A Generator for Learn OCaml Exercises

Steven Thephsourinthone

February 28, 2019

1 Introduction

The Learn OCaml [1] platform provides a nice, quick, and easy way for a student to learn functional programming in OCaml. As an in-browser tool, it bypasses the necessity of a student having to install OCaml, which is a huge beneft in itself since the installation process can get quite messy, depending on the student's operating system. Learn OCaml gives the student access to an OCaml interpreter via Learn OCaml's integrated top level allowing the student to easily try and use OCaml. Additionally, Learn OCaml provides an exercise platform which allows the instructor to create interactive exercises which provide the student with instant feedback through a report that the student can look through and see where they may have made errors, highlighting common ones like type errors, while showing the student the test cases used by the grader.

Another advantage to having such a platform is that the instructor of such a course may delegate assignments to the Learn OCaml platform and free up time for the instructor and teaching assistants that they may better use to help students with their understanding of the course material. This is a direct benefit to our Programming Languages and Paradigms course at McGill, which has grown to hundreds of students, making manual grading rather cumbersome. In the past, grading scripts have been used, but the grading system used by Learn OCaml is much simpler to use.

In order to facilitate the use of the Learn OCaml platform, it would help to have a tool which can generate all the files that an exercise requires from a single solution file. The creators of Learn OCaml have provided a tool, Learn OCaml autogen, but it is somewhat lacking in the most important area, the automatic generation of the grader. In this report, I present my own generator which includes the automatic generation of samplers, which generate values for test cases, as well as a more thorough generator for the test file, using only a single, annotated solution file.

2 Background

2.1 OCaml MOOC

First concieved in 2014, the OCaml Massive Open Online Course (MOOC) has been running one a year since 2015. The goals of the course were to familiarize the students with three key aspects of languages from the ML family, namely functional programming, static typing with type inference, and algebraic data types. Thus, the point of the course was not to rank students based on their marks, but to ensure that all who completed the course demonstrated some level of mastery over these aspects and could write medium-sized OCaml programs.

With a massive online course comes a number of challenges, like the wide variety of students, having different backgrounds, the learning environment, and how exactly to evaluate the students. It was decided that there were two key points that should be addressed: one, that every student had the same development and exercise environment, and two, that the students could recieve quick feedback through automatic evaluation. In order to make the learning enviroment consistent, a browser-based solution was chosen, which alleviated the burden of having to install OCaml from the students, and gave them quick and easy access to an OCaml interpreted that they could use to practice. This led to the development of the Learn OCaml platform.

2.2 Learn OCaml

2.2.1 Exercise Environment

In the exercise environment, the student is offered many helpful tools for writing OCaml programs. There is access to an OCaml toplevel intepreter, which allows the student to experiment with various OCaml functions and a text editor in which the student can write code, which can then be loaded into the toplevel that lets the student self-evaluate a bit the correctness of their code. The student can also check their code for syntax and type errors, and finally, in an exercise, they may submit their code for grading, which generates a lengthy report on the student's code that they can examine to write corrections if necessary.

On the instructor's end, setting up exercises to evaluate the students' progress is a fairly simple process, requiring six source files to define each exercise. First, the instructor must provide a descripton of the exercise, which was placed as an HTML fragment in the file `descr.html` (and now a Markdown file is also acceptable). A metadata file, `meta.json`, should also be provided, giving the student information on things like difficulty and contact information they could use to get help. Next, `prelude.ml` and `prepare.ml` must be provided which load the necessary preliminaries into the student's exercise environment, like type declarations or helper functions, with the contents of

`prepare.ml` being hidden from the student, allowing for things like shadowing library functions to prevent the student from using them as their solution (for example, using OCaml's `List.rev` as a solution to a list reversal question). Then, `template.ml` loads a template for the exercise into the student's exercise environment. A `solution.ml` file is required to evaluate the student's code against. Finally, a `test.ml` file is needed, which contains a test suite for evaluating the student. These format of these files lends itself quite nicely to automatic generation from a single file, the solution, which will be explored later on.

2.2.2 Graders

Learn OCaml provides the instructor with a variety of predefined graders that can be composed to evaluate an exercise. The instructor may evaluate a student's code against a solution via test cases or even check for syntatic constructions used in the student's code. The grading system uses a combination of static typing and forced pattern matching to catch most errors at compile time, which is a stark difference to other systems that first test the grader on sample submissions to gain some level of confidence in the grader, and then perhaps still find bugs after deployment. With the high level of confidence Learn OCaml's grader gives, one can update the grader to change test cases or change grading policies that may have been incorrect without worry. The exercise writer need not worry about implementing this when writing graders as a number of high level functions are available which parameterize this lower level grader code.

2.2.3 Cheating

As with any course, there is a possibility of cheating. First, the Learn OCaml exercise environment lives entirely within the student's browser, it is entirely possible that a dishonest student may reverse engineer the exercise environment to have the grader report the maximum possible score. This turned out to be a non-issue in the OCaml MOOC, however, as a elective course whose goal leans more towards teaching, the students who take the course may be far less interested in circumventing the course's grading scheme. This may turn out to be more of an issue in a mandatory course setting, however it is also quite easy to run the graders on the student's code offline to verify the marks the online grader gave them.

A second avenue of cheating lies in the test suite. If the test suite used the same test cases every time, a student may tailor their code specifically to get the results desired from these few test cases. To alleviate this, all test cases are randomly generated each time the student has the exercise environment evaluate their code.

3 Preliminaries

Before the student is dropped into the exercise environment, information is loaded from three files, `meta.json`, `prelude.ml`, and `prepare.ml`, here referred to as the *preliminaries*. In this section, I take some cues from the official autogen project, and as they do, we use a ppx extension of the form `let%{file}`, which annotate declarations with their intended destinations, for example, declarations starting with `let%meta` will put the information into the appropriate fields in the `meta.jsob` metadata file. The generation of these files is handled in `preliminaries.ml`.

3.1 meta.json

This file contains metadata about the exercise, which provide information on things like difficulty, the author of the exercise (as well as their contact information), and what sort of knowledge is required for the exercise.

I store the data provided by the `let%meta` declarations into a record, which is seralized via the `ppx_deriving` and `yojson` libraries, and written to the `meta.json` file.

3.2 prelude.ml and prepare.ml

These two files are basically the same, with just one key difference. All the `let%prelude` and `let%prepare` declarations are loaded into the exercise environment, giving the student access to whatever data types or functions declared in Learn OCaml's toplevel. The key difference is that the prelude declarations are exposed to the student, while prepare declarations are hidden, which gives the writer of the exercise some amount of control over what a student can use, for example, by shadowing the definition of a library function to prevent the student from using it (like to stop a student from defining a list reverse function using OCaml's own `List.rev` function).

4 Exercises

I refer to the handling of `template.ml` and `solution.ml` under the joined heading "Exercises". I use the same ppx extension `[%erase ...]` for both files. In `template.ml`, any code within the `[%erase ...]` extension will be replaced by a `Not_implemented` exception, while the extension is just stripped away in the `solution.ml` file. Using the `[%erase ...]` extension also allows us to erase individual branches, rather than the whole function body, in case one would like to, for example, provide the base case and have the student fill in a step case. Note that *only* functions which have been erased will show up in the solution file (I assume that these are the only functions which need to be written for the exercise). The generation of these files is handled in `exercise.ml`.

5 Testing

The testing phase is the most interesting portion.

5.1 Samplers

Learn OCaml uses "samplers" to generate random values for test cases, with ones for basic types, as well as lists and option types predefined as part of their test library. A sampler function of some type `'a` takes `unit` and returns a value of type `'a`. However, for types defined by the writer of the assignment, they must manually write samplers to generate random values for testing.

I alleviate this burden from the writer by using the *type declarations* to generate random sampling functions, inspired by the Haskell's QuickCheck [2], plus its Arbitrary and Gen type classes. The explanation of the process follows below, along with a example.

From the corresponding AST node, we have access to the type (its name), as well as its constructors and the types each of the constructors' arguments. This gives us a lot of power.

First let's take a look at a very simple type, `suit` representing the suits of trading cards.

```
type suit = Spades | Hearts | Clubs | Diamonds
```

For each constructor, I create a function which constructs it, and then the sampler will choose one of these functions to invoke, using Learn OCaml's `sample_alternatively`, giving us some random value of this type. The sampler function looks a little bit like this:

```
let rec sample_suit () =
  let constr_Spades () = Spades in
  let constr_Hearts () = Hearts in
  let constr_Clubs () = Clubs in
  let constr_Diamonds () = Diamonds in
  (sample_alternatively
      [constr_Spades; constr_Hearts; constr_Clubs; constr_Diamonds]) ()
```

Then, let's take the basic (polymorphic) tree type defined below.

```
type 'a tree = Tree of 'a tree * 'a tree | Leaf of 'a
```

Since tree is parameterized by the type variable `'a`, we take a cue from Learn OCaml's existing samplers here, which is to pass a sampler that may generate values of some type which will instantiate `'a` (for example, Learn OCaml's `sample_list` takes a sampler as an argument for the values that will populate the list, as the sampler passed to `sample_tree` will populate the tree).

So our sampling function here looks a little something like this:

```
let rec sample_tree sample_a ?(size=10) () =
  let constr_Tree () =
  Tree (sample_tree sample_a ~size:(size - 1) (),
    sample_tree sample_a ~size:(size - 1) ()) in
  let constr_Leaf () = Leaf (sample_a ()) in
  if size = 0
  then (sample_alternatively [cnstr_Leaf]) ()
  else (sample_alternatively [constr_Tree; constr_Leaf]) ()
```

Since the type `tree` is also recursive, we pass the additional optional parameter size, which will bound the size of the tree. The recursive check is just a simple check to see if the name of the type occurs in any of its constructors. For the constructors for which this is not true, we collect them as "terminal constructors" which can be called upon once the size parameter reaches zero.

My tool also passes the annotated solution file (stripped of its annotations), to OCaml's type checker, exposing to us additional type information, which allows us to choose the appropriate sampler to produce the values necessary for a function's inputs, but also allows us generate anonymous samplers for values of types that were not declared above, like products of types.

For example, take the uncurried addition function `plus` as follows:

```
let rec plus (x, y) = x + y
```

From the type checked abstract syntax tree, we can see the function takes as input a tuple of type `int * int`, but since it was not previously defined in some type declaration statement, there is no sampler for it, thus, we fill in the sampler with an anonymous function as below:

```
fun () -> (sample_int (), sample_int ())
```

The creation of samplers is done in `sampler.ml`.

5.2 Generating Tests

The most important part of each set of Learn OCaml exercise files is the `test.ml`. The full features of the testing suite are beyond this tool (hopefully, only for now), but they include things like an AST checker, which somewhat in the vein of aspect oriented programming allows you to tag certain AST nodes and perform certain pre-defined actions, for example, you can forbid a student from using the keyword `Open` to invoke an external module.

This tool is currently focused on just grading the student's solution against the solution provided by the exercise writer. Thanks to the sampler generation above, generating

the `test.ml` file is simply a composition of all the bits and pieces we've collected thus far.

The basic test case has the form below:

```
let [name] =
Section
([Text "Function"; Code "[function␣name]"],
        (test_function_[argcount]_against_solution
        [%ty: [type string]]
        "[function␣name]"
        ~gen:[# of tests]
        ~sampler:[sampler function])
)
```

By convention, I choose to name each test case as the name of the function being tested prefixed by `exercise_`.

The tool will also generate these test cases for the exercise writer. In order to do this, we must invoke OCaml's type checker (using the function `Typemod.type_structure`) on the student's code, and the rest of the generation proceeds using OCaml's `Typedtree` AST.

First, we traverse the typed AST to collect information about each function (but only those which we earlier tagged from the Exercise section). For each function, we record the function's name, its signature, the type of each argument, and the number of arguments.

Finally, in order to fill out the sampler argument for each test case, we proceed similarly to the generation of tuple samplers. In order to invoke the appropriate sampler for each function, we use the types of the arguments to retrieve the correct sampler. The value we send to ∼`sampler` is just an anonymous function that returns a tuple of invocations of the correct sampler for each argument. Note that for types parameterized by type variables, we also randomly choose one type (of `int`, `bool`, and `string`) to instantiate the type variable with, and only this instantiation will be tested.

Then we place all the test cases in the template for the test file, which will end up with references to necessary modules for the Learn OCaml grader, the samplers, the test cases, and finally the function which checks the ast and invokes all the test cases. Handling of test generation is contained within `test.ml`.

6 Conclusion

This tool generates all the files necessary to create an exercise for the Learn OCaml platform. While it's lacking in some features, I believe it goes further than the prior

learn-ocaml-autogen project. As it stands, the tool generates quite a bit of the `test.ml` file such that it would not be too difficult or time consuming to go in and make small changes if necessary, with the automatic creation of samplers already alleviating much of the exercise writer's work in creating the test file. I believe that this is a strong proof of concept that something like a so-called "one-click" generation tool is quite possible.

6.1 Future Work

The biggest hurdle is defining some way to restrict values. For example, generating negative numbers when the function we are testing makes use of the square root function, or generating zeros when that argument would be used as divisor. It would be nice if we could disallow certain arguments, or even favor some others, for example, for some function which tests some property, we may only ever generate values for which the property fails, which isn't in itself all bad, but we might like to generate cases for at least one success and one failure.

A possible solution to restricting values may lie within OCaml's attributes. Perhaps we may use an attribute which gives us extra information about certain arguments.

```
let foo x y z =
  ...
  let w = sqrt(x) in
  ...
[@@restrict (x, positive)]
```

Another issue with random sampling is coverage of test cases. For example, if we have a function that acts on lists, we would definitely want to make sure the function acts as expected on any list, including the empty list, but it may not be very likely that one would generate the empty list, especially within the standard ten test cases that the Learn OCaml test library uses. For this, I propose that we move from random sampling functions to something more like a pseudorandom stream. Each time the function is called, it would produce the next element in the stream, but we would populate the stream with at least one of each of a data type's constructors so that each one is tested on at least once. This is somewhat similar to the process by which pieces are determined in the game Tetris: rather than randomly generating a new piece each round, all the pieces are placed in what is referred to as a "bag", and shuffled, so while the order of the pieces in the bag will be random, once the bag has been exhausted, the every possible piece would have been represented once.

The last two are not entirely necessary, but would be nice features to have. First, the ability to embed PDFs in the exercise environment, which appears in the exercise enviroment and usually has instructions or information for the student. Since each assignment usually has a PDF created for it, it would be nice to use that in place of `descr.html`. Second, learn-ocaml-autogen requires type annotations in function arguments to create

the type string for the test cases, perhaps it would be good to include that functionality to avoid doing more work than we have to.

References

[1] Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. Scaling up functional programming education: under the hood of the ocaml mooc. *Proceedings of the ACM on Programming Languages*, 1(ICFP):4, 2017.

[2] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.