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

Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, Scott Owens
University of Kent, Canterbury

Third International Workshop on Refactoring
Tuesday 28th May 2019, Montréal, Canada

Why OCaml?

- OCaml is a functional programming language
- It is industrially relevant
  - Used by over 50 companies
  - 600 publicly released pacakges/libraries
  - > 11,000 open source projects
- The module system presents interesting challenges
- No existing tool support for refactoring

Renaming: A First Step

- Only substitute identifiers (no new code)
- Preserve behaviour/correctness (incl. compilability)
- Keep the footprint minimal (not simply ‘replace all’)
- This requires a ‘whole program’ analysis

Renaming in OCaml is Hard!

Expressiveness of the module system introduce complications:

- Explicit module type annotations (i.e. interfaces)
- Module and module type `include`
- Module and module type aliasing
- Module type constraints
- Functors

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

reference to
parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

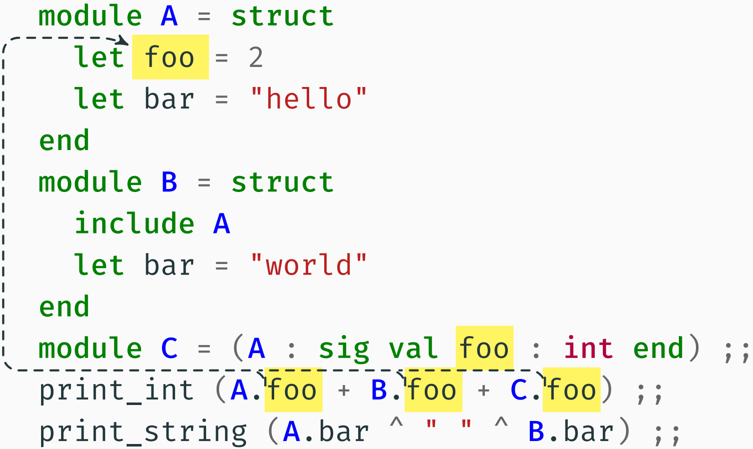
reference to
parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

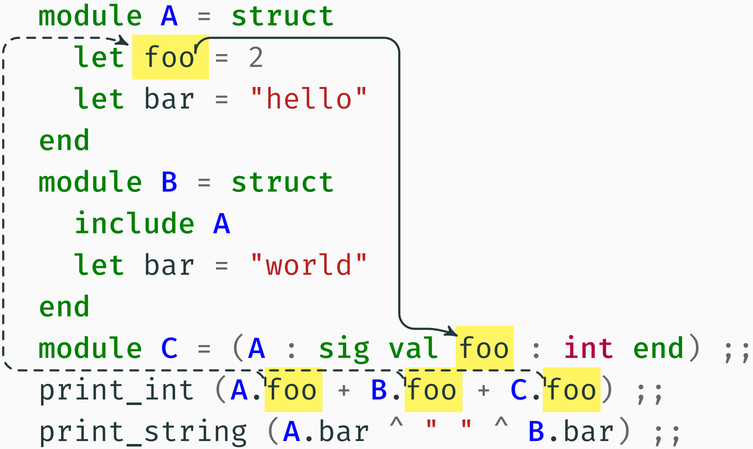
reference to parent module

Example: Module Includes and Aliases



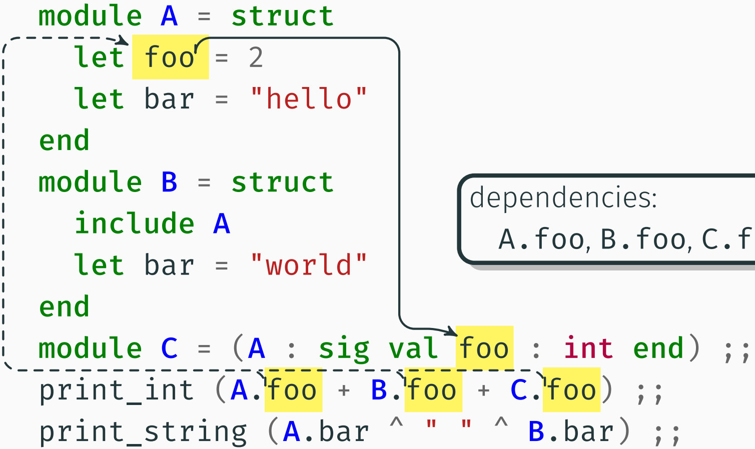
```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

Example: Module Includes and Aliases



```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

Example: Module Includes and Aliases



```
module A = struct
  let foo = 2
  let bar = "hello"
end
module B = struct
  include A
  let bar = "world"
end
module C = (A : sig val foo : int end) ;;
print_int (A.foo + B.foo + C.foo) ;;
print_string (A.bar ^ " " ^ B.bar) ;;
```

dependencies:
A.foo, B.foo, C.foo

Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int 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!")) ;;
```

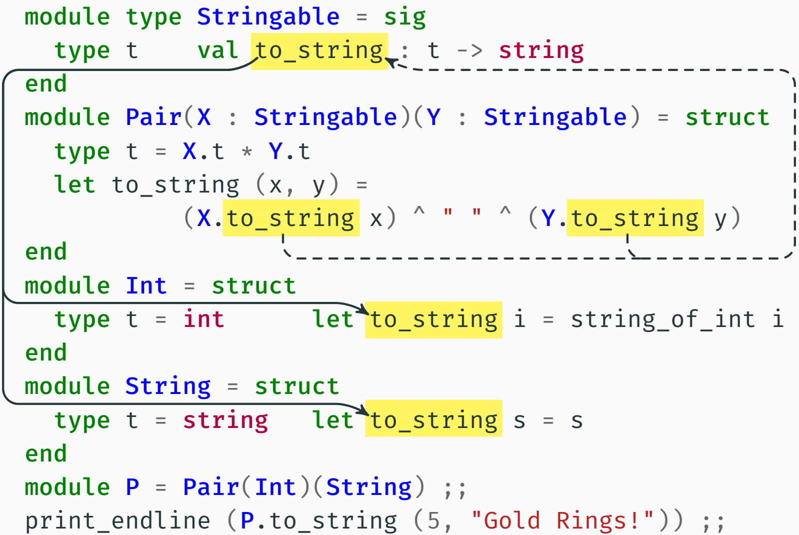
Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int 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!")) ;;
```

Example: Functors



```
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 = string_of_int 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!")) ;;
```

Example: Functors

```
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 = string_of_int i
end
module String = struct
  type t = string   let to_string s = s
```

dependencies:

`Int.to_string`, `String.to_string`,
`Stringable.to_string`, `Pair[1].to_string`, `Pair[2].to_string`

ROTOR: Main Features

- Implemented in OCaml itself
- Visitor classes used to manipulate ASTs
- Performs fine-grained module dependency analysis
- Outputs detailed information on renaming dependencies

Experimental Evaluation

- OCaml compiler (~500 files, ~2650 test cases)
    - Re-compilation successful for 70% of cases
- Jane Street standard library overlay (~900 files, ~3000 test cases)
    - Re-compilation successful for 37% of cases
    - 46% fail due to use of language preprocessor
    - 5% require changes in external libraries

Experimental Evaluation

OCaml Compiler Codebase

	Files	Hunks	Deps	Avg. Hunks/File
Max	19	59	35	15.0
Mean	3.8	5.9	1.6	1.5
Mode	3	3	1	1.0

Jane Street Standard Library Overlay

	Files	Hunks	Deps	Avg. Hunks/File
Max	50	128	1127	5.7
Mean	5.0	7.5	24.0	1.3
Mode	3	3	19	1.0

Conclusions

- Big impact for automatic refactoring in functional programming
- OCaml's module system introduces much complexity
- Require a notion of refactoring dependency
- Much work still to be done!

Future Work

- Handle more language features
  - first-class modules, module type extraction, type-level module aliases
- Other renamings
  - modules, module types, types, record fields, constructors, classes/methods
- More sophisticated renamings strategies
- Other refactorings
  - rename/add/remove function parameter, function generalisation, etc.
- IDE/build system integration

Thank You!

https://gitlab.com/trustworthy-refactoring/refactorer