ROTOR: First Steps Towards a Refactoring Tool for OCaml

Reuben N. S. Rowe and Simon J. Thompson

Dedicated tools for automatically performing program refactorings bring major improvements in productivity and reliability during refactoring [1, 6]. Although some tools provide limited support for localised refactoring tasks (e.g. `merlin`'s identifier renaming), there is currently no general purpose automatic refactoring tool for the OCaml language. In this talk, we report on our efforts in developing ROTOR[1], a prototype of such a tool [5]. The OCaml setting brings its own collection of unique challenges for refactoring, and we discuss some of these with reference to a concrete refactoring—the renaming of a value binding in a given module—and in the context of a real-world codebase, namely Jane Street's `Core` library [4].

In the talk, we will discuss the following aspects of our approach.

Leveraging of the OCaml compiler. We use the existing infrastructure provided by the OCaml compiler in order to create and manipulate the abstract representation of the code to be refactored. This has the advantage that we do not (incorrectly) duplicate existing code or functionality, and we obtain 'for free' useful metadata including location, binding, and typing information.

Use of Visitor classes. Visitors are computations that traverse complex data structures to perform specific operations or produce particular values. The core functionality that we overlay onto the compiler is a library of generic visitor classes for the datatypes representing the untyped and typed abstract syntax trees (ASTs), which we generate automatically using the recently released `visitors` PPX preprocessor for OCaml [2, 3]. 'Boilerplate' code that implements the traversal of the syntax tree is generated once, and specific behaviour is obtained by instantiating these classes and overriding the methods for visiting and operating on the particular parts of the AST. In addition to classes for iterating over and mapping between syntax trees, we can also generates classes for *reducing* them to a single value of a desired type. We use such 'reducers' to compute the results of applying refactorings. They are also useful for e.g. *finding* a particular value in a tree, such as the last occurrence of an identifier satisfying some property.

OCaml-specific language features and idioms. ROTOR must be aware of the specifics of the OCaml language. For example, the following must be taken into account when renaming function `foo` belonging to module `A`:

[1] Reliable OCaml-based Tool for OCaml Refactoring.

- OCaml allows the rebinding of identifiers and so when renaming `foo` we must be careful to find and rename the appropriate definition, i.e. the correct binding of the identifier `foo`.

- OCaml allows *punning*, meaning that labels for record fields or named function parameters can be elided when a variable with a matching name is used an argument for such a field or parameter. So, if `foo` is used as a pun for a field or function argument also named `foo`, we must introduce an explicit field or parameter label when renaming.

- It is possible to `include` one (sub)module within another, importing the values it contains, so if `A` is `include`d in a second module `B` we may have to rename not only uses of `A.foo` but also those of `B.foo`.

- OCaml features explicitly named module signatures so, when `A` is `include`d in module `B` and the type of `B` is declared be an independently defined signature `S` exposing the function `foo`, the renaming should fail since it would also require the signature `S` to be modified.

A high-level architecture for refactorings. ROTOR implements an extensible architecture for refactorings. Applying a refactoring to a source file results in a set of textual *replacement* operations, which our prototype currently uses to output file diffs. Each refactoring is implemented as a separate top-level module conforming to a common signature. This allows client code to interact with refactorings in a uniform manner whilst also facilitating a clean and modular implementation style with minimal dependencies. It also enables refactorings to maintain their own internal state. We make use of OCaml's support for *first class* modules to map between abstract representations of refactorings and the modules implementing them.

An important aspect of our model of refactoring is the concept of *dependencies* between refactorings, e.g. above, renaming `A.foo` may depend on also renaming `B.foo`. Such dependencies may be mutual, in that both refactorings must be applied in order for each individually to be correct. Thus, the concept of refactoring dependencies generalises that of preconditions for refactoring. Our architecture allows for refactorings to calculate and return their dependencies over any given codebase. We also consider the notion of refactoring dependencies as having explanatory power, helping the user to better understand the code and aiding in verifying the correctness of refactorings. For example, particular changes resulting from the refactoring can be linked to the particular dependency that generated them, and the graph of such dependencies may provide insight into both the syntactic and semantic structure of the codebase.

Use of a real-world test bed. We have set up a test bed consisting of a real-world codebase, namely (the public version of) Jane Street Capital's `core` library and its dependencies. This comprises almost 900 source files, across nearly 80 individual libraries, and provides a rich and realistic environment for testing our prototype.

References

[1] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[2] François Pottier. The OCaml `visitors` Syntax Extension. `https://gitlab.inria.fr/fpottier/visitors`.

[3] François Pottier. Visitors Unchained. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2017, Oxford, United Kingdom, September 3–9, 2017*. ACM, 2017.

[4] Jane Street Capital. Standard Library Overlay. `https://github.com/janestreet/core`.

[5] Reuben N. S. Rowe. ROTOR, OCaml Refactoring Tool. `https://gitlab.com/trustworthy-refactoring/refactorer`. Prototype.

[6] Simon J. Thompson and Huiqing Li. Refactoring Tools for Functional Languages. *J. Funct. Program.*, 23(3):293–350, 2013.