Characterising Renaming within OCaml’s Module System

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

40th ACM SIGPLAN Conference on Programming Language Design and Implementation
Tuesday 25th June 2019, Phoenix, AZ, USA

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

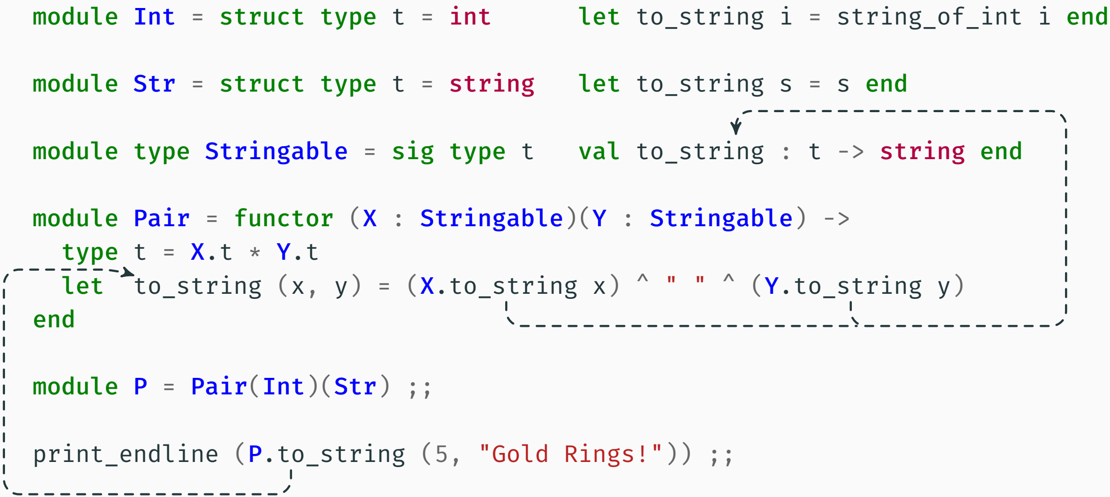
Complexities of the Module System

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

module Str = 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)(Str) ;;

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

Complexities of the Module System

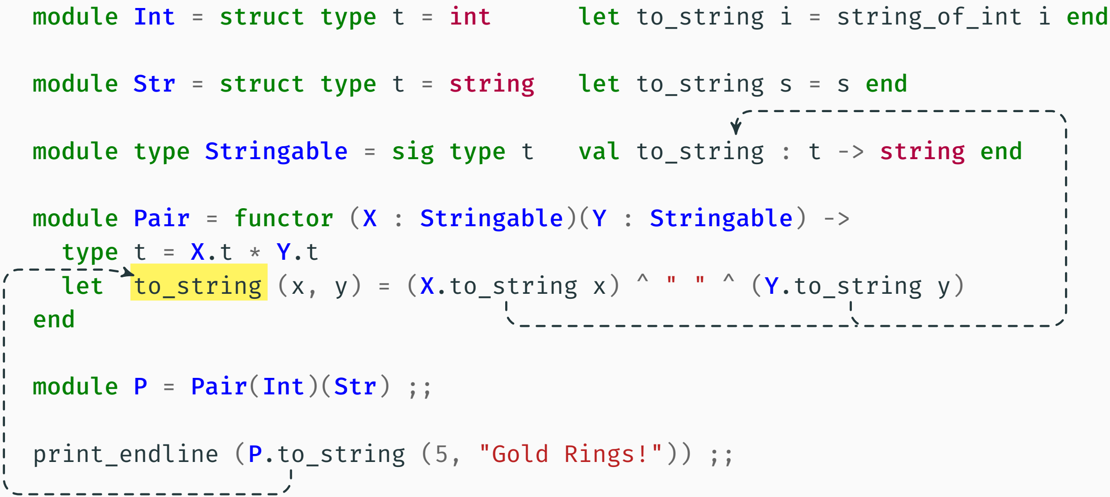
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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

Complexities of the Module System



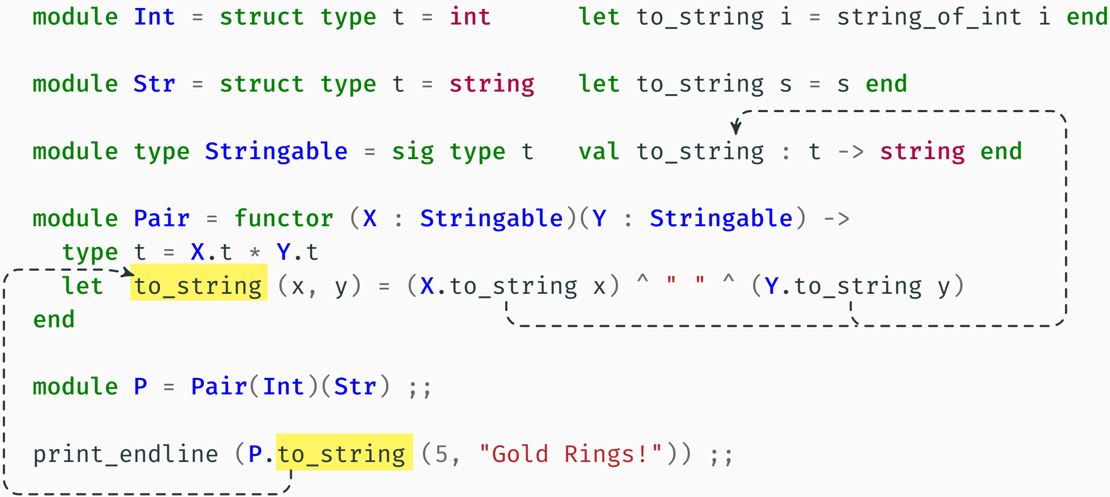
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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

Complexities of the Module System



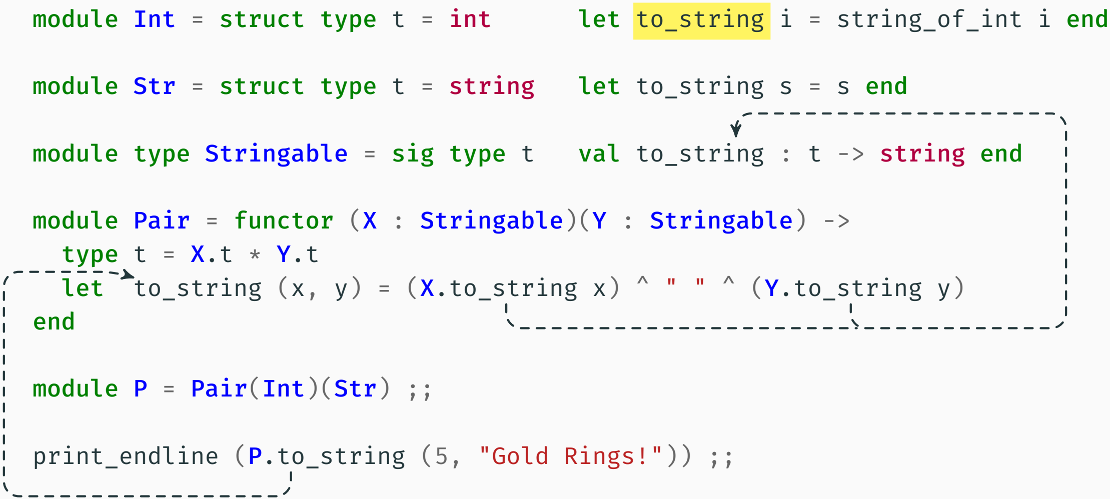
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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

Complexities of the Module System



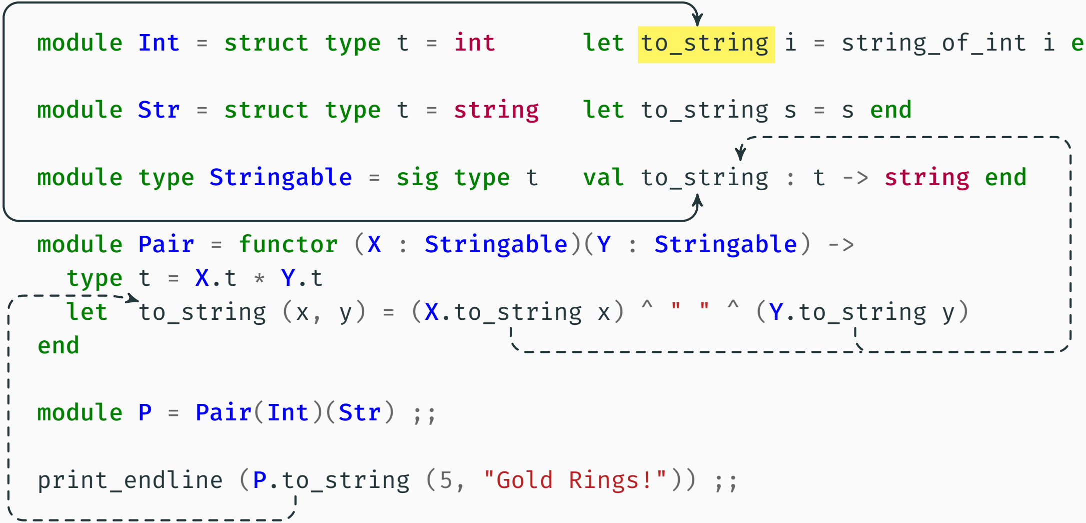
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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

Complexities of the Module System



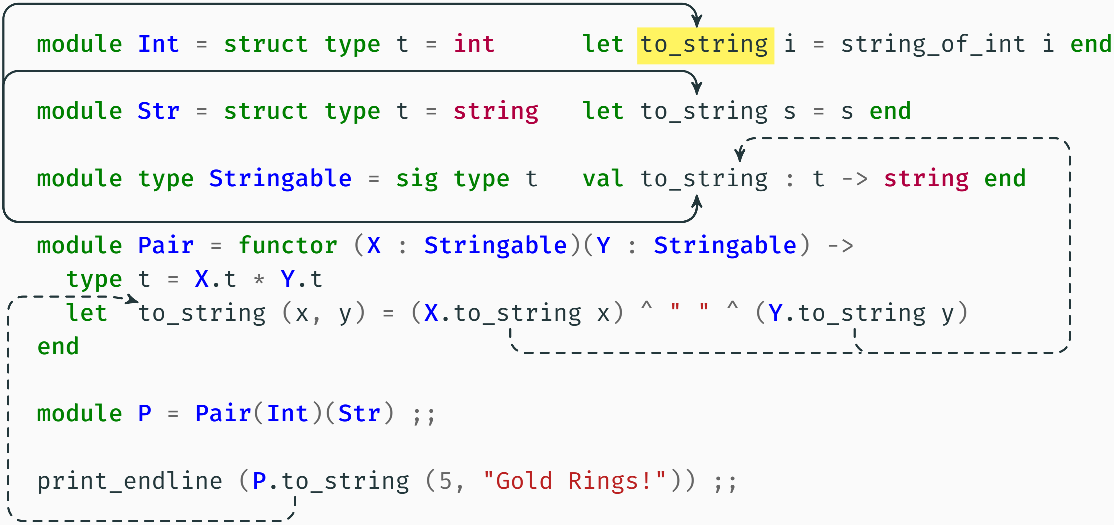
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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

Complexities of the Module System



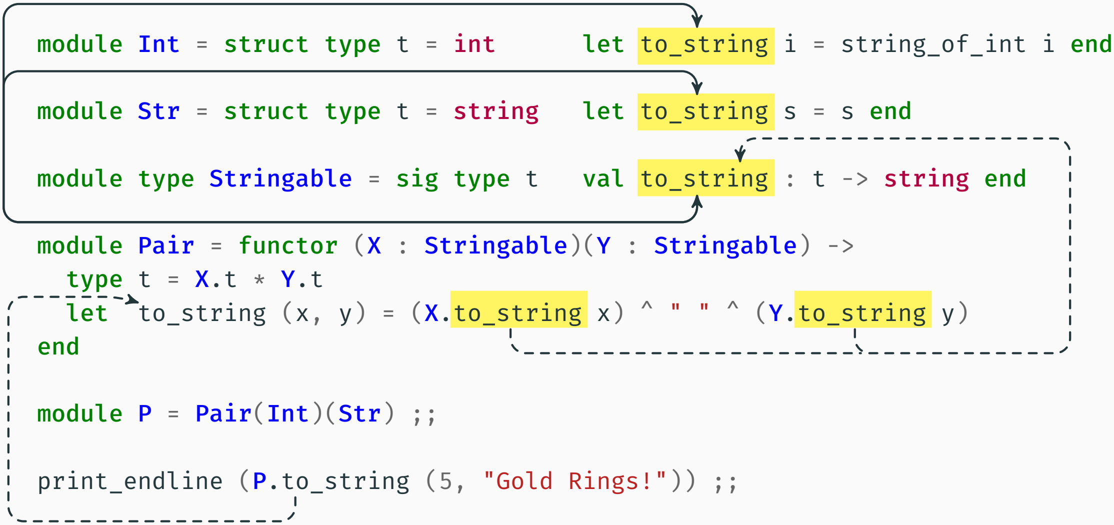
```
module Int = struct type t = int      let to_string i = string_of_int i end

module Str = 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)(Str) ;;

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 Str = 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)(Str) ;;

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

Complexities of the Module System

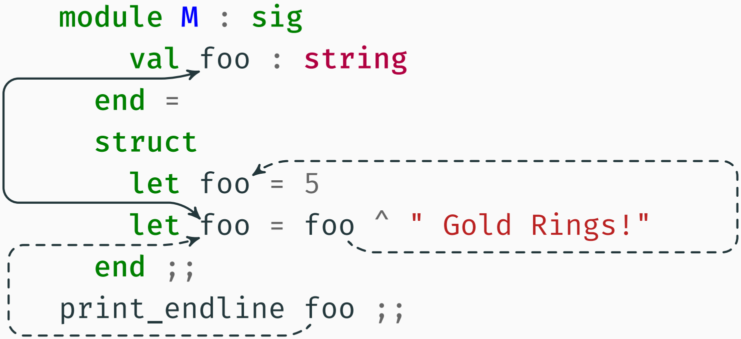


```
module Int = struct type t = int      let to_string i = string_of_int i end
module Str = struct type t = string   let to_string s = s end
module type Stringable = sig type t   val to_string : t -> string end
module Pair = functor (X : Stringable)(Y : Stringable) ->
  type t = X.t * Y.t
  let  to_string (x, y) = (X.to_string x) ^ " " ^ (Y.to_string y)
end

module P = Pair(Int)(Str) ;;

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

Shadowing

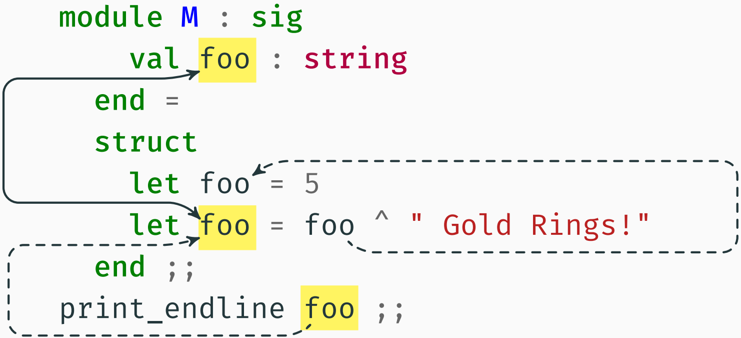
```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = foo ^ " Gold Rings!"
  end ;;
print_endline foo ;;
```

Shadowing



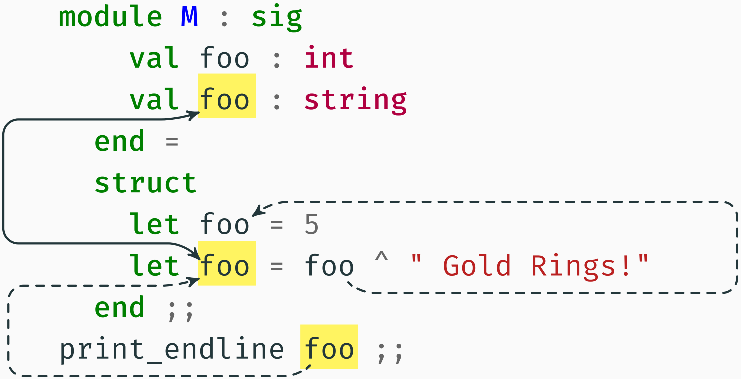
```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = foo ^ " Gold Rings!"
  end ;;
print_endline foo ;;
```

Shadowing



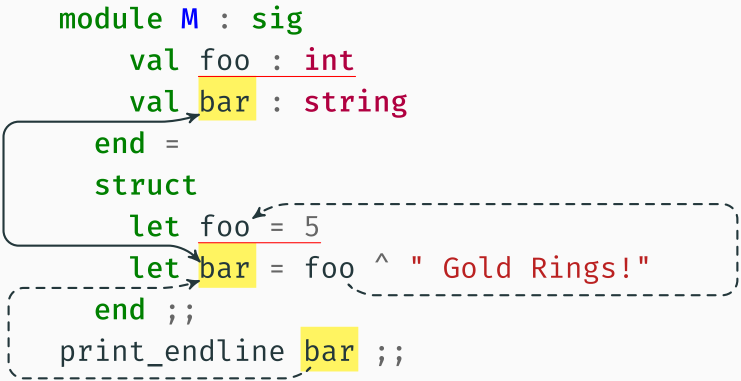
```
module M : sig
    val foo : string
  end =
  struct
    let foo = 5
    let foo = foo ^ " Gold Rings!"
  end ;;
print_endline foo ;;
```

Shadowing



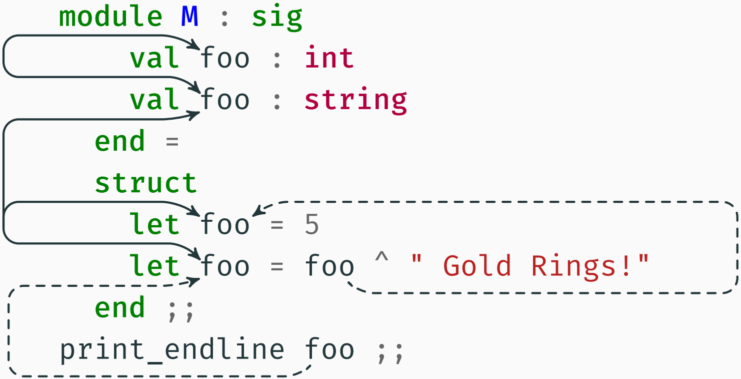
```
module M : sig
    val foo : int
    val foo : string
  end =
  struct
    let foo = 5
    let foo = foo ^ " Gold Rings!"
  end ;;
print_endline foo ;;
```

Shadowing



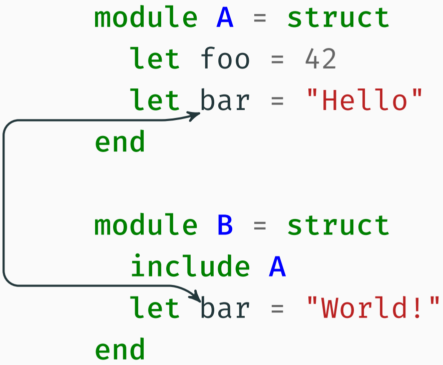
```
module M : sig
    val foo : int
    val bar : string
  end =
  struct
    let foo = 5
    let bar = foo ^ " Gold Rings!"
  end ;;
print_endline bar ;;
```

Shadowing



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

Encapsulation



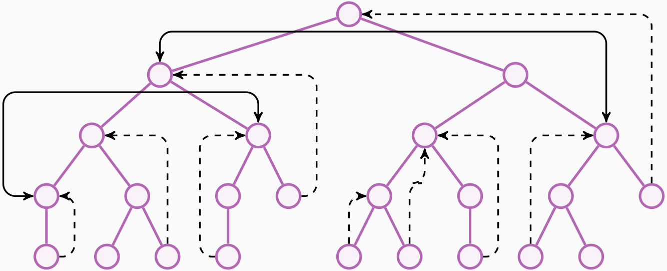
```
module A = struct
  let foo = 42
  let bar = "Hello"
end

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

Encapsulation

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

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

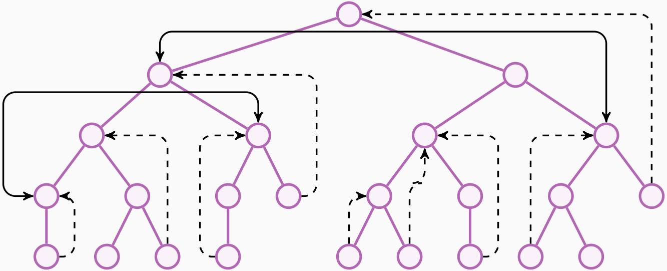
Abstract Semantics for Renaming

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



Abstract Semantics for Renaming

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

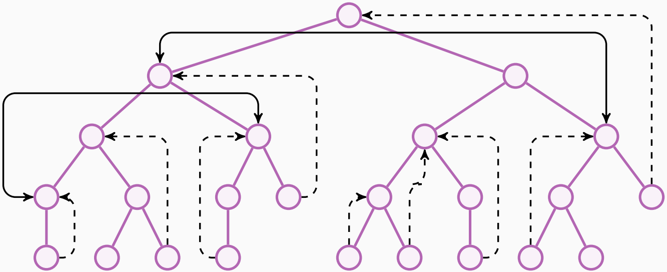


Definition (Valid Renamings)

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

Abstract Semantics for Renaming

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



Definition (Valid Renamings)

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

Theorem (Adequacy)

If $[\![P]\!] = [\![P']\!]$, then P and P' are operationally equivalent

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

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

Experimental Evaluation

- Jane Street standard library overlay (~900 files)
  - ~3000 externally visible top-level bindings
    - of which ~1400 are automatically 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