather than forcing the user to continually search for it within a `TyCtr`.

Internally, of course, Coq uses quite a different format to describe its types. This format includes not only a slew of information that the user of our library likely does not care about, but also a great number of syntactic cases we do not yet cover in our library. In particular, Coq's support for arbitrary dependent types means that any *term* can appear in a type—our representation does not support this, nor does it support nested quantification, non-parametric type indices, or a variety of other features. We hope to expand to incorporate these things in the future, as we discuss later.

3.2 A DSL for building terms

The second component of our project is a DSL for abstracting away from Coq internals when generating terms. We are in the process of designing and implementing the DSL combinators. For example, the original QuickChick derivation code for a simple `let-fix` declaration inside a `Show` instance contains the following:

```
let aux = fresh_name "aux"  in
let x'  = fresh_name "x'" in
let binderList =
    [LocalRawAssum ([(dummy_loc, Name x')],
                    Default Explicit, c')] in

let fix_dcl = (dl aux, binderList, (None, CStructRec),
               fix_body, (dl None)) in

CLetIn (dummy_loc, dl (Name "aux"),
  G_constr.mk_fix (dummy_loc, true, dl aux, [fix_dcl]),
  CApp (dummy_loc, (None, mk_c aux), [(mk_c x, None)]))
```

Understanding what each term in the expression above does and dealing with the particularities of different binding forms requires diving into the (not really documented) Coq internals. Moreover, even if we got this code to work (which we did!) maintaining or changing it is virtually impossible. Consider instead the following `gRecIn` combinator:

```
val gRecIn : string -> string list ->
             (var * var list -> coq_expr) ->
             (var -> coq_expr) ->
             coq_expr
```

First, the generic programmer needs to specify the name of the function and its arguments (although they will be made fresh internally to avoid capture). To construct the body of the fixpoint, the programmer should have at her disposal opaque symbols for the function and the arguments and use the various DSL combinators. By using an opaque representation of the various bound terms, we aim to guarantee that every term produce via our combinators is well scoped. Similarly for the body of the let, we should only be able to access the function symbol, not its arguments.

This leads to an important property of our library, and an important design tradeoff we made while constructing it: while we do not guarantee that all generic programs written in our framework produce only well-*typed* terms, we do guarantee that all programs written solely against our interface will produce well-*scoped* terms [1]. We ensure this by making sure that every combinator which requires the programmer to specify what to do with names, the names are given abstractly. For instance, when the programmer specifies what should occur in the body of the `let rec`, she gives a function of type `(var -> coq_expr)`. Recalling that `var` is an opaque, abstract type, we know that this function must therefore be parametric in its argument. Thus, by strategically using abstract types, we can enforce a kind of poor-person's parametricity, resulting in an interface much like parametric higher-order abstract syntax that can enforce well-scoped-ness, but not well-typed-ness.

Why not make sure that we can only produce well-typed terms from derivers written in our DSL as well? For one thing, OCaml's type system is susbtantially less powerful than Coq's, so in order to enforce that only well-typed terms are ever produced by OCaml programs written in our library, we would have to encapsulate all of Coq's type system in it via some manner of plugin—and because of the design of Coq, this would be quite difficult, to say the least.

Beside these technical limitations, we also wish to hit a sweet spot between ease-of-use for the programmer, and enforced correctness. In our opinion, requiring incredibly complex programming with dependent types would not make it quick and easy for programmers to write the kind of derivers we wish them to, at least given the current state of research in dependently typed generic programming. We feel that assisting the programmer in generically writing

[1] This presumes that all such generic programs are written in the purely functional fragment of OCaml, as the use of references in particular can defeat this guarantee.

well-scoped Coq code strikes a happy medium between practicality and provable correctness. In practice, what this means is that if the programmer makes a mistake when implementing a generic deriver using our framework, her code will fail at compile-time in Coq (perhaps only for some particular data types but not others). It's important to note that regardless, type-safety from the perspective of Coq is still preserved—there is no way to use our framework to introduce a bogus type equality to Coq.

Let us return to the library itself. As an example usage, consider the `let-fix` declaration using our combinators:

```
gRecIn "aux" ["x"] (fun aux [x] -> ...) (fun aux   -> ...)
```

In definitions like this, we aim to mirror the syntax of the Coq source being generated as much as possible, so as to make the mental tranlation burden for the user of the library as light as possible.

Another example of such a combinator is pattern matching. In our current design, `gMatch` has the following type:

```
val gMatch : coq_expr ->
    (constructor * string list * (var list -> coq_expr)) list ->
    coq_expr
```

This combinator takes a Coq expression (the discriminee) and a list of branches, each of which is a constructor (opaque, taken from our datatype representation) a list of strings to name the patterns, as well as a body for each branch with access to the particular pattern variables of that constructors, and produces a Coq term corresponding to the entire expression.

4 A worked example

To illustrate how one might use the library, we now present in its entirety the implementation of the generic deriver for the `Show` typeclass. We will then walk through it step-by-step to explain how it is constructed.

```
let show_body x =

  let branch rec_name (ctr,ty) =

    (ctr, generate_names_from_type "p" ty,
    fun vs -> str_append (gstr (constructor_to_string ctr ^ "  "))
        (fold_ty_vars
            (fun _ v ty' ->
              str_appends [ gstr "( "
                          ; gapp (if iscurrenttyctr ty'
                                  then gvar rec_name
                                  else ginject "show")
                              [gvar v]
                          ; gstr " )"
```

```
                        ])
            (fun s1 s2 -> str_appends [s1; gstr " "; s2])
            emptystring ty vs))
  in

  gRecFunIn "aux" ["x'"]
    (fun (aux, [x']) ->
      gMatch (gVar x')
        (List.map (branch aux) ctrs))
            (fun aux -> gApp (gVar aux) [gVar x])
in

let show_fun = gFun ["x"] (fun [x] -> show_body x) in
gRecord [("show", show_fun)]
```

Because OCaml does not have `where`-clauses, the nested structure of this definition is presented in a somewhat upside-down manner. As such, we shall examine it from the bottom up.

4.1 Generically building typeclass instances

4.1.1 The `Show` typeclass

A typeclass in Coq is merely a record mapping method names (as field names) to their implementations (as values of that field). In the case of the `Show` typeclass, we need to produce a record with exactly one field, named `show`, which is bound to a function of type `(T -> string)` for whatever type `T` we are deriving `Show` for.

```
gRecord [("show", show_fun)]
```

To do this (above), we invoke the `gRecord` function, which takes a description of the contents of a record and returns a Coq term corresponding to the actual record.

We then define the function which is bound to the `show` method.

```
let show_fun = gFun ["x"] (fun [x] -> show_body x) in ...
```

The function takes one argument, `x`, and has a body equal to whatever `show_body` is (given that abstract variable). It's worth noting here that one piece of lightweight dependent typing would be helpful in improving the library interface: it should always be the case that the list of concrete names passed to `gFun` be the same length as the list of abstract names provided to its function argument. We do not statically describe or enforce this, and it would be nice to use length-indexed lists to do this in future.

So what does the body of the `show` function consist of? Well, it's a (potentially) recursive function of one parameter. We define it as below.

```
  gRecFunIn "aux" ["x'"]                    (* 1 *)
```

```
(fun (aux, [x']) ->                    (* 2 *)
  gMatch (gVar x')                     (* 3 *)
    (List.map (branch aux) ctrs))      (* 4 *)
(fun aux -> gApp (gVar aux) [gVar x])  (* 5 *)
```

By lines, we:

1. define a recursive function `aux` of one argument `x'`
2. ...
3. which matches on that argument,
4. and has a case for each constructor of the data type, where the RHS of the case is determined by the `branch` function (discussed below)
5. and then apply that function to the variable x (passed in from above).

There are two present unknowns in the above code: the list `ctrs`, and the function `branch`. `ctrs` is provided to us by initialization code not shown in the example which uses our library function `coerce_reference_to_dt_rep`. This function takes a native Coq internal type and gives us back a `coq_type`, as well as a variety of other information about it, including a list of its constructors, here bound as `ctrs`.

As for the `branch` function which defines what to do for each possible constructor of the data type we are `show`ing: its logic is relatively simple as well, but let's break it down into pieces to clarify just how it works.

```
let branch rec_name (ctr,ty) =
  (ctr, generate_names_from_type "p" ty,
  fun vs -> str_append (gstr (constructor_to_string ctr ^ "  "))
      (fold_ty_vars
          (fun _ v ty' ->
            str_appends [ gstr "( "
                        ; gapp (if iscurrenttyctr ty'
                                then gvar rec_name
                                else ginject "show")
                            [gvar v]
                        ; gstr " )"
                        ])
          (fun s1 s2 -> str_appends [s1; gstr " "; s2])
          emptystring ty vs))
```

First, for our library, any case of a match is specified by a triple of a constructor, a list of concrete bound pattern variables names, and a function from (opaque) variables to an RHS. The first two elements of this triple are quick to write:

```
(ctr, generate_names_from_type "p" ty, ...)
```

Here, `generate_names_from_type` creates the appropriate number of names, prefixing them with a given string and suffixing them with unique numbers to freshen them.

For each RHS of the match, we need to construct a function from opaque variables to resultant Coq expressions. Recall that `show : T -> string` for some `T`—so ultimately, we need to return a string. Now we are at the meat of the problem: given a value built of a particular constructor of a particular data type, how do we convert it into a string representation generically?

Let us consider the particular case of the `Tree` type discussed earlier. When we want to `show` a `(Node x l r)`, we need to first make a string for the constructor—easy!—then prefix it to the space-interpolated concatenation of the parenthesized showings of each of the `Node`'s arguments. But how to `show` each of these arguments? If, in the case of `x`, such an argument is not another `Tree`, we merely appeal to the `Show` instance of that type. On the other hand, if—as in the case of `l` and `r`—an argument *is* another `Tree`, we need to make a recursive call to *this* `show` function. This distinction is necessitated by the way that Coq's typeclass mechanism functions: the existence of an instance for a given type is not accessible within the definition of that very instance.

All the information we need to generically define such a show-function is present in the type of the data constructor whose case we are defining. If we consider such a type `(A -> B -> C -> T)` as a list of types to the left of an arrow `[A, B, C]`, then we can fold across this list to obtain the desired function.

```
(fold_ty_vars
    (fun _ v ty' ->
      str_appends [ gstr "( "
                  ; gapp (if iscurrenttyctr ty'
                          then gvar rec_name
                          else ginject "show")
                      [gvar v]
                  ; gstr " )"
                  ])
    (fun s1 s2 -> str_appends [s1; gstr " "; s2])
    emptystring ty vs))
```

This produces an expression which, for each variable bound by the pattern match, calls the appropriate `show` function (either recursive or non-recursive) and wraps the result in parentheses, concatenating the results by interpolating spaces.

And there you have it: in only a few more lines than the non-generic `Show` instance, we have defined a generic deriver for `Show` instances that works on all polynomial recursive types.

4.1.2 The `Arbitrary` and `Shrink` typeclasses

Arbitrarily generating random elements of a type is slightly more intricate than merely showing elements of a type as strings, but it can be similarly implemented in our framework. At a high level, the key complication is that all generated code must be monadic, since generators exist in the `Gen` monad. Other than that, the code follows similar patterns to that of `Show`. In QuickChick's version

of the `Arbitrary` typeclass, the `arbitrary` function is additionally passed a maximum size for the object it is to generate—this is necessary to pass Coq's termination checker. Thus, the derivation of an `Arbitrary` instance proceeds as follows.

1. We create a recursive function which takes a size as an argument.
2. We match on the size given. If the size is zero, we generate one of the base cases of the data type; if it is not zero, we pick randomly between either generating a base case, or generating arguments to a random non-base-case constructor.

For the `Shrink` typeclass, we employ a very similar approach to `Show`: we match on the constructor, and for each possible constructor, we either return any of its arguments, or we return it, but with a shrunk argument somewhere.

5 Current status, future work

At present, our implementation is capable of handling any polynomial recursive type in Coq. This excludes such types as mutually inductive types, nested types, existential types, and more generally, types with dependencies. For many of these categories of types, there is no large technical difficulty in accomodating them within our system—just implementation effort required. We eventually aim to support a large fraction of Coq's types, including many forms of dependent type, especially targeting types which are similar in structure to refinement types—e.g. `(t : Tree A | isBST t)`. In the more near-term, we aim to support nested types and mutual induction.

Our implementation of the generic deriving library consists of about 500 lines of OCaml. The auto-deriving plugin built atop our library for property-based testing consists of only an additional 200 lines of OCaml, and derives all three of `Show`, `Arbitrary`, and `Shrink`.

6 Evaluation

One of our biggest initial claims in the creation of this library is that it would make life substantially easier for the programmer who wishes to create generic derivers. While the authors feel satisfied that we have accomplished this goal, we are hardly objective evaluators. However, we have *users*! Zoe Paraskevopoulou (who has worked on QuickChick[3], but not the internals of our library) used our library to build an automatic *proof* deriver to accompany the existing PBT typeclass derivers.

For every type for which we want to derive an `Arbitrary` instance, a variety of lemmata need to be proven in order for us to be confident that the instance behaves as expected. In particular, we want to know that the generator can generate all elements of the type; that is, that the operational semantics of the generator align with the set-denotation of the type. Automating such a proof

using Coq's Ltac scripting language is difficult because the structure of the proof is determined by the shape of the types and data constructors involved, which aren't easily manipulable in the appropriate ways within Ltac. On the other hand, this is exactly the information exposed by our API. The generic deriver for such a proof—using our library—is only about 200 lines of OCaml, and it works for a wide variety of types. By contrast, just a single proof *produced* by this deriver can often be almost as long or longer than the source code of the deriver itself. We feel that this is strong evidence to support our claims of ease-of-use and high power-to-weight ratio.

7 And one more thing...

The code is available now on GitHub, so you can try it out for yourself!
`https://github.com/QuickChick/QuickChick`

Enjoy!

References

[1] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000. URL: `http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf`.

[2] Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. QuickChick: Property-based testing for Coq. The Coq Workshop, July 2014. URL: `http://prosecco.gforge.inria.fr/personal/hritcu/talks/coq6_submission_4.pdf`.

[3] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *6th International Conference on Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. URL: `http://prosecco.gforge.inria.fr/personal/hritcu/publications/foundational-pbt.pdf`.