ROTOR: A Tool for Renaming Values in OCaml's Module System

Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, Scott Owens
School of Computing
University of Kent
Canterbury, UK
{r.n.s.rowe, h.feree, s.j.thompson, s.a.owens}@kent.ac.uk

***Abstract*—The functional programming paradigm presents its own unique challenges to refactoring. For the OCaml language in particular, the expressiveness of its module system makes this a highly non-trivial task and there is currently no automated support for large-scale refactoring in the OCaml language.**

We present ROTOR, a tool for automatically renaming top-level value definitions in OCaml's module system. To compute the effect of renaming, ROTOR relies on a novel concept which we call a *value extension*. This is a collection of related declarations in a program that must all be renamed at once. In practice, this leads to a notion of *dependency*: renaming a function `foo` in module `A` (mutually) depends on renaming function `foo` in module `B` etc.

We describe important aspects of ROTOR's design, implementation, and evaluation on two large codebases: Jane Street's `core` library and its dependencies, and the OCaml compiler itself. In these real-world settings we find that some cases involve a surprisingly complex network of dependencies, and that the use of the PPX preprocessor system presents significant challenges.

I. INTRODUCTION

Refactoring is a necessary and ongoing process in both the development and maintenance of any codebase [1]. Whilst individual refactoring steps are often conceptually very simple, applying them in practice can be complex and error-prone. Since refactorings are context sensitive, text-processing utilities are only effective up to a point: tools for refactoring must be *language-aware*.

In this paper, we report on our efforts to build the first large-scale automatic refactoring tool for the OCaml language [2]. As a first step, we focus on renaming; this is a quintessential refactoring that often displays much of the complexity inherent in refactoring more generally. In particular, we implement automatic renaming of module-level value bindings.

OCaml has a rich and expressive module system, which makes renaming a particularly complex task. Types for modules may be independently declared, bound to (module type) identifiers, modified by various kinds of constraints, and used as annotations that trigger compiler checks. The use of functors (i.e. functions at the module level) introduces further coupling between the modules and module types declared in a program. Declarations in both modules and module types may be shadowed, and duplicated through the use of `include` statements.

Some of the issues are illustrated by the following example.

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
module Pair(X : Stringable)(Y : Stringable) =
struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end
module Int = struct
  type t = int
  let to_string i = int_to_string i
end
module String = struct
  type t = string
  let to_string s = s
end
module P = Pair(Int)(String) ;;
print_endline (P.to_string (5, "Gold Rings")) ;;
```

This program defines a functor `Pair` that takes two modules as arguments, which must conform to the `Stringable` module type. It also defines two structures `Int` and `String`, using them as arguments in an application of `Pair` and binding the result to the module `P`. Suppose we wanted to rename the `to_string` function in the `Int` module. To do so correctly, we must take the following into account:

- Since `Int` is used as the first argument to an application of `Pair`, the `to_string` member of `Pair`'s first parameter must be renamed.
- The first parameter of `Pair` is declared to be of module type `Stringable`, so `to_string` in `Stringable` must be renamed.
- The second parameter of `Pair` is also declared to be of module type `Stringable`, so its `to_string` member must be renamed.
- The `String` module is used as the second argument to an application of `Pair`, and so its `to_string` member must also be renamed.

There are a couple of salient points in this example worth highlighting. Firstly, computing the desired result requires knowledge of the (static) semantics of the language. Although we could correctly rename the `to_string` function in the `String` module by carrying out a global find-and-replace, this would result in renaming too many things. In particular, we do not need to rename the `to_string` function declared in the body of the `Pair` functor.

Secondly, renaming value bindings in OCaml is not simply a matter of α-renaming, à la Lambda Calculus [3]. The declaration of `to_string` in the `Stringable` module type cannot really be said to *bind* the declarations of `to_string` in the modules `Int` and `String`, as the abstraction $\lambda x.M$ binds occurrences of the variable x in its body M. As a functional programming language, OCaml does of course incorporate the notion of binding found in Lambda Calculus. However here we also observe a separate phenomenon. The declarations of `to_string` in the modules `Int` and `String` and the module type `Stringable` are all *related*, but in a different sense to that of (lambda) binding. We call a set of such related declarations the *extension* of a value. By this we mean to express the intuition that each of the declarations is a syntactic facet of the same underlying concept being modelled within the program.

Renaming the binding `to_string` in the `Int` module thus actually *depends* on renaming other bindings in the program: failing to rename any one of them would result in the program being rejected by the compiler. Moreover, this is not simply an artifact of choosing to rename this particular binding; if we were to start with, say, `to_string` in `String` or `Stringable` we would still have to rename the same set of bindings. These bindings are all *mutually* dependent on each other. Consequently, the phenomenon we observe here is distinct from the notion of a refactoring precondition [4]. We have formalised the above notions of value extension and renaming dependency for a large subset of OCaml using an abstract denotational model [5]. This has allowed us to characterise when two declarations should be considered related, and thus need to be renamed together.

An interesting aspect revealed by our analysis is that renaming must be careful to maintain any shadowing present in a program. In some languages, such as Haskell, the scope of a (top-level) binding is the whole module in which it is declared; thus redeclaration is simply erroneous. In the ML family of languages, on the other hand, shadowing is possible due to its lexical scoping: a binding only comes into scope at the point it is declared. Consider the following example.

```
1 module M : sig
2   val foo : bool
3   val foo : string
4 end = struct
5   let foo = 42
6   let foo = "Hello world"
7 end
```

Here there are shadowed declarations in both the module and the module type. Outside the module, references to the binding `M.foo` resolve to the binding of the string literal `"Hello world"`, with a declared type of `string`. If we rename only these declarations, on lines 3 and 6 respectively, the result is the unshadowing of the previous declarations on lines 2 and 5. However, since the unshadowed declaration of type `bool` does not match the actual type of the now corresponding unshadowed binding to the value `42`, the program will be rejected by the compiler. That is, the meaning of the program has been changed.

It is also worth highlighting at this point that there are alternative refactorings one might wish carry out in order to localise the changes involved in a renaming. In the example considered above, we could introduce a new module expression in the application of `Pair` that wraps the reference to the `Int` module and reintroduces a binding with the old name.

```
module P =
  Pair (struct include Int
            let to_string = ⟨new_name⟩ end)
       (String)
```

Currently we do not support such 'enhanced' renaming strategies, but this is an objective for future versions of our tool.

II. Rotor: Design and Implementation

We have built a tool called Rotor, which is capable of automatically renaming value bindings, as illustrated above, in large multi-file OCaml codebases. It is publicly available as an open source software project [6]. Rotor is itself written in OCaml and is easily installed via Opam, OCaml's package manager. Once installed, Rotor is invoked from within the directory containing the source code to be refactored as follows:

```
> rotor -r rename ⟨identifier⟩ ⟨new-name⟩
```

The `-r rename` option indicates that Rotor should rename the value binding denoted by ⟨*identifier*⟩ to ⟨*new-name*⟩. The format of ⟨*identifier*⟩ is discussed below. Rotor outputs a diff patch comprising the changes required to enact the renaming. This allows Rotor to be easily integrated with other tools (e.g. a diff viewer) and for the user to examine the results of carrying out renaming before applying them to the filesystem.

We now discus aspects of Rotor's implementation.

A. Identifying Program Elements

OCaml programs have a hierarchical structure, in which both modules and module types can be nested within one another. OCaml uses 'dot notation' for identifiers, in which the infix operator dot ('`.`') indicates this hierarchical nesting. These identifiers can refer to many different sort of elements in a program (modules, module types, values, etc.). Whilst the context in which an identifier appears indicates which sort of element is being referred to, it is always interpreted to be nested within (sub)modules only. Thus `Foo.Bar.S` could refer to a module type `S` nested in the `Bar` submodule of the module `Foo`. Similarly, `Bigarray.Genarray.create` refers to the `create` function in the `Genarray` submodule of OCaml's standard library module `Bigarray`.

As we have seen above, Rotor needs to be able to refer to elements with a finer degree of structure. For instance, value declarations (e.g. `to_string`) within module types (e.g. `Stringable`). It also needs to refer to parameters of functors, and even distinguish between functors and structures (as well as functor types and signatures).

We therefore had to generalise OCaml's path notation in two ways. Firstly, instead of treating the dot as an infix operator, we use it as a *prefix* operator on names to indicate an element of a particular sort and introduce new prefix operators to express other sorts (e.g. module, module type, value). Secondly, the hierarchical structure is now represented by the sequencing of prefixed names. ROTOR currently uses the operators '`.`', '#', '%', '*', and ':' to indicate structures, functors, structure types (i.e. signatures), functor types, and values, respectively. This scheme is extensible: we can easily add representations for other sorts of OCaml language element (e.g. value types, exceptions, classes, objects, etc.) by adding new prefix operators. We also introduce an indexer element of the form `[`i`]`, to stand for the i^{th} parameter of a functor or functor type. ROTOR's uses the following syntax for identifiers.

$$\begin{aligned} \langle \textit{signifier} \rangle &::= \text{`}\texttt{.}\text{'} \mid \text{`}\texttt{\#}\text{'} \mid \text{`}\texttt{\%}\text{'} \mid \text{`}\texttt{*}\text{'} \mid \text{`}\texttt{:}\text{'} \\ \langle \textit{id\_link} \rangle &::= \langle \textit{signifier} \rangle \ \langle \textit{name} \rangle \mid \text{`}\texttt{[}\text{'} \ \langle \textit{index} \rangle \ \text{`}\texttt{]}\text{'} \\ \langle \textit{identifier} \rangle &::= \langle \textit{id\_link} \rangle \mid \langle \textit{id\_link} \rangle \ \langle \textit{identifier} \rangle \end{aligned}$$

where the nonterminal ⟨*name*⟩ denotes a standard OCaml (short) identifier, and ⟨*number*⟩ denotes a positive integer literal. So, for example, `.Set%S:add` refers to the `add` value declaration within the **`S`** module type within the **`Set`** module. Similarly, `.Set#Make[1]:compare` refers to the declaration of the `compare` value in the first parameter of the **`Make`** functor within the **`Set`** module.

B. Reuse of the OCaml Compiler

ROTOR must carry out many common language processing tasks. For example, it must parse source code into abstract syntax trees (ASTs). It also relies on binding analysis, to determine whether an occurrence of an identifier resolves to the binding to be renamed or not. Rather than reimplement existing functionality ROTOR uses OCaml's `compiler-libs` package, which provides an interface to the compiler. This allows it to obtain and manipulate abstract syntax trees directly, delegating all parsing and type checking to the compiler itself. Moreover, ROTOR can avoid having to be aware of complex build environments (including passes of preprocessors, e.g. PPX) by reading the ASTs directly from the `.cmt` files produced by the compiler and stored on the filesystem. The OCaml compiler also stores detailed information in the AST regarding the source file locations of each syntactic element. Thus it is very straightforward for ROTOR to generate the diff patch describing the renaming.

C. Use of Visitor Classes

The core of ROTOR's operation involves performing traversals of various types over the program's abstract syntax trees. For this, we have made use of the recently developed `visitors` syntax extension for OCaml [7]. This automatically generates classes whose methods perform a bottom-up traversal of values of a given set of datatypes. By default, these visitors do not perform any operation in particular, beyond the basic traversal. However by overriding particular methods complex computations can be carried out on these data values. This extension thus provides a basis in OCaml for similar capabilities to those found in Haskell's SYB generic programming library [8].

D. Computing Renaming Dependencies

For a given value binding, ROTOR computes its set of renaming dependencies (i.e. the value extension to which it belongs) using a worklist algorithm, starting with a working set consisting of the primary binding to be renamed. To process each dependency in the worklist, ROTOR analyses the AST of each source file that depends on the module containing the binding. ROTOR then generates dependencies based on identifying certain syntactic patterns, with each newly generated dependency that has not already been processed being added to the worklist. ROTOR outputs the full list of dependencies generated, along with provenance information, to a log file.

1) Module and Signature Includes: In renaming a binding `.A:foo`, if ROTOR detects that module **`A`** is included in another module **`B`**, e.g.

```
module B = struct include A end
```

then a dependency `.B:foo` is generated. Analogously,

```
module type T = sig include S end
```

would generate the dependency `%T:foo` for the binding `%S:foo`. The reverse is also true.

2) Module and Module Type Aliases: Dependencies are generated similarly when module or module types are aliased.

```
module B = A
module type S = T
```

Here, the dependencies `.B:foo` and `%S:foo` would be generated for bindings `.A:foo` and `%T:foo`, respectively, and vice versa.

3) Module Type Annotations: In renaming `%S:foo`, dependencies are generated by module type annotations, e.g.

```
module A : S = ...
```

Here the dependency `.A:foo` is generated. In the opposite direction, the dependency `%S:foo` is only generated for renaming `.A:foo` when the module type **`S`** actually contains a declaration of `foo` (N.B. it need not: module types can be used to hide sub-components of modules). Module type annotations on functor parameters also generate dependencies, e.g.

```
module F (X : S) = ...
```

generates `#F[1]:foo` for `%S:foo`, and vice versa.

4) Functor Applications: In renaming in functor parameters, e.g. `#F[1]:foo`, the application

```
module M = F (N)
```

generates the dependency `.N:foo`. In the reverse direction, `.N:foo` would only generate `#F[1]:foo` if the declared type of **`F`**'s first parameter contains a declaration for `foo`.

5) Module Type Constraints: In renaming `.M:foo`, if ROTOR encounters a module type constraint

```
S with module N = M
```

a dependency `%S.N:foo` is generated, but only if **`N`** contains a declaration of `foo` (the type of module **`N`** is only required to be a supertype of that of module **`M`**). A dependency is also generated in the reverse direction.

E. Modes of Failure

In certain cases, ROTOR may not be able to compute a renaming and so will exit with an error. These include when it detects that replacing an identifier at some point in the program would change the shadowing structure. That is, the new name would shadow a binding that already exists at that point, or else replacing the identifier would cause a previous binding to be unshadowed. It is also possible ROTOR will detect a renaming dependency requiring changes outside of the codebase that it knows about (for example, in an external library).

III. Experimental Evaluation

We evaluated ROTOR on two substantial, real-world codebases. Firstly, Jane Street's standard library overlay [9], comprising 869 source files in 77 libraries. Secondly, part of the OCaml (4.04.0) compiler itself [10] consisting of 502 source files. We analysed each codebase to extract sets of around 3000 and 2600 value bindings, respectively, which we used as test cases. We selected a fresh name not occurring in either codebase; for each test case, we ran ROTOR to rename the binding to this new name and tested the result by applying the resulting patch and attempting to re-compile.

A. Jane Street Codebase

For the Jane Street testbed, we found that ROTOR's success rate was limited. About 3% of the test cases fail because they have dependencies outside the codebase, namely they require renaming functions in OCaml's standard library. This is perhaps unsurprising since this codebase is designed as an overlay of the standard library. A further 6% fail because they require changes in source files that are automatically generated by the build system (i.e. they are not part of the source code).

A large portion of cases (40%) fail because of the heavy use of the PPX preprocessor system made by this codebase. Most of these involve names which are automatically generated via meta-programmatic means. Thus, renaming them would involve complex reasoning (e.g. renaming parts of string literals), which is beyond the current state of our research. Other of these cases fail because the preprocessor code reuses source code location information but does not always properly set a flag indicating the code is automatically generated. Thus ROTOR generates changes to the source code which should not be applied, resulting in syntax or typing errors.

A small number (5%) fail due to language extensions to OCaml that ROTOR does not yet handle. These include local module bindings (inside expressions), first-class modules, and module type extraction. 9% of cases fail due to implementation bugs in ROTOR. However recompilation succeeds for 37% of test cases. For these, we observe the following:

	Max	Mean	Mode
Files	50	5.0	3
Hunks	128	7.5	3
Dependencies	1127	24.0	19

These statistics reveal the potential complexity of carrying out renaming in real-world code. The maximum number of renaming dependencies observed is over a thousand, and involve a footprint of 128 changes in 50 individual files.

B. OCaml Compiler Codebase

For the compiler testbed, the success rate is much more promising at almost 70%. This is primarily due to the fact that this codebase does not use PPX preprocessing at all. The statistics that we observe from the compiler are as follows.

	Max	Mean	Mode
Files	19	3.8	3
Hunks	59	5.9	3
Dependencies	35	1.6	1

The test cases in the OCaml compiler are clearly simpler than in the Jane Street codebase, involving fewer renaming dependencies and smaller footprints. Nonetheless, about thirty of these cases generate sets of 5 or more dependencies, and over 100 have non-trivial sets of dependencies.

IV. Conclusions and Future Work

We have presented ROTOR, the first fully automatic refactoring tool for multi-file OCaml code. Currently, ROTOR performs renaming of module-level value bindings. The sophistication of OCaml requires a general strategy for identifying program components, and a notion of dependency between renamings. Our evaluation of ROTOR on two large, real-world codebases shows that this is a difficult task in practice. Language preprocessors, e.g. PPX, present significant challenges to automation. In future, we would like to extend ROTOR to automatically rename other identifiers, such as those for modules, module types, value types, and constructors. We would also like to implement more complex refactoring tasks, and integrate ROTOR further with other tools.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. (2018) The OCaml System Release 4.07 Documentation and User's Manual. [Online]. Available: http://caml.inria.fr/pub/docs/manual-ocaml/

[3] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, 2nd ed., ser. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

[4] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[5] R. N. S. Rowe, H. Férée, S. J. Thompson, and S. Owens, "Characterising Renaming within OCaml's Module System: Theory and Implementation," submitted (2019).

[6] (2018) ROTOR: A Prototype Refactoring Tool for OCaml. [Online]. Available: https://gitlab.com/trustworthy-refactoring/refactorer/

[7] F. Pottier, "Visitors Unchained," *PACMPL*, vol. 1, no. ICFP, pp. 28:1–28:28, 2017.

[8] R. Lämmel and S. Peyton Jones, "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming," in *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*. New York, NY, USA: ACM, 2003, pp. 26–37.

[9] Jane Street. (2018) Standard library overlay. [Online]. Available: https://github.com/janestreet/core

[10] (2016) The Core OCaml System: Compilers, Runtime System, Base Libraries (version 4.04.0). [Online]. Available: https://github.com/ocaml/ocaml/tree/4.04.0