Characterising Renaming within OCaml's Module System

Reuben N. S. Rowe[1], Hugo Férée[2], Simon J. Thompson[3], Scott Owens[3]

TU Delft Programming Languages Seminar
Wednesday 4th December 2019

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

Delft University of Technology

Motivation

- Refactorings in the wild can be large, tedious, error-prone
- Most refactoring research targets object-oriented languages
- More recent work targets Haskell and Erlang
- OCaml presents different challenges/opportunities

The First Step

Renaming (top-level) value bindings within modules

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

Our Contributions

1. Abstract semantics for a subset of OCaml
    - Characterises changes needed to rename value bindings
2. Coq formalisation of abstract semantics and renaming theory
3. Prototype tool, ROTOR, for automatic renaming in full OCaml

What is Difficult in 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

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

Complexities of the Module System

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

module Pair = functor (X : Stringable)(Y : Stringable) ->
  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!")) ;;
```

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

Encapsulation

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

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

Encapsulation

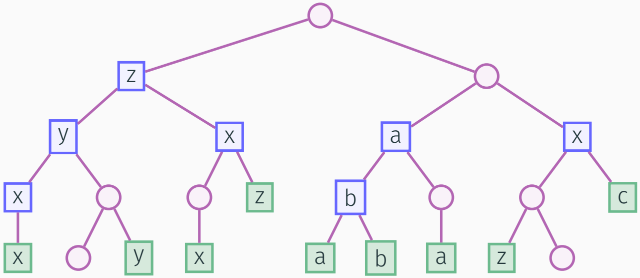
```
module A = struct
  let foo = 42
  let bar = "Hello"
end

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

Renaming, Abstractly

$P =$

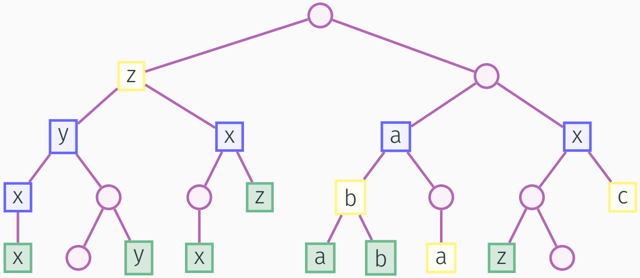


We distingish identifiers that are:

- declarations, e.g. x
- references, e.g. b

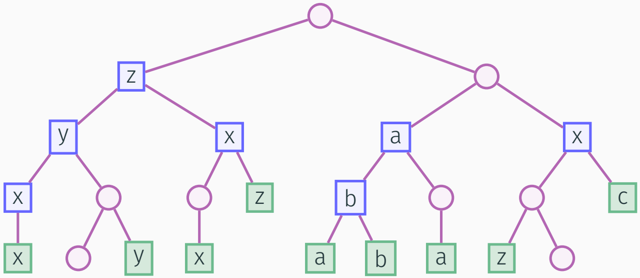
Renaming, Abstractly

$P' =$



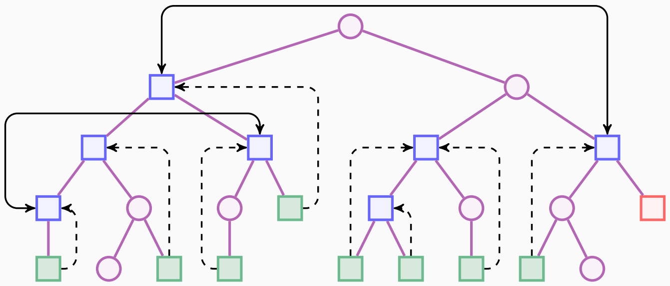
- $P \twoheadrightarrow P'$ means P' is a renaming of P: it changes **only** identifiers
- The set of changed identifiers is called the footprint, e.g. z

Abstract Semantics for Renaming

$P \quad = $ 

Abstract Semantics for Renaming

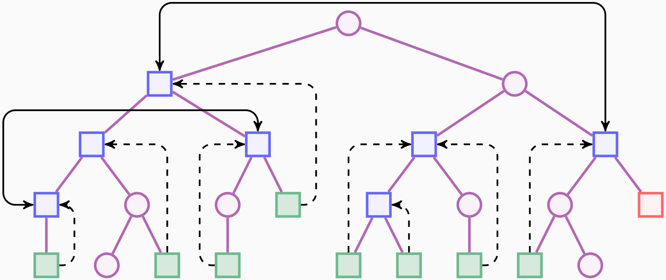
$$[\![P]\!] = (V, \rightarrowtail, \mathbb{E}) \qquad =$$



V nodes, $\rightarrowtail$ name resolution, $\mathbb{E}$ declaration dependency, □ unresolved reference

Abstract Semantics for Renaming

$$[\![P]\!] = (V, \rightarrowtail, \mathbb{E}) \quad =$$



V nodes, $\rightarrowtail$ name resolution, $\mathbb{E}$ declaration dependency,  unresolved reference

Definition (Valid Renamings)

$P \twoheadrightarrow P'$ is a valid renaming when $[\![P]\!] = [\![P']\!]$

A Renaming Theory

1. Valid renamings induce an equivalence relation on programs

Theorem

For all programs P, P', and P'':

- $P \twoheadrightarrow P$ *is valid*
- *if* $P \twoheadrightarrow P'$ *is valid then so is* $P' \twoheadrightarrow P$
- *if* $P \twoheadrightarrow P'$ *and* $P' \twoheadrightarrow P''$ *are valid then so is* $P \twoheadrightarrow P''$

A Renaming Theory

2. Renamings are characterised by (mutual) dependencies.

$$\text{deps}(P, P') = \{v \mid v \in \text{footprint}(P, P') \text{ and } v \text{ a declaration}\}$$

Theorem Suppose $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (V, \rightarrowtail, \mathbb{E})$, and let $L = \{v \mid v \in \text{deps}(P, P') \vee \exists v' \in \text{deps}(P, P').\, v \rightarrowtail v'\}$; then

1. $L \subseteq \text{footprint}(P, P')$
2. $v \rightarrowtail \bot$ for all $v \in \text{footprint}(P, P') \setminus L$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (V, \rightarrowtail, \mathbb{E})$, then $\text{deps}(P, P')$ has a partitioning that is a subset of $V_{/\mathbb{E}}$

A Renaming Theory

3. We can construct a minimal renaming for any binding

Theorem

Let $[\![P]\!] = (V, \rightarrowtail, \mathbb{E})$*,* $v \in V$ *be a declaration in* P *and* i *a fresh value identifier, then* $P \twoheadrightarrow P'$ *is a valid renaming, where* $P' = P[v' \mapsto i \mid v' \in [v]_{\mathbb{E}} \vee \exists v'' \in [v]_{\mathbb{E}}.\, v' \rightarrowtail v'']$

($[v]_{\mathbb{E}}$ denotes the $\mathbb{E}$-equivalence class containing v)

4. Valid renamings can be factorised into atomic renamings

Theorem

Suppose $P \twoheadrightarrow P'$ *is valid with* $[\![P]\!] = (V, \rightarrowtail, \mathbb{E})$*, and let* v *and* v' *be two distinct declarations in* $\mathrm{deps}(P, P')$ *such that* $(v, v') \notin \mathbb{E}$ *and the new name for* v *is fresh; then there exists* P'' *such that both* $P \twoheadrightarrow P''$ *and* $P'' \twoheadrightarrow P'$ *are valid, with* $\mathrm{footprint}(P, P'') \subset \mathrm{footprint}(P, P')$ *and* $\mathrm{footprint}(P'', P') \subset \mathrm{footprint}(P, P')$.

Adequacy of the Semantics

We define a denotational semantics $(\!|\cdot|\!)$ of the operational meaning of programs

- Extends the model defined by Leroy (POPL '95)
- But module types contribute to the meaning of programs

Theorem
If $P \twoheadrightarrow P'$ *is a valid renaming, then* $(\!|P|\!) = (\!|P'|\!)$

Adequacy of the Semantics

We define a denotational semantics $(\!|\cdot|\!)$ of the operational meaning of programs

- Extends the model defined by Leroy (POPL '95)
- But module types contribute to the meaning of programs

Theorem
If $P \twoheadrightarrow P'$ *is a valid renaming, then* $(\!|P|\!) = (\!|P'|\!)$

We do not have a completeness result since 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

- 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 is 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

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

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

Future Work

- Handle more language features
- Other renamings, more sophisticated transformations
- Other kinds of refactorings
- IDE/build system integration

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

https://zenodo.org/record/2646525

With thanks for support from:

EPSRC Engineering and Physical Sciences Research Council

Jane Street