API migration: `compare` transformed

Joseph Harrison
Simon Thompson
Steven Varoumas
{J.R.Harrison,S.J.Thompson,S.Varoumas}@kent.ac.uk
University of Kent, Canterbury, UK

Reuben Rowe
reuben.rowe@rhul.ac.uk
Royal Holloway
University of London, UK

Abstract

In this talk we describe our experience in using an automatic API-migration strategy dedicated at changing the signatures of OCaml functions, using the Rotor refactoring tool for OCaml. We perform a case study on open source Jane Street libraries by using Rotor to refactor comparison functions so that they return a more precise variant type rather than an integer. We discuss the difficulties of refactoring the Jane Street code base, which makes extensive use of ppx macros, and ongoing work implementing new refactorings.

1 Motivation

A common part of the software engineering life cycle is refactoring source code: changes need to be made as applications evolve and as technical debt is incurred. Refactoring can be performed manually, but automatic refactoring saves time and reduces room for human error.

Our aim here is to apply meaning-preserving API changes to a large codebase, going beyond changing the name of OCaml values, which was the initial feature of our refactoring tool Rotor [5, 6]. This talk is an experience report on providing an automated and systematic way of changing the types of functions, applied in a case study of the Jane Street libraries. Because of the size of these projects, such changes could not be applied manually to the entire codebase, and this served as an opportunity to add new functionality to Rotor so that this refactoring, as well as other API migrations, can be handled automatically.

2 Approach

Rotor has a high-level model for refactorings that uses visitor classes and the OCaml compiler's libraries to automatically rename *identifiers* across OCaml projects of arbitrary size and complexity. We will show that by extending Rotor with support for targeted code inlining and β-reduction of anonymous function applications, we can refactor OCaml *values* in various ways. For example, with using these techniques we can automatically:

- add/remove arguments to/from functions; and
- modify the order of function arguments; and
- change the type of a function's arguments or result.

In many cases we also use automatic code generation to create *adapters*: small anonymous functions injected at call sites which transform existing function calls to match the updated function definitions. Furthermore, we use Rotor's notion of *dependencies* to automatically update all relevant definitions in source and interface files, for example updating the appropriate type declarations in module signatures.

3 Case Study: `compare` in Jane Street code

Jane Street Capital's open source projects `base` and `core` consist of over 90,000 lines of OCaml code across more than 500 source files. The libraries in these projects often expose a `compare` function for the types or data structures they export; similarly to `Stdlib.compare` these functions return an integer value of -1, 0, or 1. For reasons of precision, these functions could return a specific *ordering* type, such as `Ord.t`, that explicitly states the order relation, rather than coding this as an integer. This representation also has the advantage of containing no "junk" values, such as 42 and -37.

```
module Ord = struct
  type t = Less | Equal | Greater

  let to_int = function
  | Less    -> -1
  | Equal   ->  0
  | Greater ->  1
end
```

The programming style used in the `base` and `core` libraries dictates that for each module `M` that exports a type `t`, the comparison function will be named `M.compare` with type `M.t -> M.t -> int`. Our goal is to refactor each exported `compare` function so that it returns an `Ord.t` instead of an `int`. For each such module, the approach is as follows:

1. The programmer provides a new comparison function (e.g. `M.compare_new : t -> t -> Ord.t`), but leaves the old function `M.compare` unchanged.
2. The programmer modifies `M.compare` so that it is a thin wrapper for the new function, e.g.:
   ```
   let compare x y =
     Ord.to_int (compare_new x y)
   ```
3. The programmer runs Rotor, instructing it to replace calls to `M.compare` with calls to `M.compare_new`.
4. Rotor automatically rewrites all call sites, performing inlining and β-reduction.
5. `M.compare` is deleted, and `M.compare_new` is renamed to `M.compare`.

Once this process is repeated for all comparison functions defined in `base` and `core`, all of the comparison functions which previously returned integers will have been replaced with comparison functions which return variants of `Ord.t`.

As part of this process the definition of the old comparison functions were inlined, and then β-reduction applied to the result to produce more readable code. For example, the call `M.compare x y` is first refactored to:

```
(fun a b -> Ord.to_int (M.compare_ord a b)) x y
```

However, this code is somewhat verbose due to the unnecessary anonymous function definition where `M.compare` once was. By repeatedly β-reducing the refactored expression we can obtain more readable code:

```
Ord.to_int (M.compare_ord x y)
```

`ppx_compare` Both `base` and `core` make extensive use of the `ppx_compare`[7] library, which uses macros to generate comparison functions for user-defined types. If we are to refactor this code successfully, we need to take into account the generated code: there are two approaches to doing this.

The first approach is to modify the macro definitions responsible for generating the comparison functions. Where previously the programmer would modify the definition of a comparison function as part of the refactoring process (steps 1 and 2 above), s/he would instead modify the corresponding *macro* definition itself.

For this approach to be successful, though, Rotor requires a reliable method for detecting AST nodes that were not present in the original source file, because Rotor uses the location information stored in AST nodes when generating text replacements. There is a convention that generated AST nodes are marked with *ghost* locations, but this is not strictly enforced, making it unreliable. We therefore propose changes to `ppx_compare` and `ppxlib` which *force* AST nodes generated by macros to be clearly identified.

The second approach is to store the results of macro expansions in the source files where they were invoked by using the inlining features of `ppxlib`. Instead of generating code during *each* compilation the macros are run *once* and their output is stored in the original source files. The refactoring can then proceed as normal, and Rotor no longer needs to give any special consideration to macros. Once the macros have been expanded however, there is no way to *contract* them again: the refactored code will be out-of-sync with the macro definition and if expanded once again portions of the refactoring will be undone.

4 Related work

Previous authors have suggested that the process of generating refactorings on API evolution could be generated automatically: the key insight being that the new API should be able to implement all the services of the old API. Using this adaptor, body [4], wrapper [8] or twinning [3], allows replacement of calls to the old API by calls to the new, and once replaced the code can further be rewritten automatically too [1]. A related approach uses data-flow to determine replacement points, but additionally requires rewrites to be described explicitly [2].

5 Ongoing & Future Work

Ongoing work is focused on generalising the approach described in this talk in order to handle different kinds of refactorings related to changes appearing at the interface level, such as swapping the order of functions arguments or adding a new argument to a function, as mentioned in section 2. Furthermore, current work in progress is dedicated to automatically checking the semantic equivalence of a program before and after application of such refactorings. This makes use of the aforementioned inlining and β-reduction transformations in order to transform both versions of a program to (simpler) normal forms: equality of the normal forms implies equivalence of the two program versions.

Finally, further code transformations might be used as a way of "cleaning-up" the inlined adapter code introduced by such refactorings by getting rid of the added wrapper functions: for example, ideally, the following expression:

```
Int.equal (Ord.to_int (compare x y)) 1
```

could have its wrapper code removed and be turned into:

```
Ord.equal (compare x y) Ord.Greater
```

Although such transformations would often produce cleaner code, they might be difficult to automate because they rely on a high-level semantics of the program, and would still need to be steered by the programmer to preserve the meaning of the transformed expressions.

References

[1] Huiqing Li and Simon Thompson. 2012. Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs. In *ASE'12*, Tim Menzies and Motoshi Saeki (Eds.). IEEE Computer Society.

[2] László Lövei. 2009. Automated Module Interface Upgrade. In *Proceedings of the 2009 ACM SIGPLAN Erlang Workshop*. Edinburgh, Scotland, 11–21.

[3] M. Nita and D. Notkin. 2010. Using twinning to adapt programs to alternative APIs. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*.

[4] Jeff H. Perkins. 2005. Automatically Generating Refactorings to Support API Evolution. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. Association for Computing Machinery, 111–114.

[5] Reuben Rowe and Simon Thompson. 2017. Rotor: First Steps Towards a Refactoring Tool for OCaml. In *The OCaml Users and Developers Workshop 2017*.

[6] Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, and Scott Owens. 2019. Characterising renaming within OCaml's module system: theory and implementation. In *Proceedings (PLDI 2019)*. ACM.

[7] Jane Street. 2020. ppx_compare : Generation of comparison functions from types. https://github.com/janestreet/ppx_compare.

[8] Thiago Tonelli, Krzysztof Czarnecki, and Ralf Lämmel. 2010. Swing to SWT and back: Patterns for API migration by wrapping. In *Proceedings ICSM '10*. IEEE Computer Society, 1–10.