Refactoring in OCaml: Challenges and Solutions

Reuben N. S. Rowe[1]
Joint work with: Hugo Férée[2], Simon J. Thompson[3], Scott Owens[3,4]

EuroProofNet Workshop on the Development, Maintenance, Refactoring and Search of Large Libraries of Proofs
Saturday 24th September 2022

[1]Royal Holloway, University of London,
[2]Université Paris-Diderot,
[3]University of Kent, Canterbury
[4]Facebook

Why I Am ~~Not~~ Talking About Proof Assistants

Proof Assistants are (enhanced) functional programming languages

- Coq modules based on OCaml's modules
- Agda's module system facilitates aliasing and re-exporting
- Type classes in Isabelle/Lean induce dependencies between definitions

Challenges for Refactoring OCaml

OCaml's module system is very expressive.

- Structures and signatures
- Module/signature **`include`**
- Functors: (higher-order) functions between modules
- Module type constraints and (type level) module aliases
- Module type extraction
- Recursive and first-class modules

Renaming: A Minimal, Non-trivial Setting

Renaming (top-level) value bindings within modules

```
module M : sig
  val foo : 'a -> 'a
  val bar : int
end = struct
  let foo x = x
  let bar = 42
end
```

- Get the 'basics' right first, the rest will follow
- Already requires solving problems relevant to all refactorings

Complexities of the Module System: Functors and Module Types

```
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 Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Complexities of the Module System: Functors and Module Types

```
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 Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Identifiers in declarations renamed along with the references that resolve to them

Complexities of the Module System: Functors and Module Types

```
module Int = struct
  type t = int
  let to_string i = string_of_int i
end
```

```
module Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

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

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Declarations may be connected via module type annotations

Complexities of the Module System: Functors and Module Types

```
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 Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Declarations may be connected via module type annotations

Complexities of the Module System: Functors and Module Types

```
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 Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Declarations may be connected via module type annotations

Complexities of the Module System: Functors and Module Types

```
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 Pair =
  functor (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 P = Pair(Int)(String)

print_endline (P.to_string (5, "Gold Rings!"))
```

```
module type Stringable = sig
  type t
  val to_string : t -> string
end
```

Declarations may be connected via module type annotations

Shadowing

```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.foo ;;
```

Shadowing

```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.foo ;;
```

Shadowing

```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.foo ;;
```

Shadowing

```
module M : sig
    val foo : float
    val foo : string
  end =
  struct
    let foo = 5
    let foo = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.foo ;;
```

Shadowing

```
module M : sig
    val foo : float
    val bar : string
  end =
  struct
    let foo = 5
    let bar = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.bar ;;
```

Shadowing

```
module M : sig
    val foo : float
    val foo : string
  end =
  struct
    let foo = 5
    let foo = (string_of_int foo) ^ " Gold Rings!"
  end ;;
print_endline M.foo ;;
```

Module Includes and Encapsulation

```
module A = struct
  let foo = 42
  let bar = "Hello"
end

module B = struct
  include A
  let bar = "World!"
end
```

Module Includes and Encapsulation

```
module A = struct
  let foo = 42
  let bar = "Hello"
end

module B = struct
  include (A : sig val foo : int end)
  let bar = "World!"
end
```

Some Observations

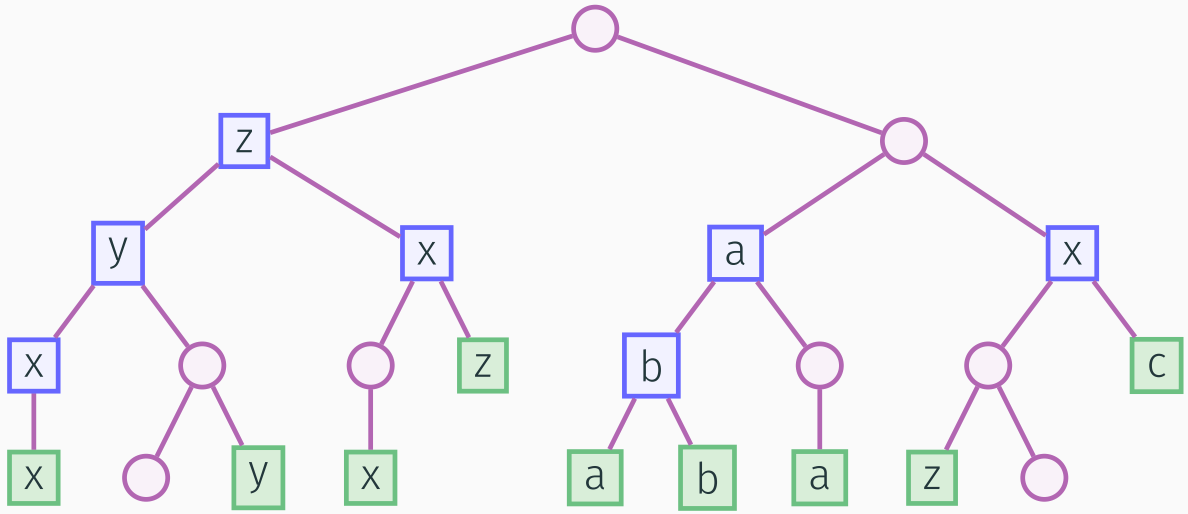
- Basic renamings rely on binding resolution information
- Program structure induces dependencies between renamings
- Disparate parts of a program can together make up a single logical meta-level entity

Our Solution

We devised an abstract, denotational semantics for programs

- Covers a subset of OCaml
- Characterises changes needed to rename value bindings
- Provides a framework for developing a ‘theory of renaming’
- Abstract semantics and renaming theory formalised in Coq

Renaming, Abstractly

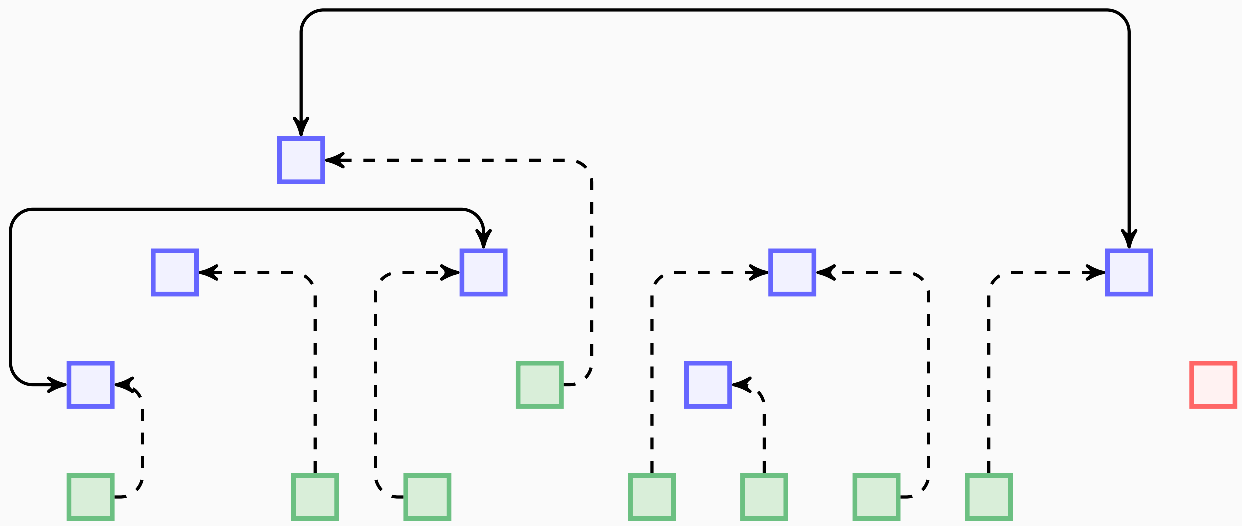
$P =$ 

We distinguish two types of identifiers: declarations (x) and references (b)

A *renaming* of P to P’ changes *only* identifiers

- AST structure is identical otherwise

Renaming, Abstractly

$[\![P]\!]$ $=$ 

Definition (Valid Renamings)

A renaming of P to P' is *valid* when $[\![P]\!] = [\![P']\!]$

A Renaming Theory

1. Valid renamings induce an equivalence relation on programs
2. Renamings are characterised by (mutual) dependencies
3. We can construct a minimal renaming for any binding
4. Valid renamings can be factorised into atomic renamings
5. If $[\![P]\!] = [\![P']\!]$, then P and P' are operationally equivalent
    - Do not have the converse: valid renamings must preserve shadowing

Language Coverage

modules and module types
functors and functor types
module and module type **`open`**
module and module type **`include`**
module and module type aliases
constraints on module types
module type extraction
simple λ-expressions (no value types)

recursive modules
first class modules
type-level module aliases
complex patterns, records
references
the object system

Rotor: A Tool for Automatic Renaming in OCaml

- Implemented in OCaml, integrated into the OCaml ecosystem
- Outputs patch file and information on renaming dependencies
- Fails with a warning when renaming not possible:
    1. Binding structure would change (i.e. name capture)
    2. Requires renaming bindings external to input codebase

Dealing with Practicalities

- ROTOR only *approximates* our formal analysis
  - Only intra-file binding information provided by compiler
  - Inter-file binding information remains as logical paths
- Code can be generated by the OCaml pre-processor (PPX)
  - ROTOR reads the post-processed ASTs directly from files
  - Not all generated code correctly flagged as ‘ghost’ code

Lessons From Implementation

Reuse existing ecosystem as much as possible

- OCaml's `compiler-libs` package interfaces with compiler
  - Don't have to do parsing/type inference ourselves
  - Can rely on build artifacts that store AST representations
- Ocaml's `visitors` library generates code for AST traversals
  - Automates generation of biolerplate code
- We integrate with the `dune` build tool
  - Provides information about the 'workspace' and build environment

Experimental Evaluation

- Jane Street standard library overlay (~900 files)
  - ~3000 externally visible top-level bindings (~1400 generated by PPX)
  - Re-compilation after renaming successful for 68% of cases
  - 10% require changes in external libraries
- OCaml compiler (~500 files)
  - ~2650 externally visible top-level bindings
  - Self-contained, no use of PPX preprocessor
  - Re-compilation after renaming successful for 70% of cases

Next Steps (for EPF WG4)

- Can our high-level observations be lifted to proof languages?
- What new name resolution and dependency phenomena do we find in this context?
- Do proof assistants have sufficient tool/ecosystem infrastructure?

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

With thanks for support from: EPSRC Engineering and Physical Sciences Research Council, Jane Street