Towards a Formal Theory of Renaming for OCaml

Reuben Rowe

joint work-in-progress with Simon Thompson and Hugo Férée

University of Birmingham Theory Seminar
Friday 19th October 2018

What's the Story?

- Renaming (e.g. functions) in (real-world) OCaml is hard!

What’s the Story?

- Renaming (e.g. functions) in (real-world) OCaml is hard!
  - usual engineering problems
    - multi-file programs
    - preprocessors
    - build systems …

What's the Story?

- Renaming (e.g. functions) in (real-world) OCaml is hard!
  - usual engineering problems
    - multi-file programs
    - preprocessors
    - build systems …
  - but also due to powerful module system
    - functors
    - module types
    - aliases and constraints …

What's the Story?

- Renaming (e.g. functions) in (real-world) OCaml is hard!
  - usual engineering problems
    - multi-file programs
    - preprocessors
    - build systems ...
  - but also due to powerful module system
    - functors
    - module types
    - aliases and constraints ...
- Need a formal mechanism for reasoning about renaming
  - Abstract denotational semantics

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

reference to parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

reference to parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

reference to parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

reference to parent module

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let foo = foo
  let bar = 3
end
module C = (A : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Includes and Aliases

```
module A = struct
  let foo = 1
  let bar = 2
end
module B = struct
  include A
  let foo = foo
  let bar = 3
end
module C = (struct include A
    let foo = foo
  end : sig val foo : int end)
print_int (A.foo + B.foo + C.foo) ;;
print_int (A.bar + B.bar) ;;
```

Example: Module Interfaces

```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

```
module M1 : Magma =
struct
  type t = int
  let op x y = …
  let equal x y = …
  let choose () = …
end
```

```
module M2 : Magma =
struct
  type t = float
  let op x y = …
  let equal x y = …
  let choose () = …
end ;;
```

```
let x = M1.choose () in M1.equal (M1.op x x) x ;;
let y = M2.choose () in M2.equal (M2.op x x) x ;;
```

Example: Module Interfaces

```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

```
module M1 : Magma =
struct
  type t = int
  let op x y = …
  let equal x y = …
  let choose () = …
end
```

```
module M2 : Magma =
struct
  type t = float
  let op x y = …
  let equal x y = …
  let choose () = …
end ;;
```

```
let x = M1.choose () in M1.equal (M1.op x x) x ;;
let y = M2.choose () in M2.equal (M2.op x x) x ;;
```

Example: Module Interfaces

```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

```
module M1 : Magma =
struct
  type t = int
  let op x y = …
  let equal x y = …
  let choose () = …
end
```

```
module M2 : Magma =
struct
  type t = float
  let op x y = …
  let equal x y = …
  let choose () = …
end ;;
```

```
let x = M1.choose () in M1.equal (M1.op x x) x ;;
let y = M2.choose () in M2.equal (M2.op x x) x ;;
```

Example: Module Interfaces

```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

dependencies induced by using interfaces

```
module M1 : Magma =
struct
  type t = int
  let op x y = …
  let equal x y = …
  let choose () = …
end
```

```
module M2 : Magma =
struct
  type t = float
  let op x y = …
  let equal x y = …
  let choose () = …
end ;;
```

```
let x = M1.choose () in M1.equal (M1.op x x) x ;;
let y = M2.choose () in M2.equal (M2.op x x) x ;;
```

Example: Module Interfaces

```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

dependencies induced by using interfaces

```
module M1 : Magma =
struct
  type t = int
  let op x y = …
  let equal x y = …
  let choose () = …
end
```

```
module M2 : Magma =
struct
  type t = float
  let op x y = …
  let equal x y = …
  let choose () = …
end ;;
```

```
let x = M1.choose () in M1.equal (M1.op x x) x ;;
let y = M2.choose () in M2.equal (M2.op x x) x ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
struct
  type t = X.t * Y.t
  let to_string (x, y) =
    X.to_string x ^ " " ^ Y.to_string y
end
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P. to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Example: Functors

```
module Pair
  (X : sig type t val to_string : t -> string end)
  (Y : sig type t val to_string : t -> string end) =
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 = int_to_string i
end
module String = struct
  type t = string let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Some Observations

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

A Formal Theory of Renaming: Roadmap

1. Programs as ASTs and renamings as operations on them
    - AST 'locations' allow name-independent representations
2. Define a semantic structure that separately captures:
    - Binding resolution information
    - Meta-level program relationships relevant to renaming
    - Information about concrete names
3. Map programs onto these semantic structures
    - formal properties at the 'right level of abstraction'
    - methods for constructing/checking renamings

Viewing Programs Abstractly

$$\text{AST} \quad \sigma \quad : \quad \mathcal{L}oc \to \mathcal{S}yn$$

$\ell \in \mathcal{L}oc$ is a declaration when it is a value identifier ($\sigma(\ell) \in \mathcal{V}$) and there is ℓ' such that

$\sigma(\ell') =$ `val` v_ℓ `: …` $\qquad$ $\sigma(\ell') =$ `let` v_ℓ `= … in …`

$\sigma(\ell') =$ `let` v_ℓ `= …` $\qquad$ $\sigma(\ell') =$ `fun` v_ℓ `-> …`

Non-declaration value identifiers are called references

Renamings as Operations on ASTs

A renaming $\sigma \twoheadrightarrow \sigma'$ is a pair of ASTs σ and σ' such that

1. $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$
2. $\sigma(\ell) \in \mathcal{V} \Leftrightarrow \sigma'(\ell) \in \mathcal{V}$
3. $\sigma(\ell) \notin \mathcal{V} \Rightarrow \sigma(\ell) = \sigma(\ell')$

We define the footprint of a renaming $\sigma \twoheadrightarrow \sigma'$

$$\mathsf{foot}(\sigma, \sigma') = \{\ell \mid \ell \in \mathsf{dom}(\sigma) \wedge \sigma(\ell) \neq \sigma'(\ell)\}$$

We define the dependencies of a renaming $\sigma \twoheadrightarrow \sigma'$

$$\mathsf{deps}(\sigma, \sigma') = \{\ell \mid \ell \in \mathsf{foot}(\sigma, \sigma') \text{ and } \ell \text{ a declaration of } \sigma\}$$

Two Important Questions

1. When is a renaming $\sigma \twoheadrightarrow \sigma'$ valid?

2. For a given AST σ and $\ell \in \text{decl}(\sigma)$, find σ' such that
    - $\sigma \twoheadrightarrow \sigma'$ is valid
    - $\text{foot}(\sigma, \sigma')$ is minimal and contains ℓ

An Abstract Semantic Structure

$$\Sigma \quad = \quad (\rightarrowtail, \mathbb{E}, \rho)$$

Our semantic entities consist of

- A binding resolution function $\rightarrowtail\, : \mathcal{L}oc \rightharpoonup \mathcal{L}oc$
- A value extension relation $\mathbb{E} : \mathcal{L}oc \times \mathcal{L}oc$
- A syntactic reification function $\rho : \mathcal{L}oc \rightharpoonup \mathcal{I}$ mapping locations to identifiers

Interpreting Programs: Example Revisited

```
module A_1 = struct
  let foo_2 = 1
  let bar_3 = 2
end
module B_4 = struct
  include A
  let bar_5 = 3
end
module C_6 = (A : sig val foo_7 : int end)
print_int_8 (A.foo_9 + B.foo_10 + C.foo_11) ;;
print_int_12 (A.bar_13 + B.bar_14) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \} \quad \mathbb{E} = \{(2, 7)\}$

$$\begin{aligned}\rho = \{&1 \mapsto \mathsf{A}, 2 \mapsto \texttt{foo}, 3 \mapsto \texttt{bar}, 4 \mapsto \mathsf{B}, 5 \mapsto \texttt{bar}, 6 \mapsto \mathsf{C}, 7 \mapsto \texttt{foo},\\ &8 \mapsto \texttt{print\_int}, 9 \mapsto \texttt{foo}, 10 \mapsto \texttt{foo}, 11 \mapsto \texttt{foo}, 12 \mapsto \texttt{print\_int},\\ &13 \mapsto \texttt{bar}, 14 \mapsto \texttt{bar}\}\end{aligned}$$

Interpreting Programs: Example Revisited

```
module A₁ = struct
  let foo₂ = 1
  let bar₃ = 2
end
module B₄ = struct
  include A
  let bar₅ = 3
end
module C₆ = (A : sig val foo₇ : int end)
print_int₈ (A.foo₉ + B.foo₁₀ + C.foo₁₁) ;;
print_int₁₂ (A.bar₁₃ + B.bar₁₄) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \} \quad \mathbb{E} = \{(2,7)\}$

$$\rho = \{1 \mapsto \mathsf{A}, 2 \mapsto \texttt{foo}, 3 \mapsto \texttt{bar}, 4 \mapsto \mathsf{B}, 5 \mapsto \texttt{bar}, 6 \mapsto \mathsf{C}, 7 \mapsto \texttt{foo}, 8 \mapsto \texttt{print\_int}, 9 \mapsto \texttt{foo}, 10 \mapsto \texttt{foo}, 11 \mapsto \texttt{foo}, 12 \mapsto \texttt{print\_int}, 13 \mapsto \texttt{bar}, 14 \mapsto \texttt{bar}\}$$

Interpreting Programs: Example Revisited

```
module A₁ = struct
  let foo₂ = 1
  let bar₃ = 2
end
module B₄ = struct
  include A
  let bar₅ = 3
end
module C₆ = (A : sig val foo₇ : int end)
print_int₈ (A.foo₉ + B.foo₁₀ + C.foo₁₁) ;;
print_int₁₂ (A.bar₁₃ + B.bar₁₄) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \} \quad \mathbb{E} = \{(2,7)\}$

$$\rho = \{1 \mapsto \mathsf{A}, 2 \mapsto \texttt{foo}, 3 \mapsto \texttt{bar}, 4 \mapsto \mathsf{B}, 5 \mapsto \texttt{bar}, 6 \mapsto \mathsf{C}, 7 \mapsto \texttt{foo}, \\ 8 \mapsto \texttt{print\_int}, 9 \mapsto \texttt{foo}, 10 \mapsto \texttt{foo}, 11 \mapsto \texttt{foo}, 12 \mapsto \texttt{print\_int}, \\ 13 \mapsto \texttt{bar}, 14 \mapsto \texttt{bar}\}$$

Interpreting Programs: Example Revisited

```
module A₁ = struct
  let foo₂ = 1
  let bar₃ = 2
end
module B₄ = struct
  include A
  let bar₅ = 3
end
module C₆ = (A : sig val foo₇ : int end)
print_int₈ (A.foo₉ + B.foo₁₀ + C.foo₁₁) ;;
print_int₁₂ (A.bar₁₃ + B.bar₁₄) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \} \quad \mathbb{E} = \{(2,7)\}$

$\rho = \{1 \mapsto \mathsf{A}, 2 \mapsto \texttt{foo}, 3 \mapsto \texttt{bar}, 4 \mapsto \mathsf{B}, 5 \mapsto \texttt{bar}, 6 \mapsto \mathsf{C}, 7 \mapsto \texttt{foo},$
$8 \mapsto \texttt{print\_int}, 9 \mapsto \texttt{foo}, 10 \mapsto \texttt{foo}, 11 \mapsto \texttt{foo}, 12 \mapsto \texttt{print\_int},$
$13 \mapsto \texttt{bar}, 14 \mapsto \texttt{bar}\}$

Interpreting Programs: Example Revisited

```
module A₁ = struct
  let foo₂ = 1
  let bar₃ = 2
end
module B₄ = struct
  include A
  let bar₅ = 3
end
module C₆ = (A : sig val foo₇ : int end)
print_int₈ (A.foo₉ + B.foo₁₀ + C.foo₁₁) ;;
print_int₁₂ (A.bar₁₃ + B.bar₁₄) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \} \quad \mathbb{E} = \{(2,7)\}$

$\rho = \{1 \mapsto \mathsf{A}, 2 \mapsto \texttt{foo}, 3 \mapsto \texttt{bar}, 4 \mapsto \mathsf{B}, 5 \mapsto \texttt{bar}, 6 \mapsto \mathsf{C}, 7 \mapsto \texttt{foo}, 8 \mapsto \texttt{print\_int}, 9 \mapsto \texttt{foo}, 10 \mapsto \texttt{foo}, 11 \mapsto \texttt{foo}, 12 \mapsto \texttt{print\_int}, 13 \mapsto \texttt{bar}, 14 \mapsto \texttt{bar}\}$

Interpreting Programs: Example Revisited

```
module A1 = struct
  let foo2 = 1
  let bar3 = 2
end
module B4 = struct
  include A
  let bar5 = 3
end
module C6 = (A : sig val foo7 : int end)
print_int8 (A.foo9 + B.foo10 + C.foo11) ;;
print_int12 (A.bar13 + B.bar14) ;;
```

$\rightarrowtail = \{9 \mapsto 2, 10 \mapsto 2, 11 \mapsto 2, 13 \mapsto 3, 14 \mapsto 5, 8 \mapsto \bot, 12 \mapsto \bot, \}$ $\quad \mathbb{E} = \{ (2,7) \}$

$$\rho = \{1 \mapsto \mathtt{A}, 2 \mapsto \mathtt{foo}, 3 \mapsto \mathtt{bar}, 4 \mapsto \mathtt{B}, 5 \mapsto \mathtt{bar}, 6 \mapsto \mathtt{C}, 7 \mapsto \mathtt{foo}, \\ 8 \mapsto \mathtt{print\_int}, 9 \mapsto \mathtt{foo}, 10 \mapsto \mathtt{foo}, 11 \mapsto \mathtt{foo}, 12 \mapsto \mathtt{print\_int}, \\ 13 \mapsto \mathtt{bar}, 14 \mapsto \mathtt{bar}\}$$

Namelessly Representing Modules and Module Types

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\text{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

$$\text{Structure/Signature} \quad + \quad \text{functor (type)}$$

```
⟦functor
  (X₁ : sig type t val to_string₂ : t -> string end) ->
    functor
      (Y₃ : sig type t val to_string₄ : t -> string end) ->
        struct
        type t = X.t * Y.t
        module Left₅ = X
        module Right₆ = Y
        let to_string₇ (x, y) = …
      end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, (5, \{2\}), (6, \{4\})\}))$$

Namelessly Representing Modules and Module Types

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\mathrm{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

Structure/Signature + functor (type)

```
⟦functor
→ (X₁ : sig type t val to_string₂ : t -> string end) ->
    functor
      (Y₃ : sig type t val to_string₄ : t -> string end) ->
        struct
        type t = X.t * Y.t
        module Left₅ = X
        module Right₆ = Y
        let to_string₇ (x, y) = …
      end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, (5, \{2\}), (6, \{4\})\}))$$

Namelessly Representing Modules and Module Types

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\text{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

$$\text{Structure/Signature} \quad + \quad \text{functor (type)}$$

```
⟦functor
  (X1 : sig type t val to_string2 : t -> string end) ->
    functor
  ⟶(Y3 : sig type t val to_string4 : t -> string end) ->
      struct
      type t = X.t * Y.t
      module Left5 = X
      module Right6 = Y
      let to_string7 (x, y) = …
    end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, (5, \{2\}), (6, \{4\})\}))$$

Namelessly Representing Modules and Module Types

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\text{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

$$\text{Structure/Signature} \quad + \quad \text{functor (type)}$$

```
⟦functor
  (X₁ : sig type t val to_string₂ : t -> string end) ->
    functor
      (Y₃ : sig type t val to_string₄ : t -> string end) ->
        struct
        type t = X.t * Y.t
        module Left₅ = X
        module Right₆ = Y
  ⟶     let to_string₇ (x, y) = …
      end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, (5, \{2\}), (6, \{4\})\}))$$

Namelessly Representing Modules and Module Types

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\text{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

$$\text{Structure/Signature} \quad + \quad \text{functor (type)}$$

```
⟦functor
  (X₁ : sig type t val to_string₂ : t -> string end) ->
    functor
      (Y₃ : sig type t val to_string₄ : t -> string end) ->
        struct
        type t = X.t * Y.t
  ⟶    module Left₅ = X
        module Right₆ = Y
        let to_string₇ (x, y) = …
      end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, \{(5, \{2\}), (6, \{4\})\}\}))$$

We represent in the initial algebra of the functor:

$$F(X) = \wp_{\text{fin}}(X + (\mathcal{L}oc \times X)) + ((\mathcal{L}oc \times X) \times X)$$

$$\text{Structure/Signature} \quad + \quad \text{functor (type)}$$

```
⟦functor
  (X₁ : sig type t val to_string₂ : t -> string end) ->
    functor
      (Y₃ : sig type t val to_string₄ : t -> string end) ->
        struct
        type t = X.t * Y.t
        module Left₅ = X
  ⟶    module Right₆ = Y
        let to_string₇ (x, y) = …
      end⟧
```

$$= ((1, \{2\}), ((3, \{4\}), \{7, (5, \{2\}), (6, \{4\})\}))$$

Semantic operations model effects of syntactic constructs

e.g. join $\otimes_\rho$ produces a value extension:

$$
\begin{aligned}
&(V_1, M_1) \otimes_\rho (V_2, M_2) = \\
&\qquad \{(\ell_1, \ell_2) \mid \ell_1 \in V_1 \land \ell_2 \in V_2 \land \rho(\ell_1) = \rho(\ell_2)\} \cup \\
&\qquad\qquad \bigcup \{\Delta_1 \otimes_\rho \Delta_2 \mid M_1(\ell_1) = \Delta_1 \land M_2(\ell_2) = \Delta_2 \land \rho(\ell_1) = \rho(\ell_2)\}
\end{aligned}
$$

$$((\ell_1, \Delta_1), \Delta_1') \otimes_\rho ((\ell_2, \Delta_2), \Delta_2') = (\Delta_1 \otimes_\rho \Delta_2) \cup (\Delta_1' \otimes_\rho \Delta_2')$$

Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle\Sigma,\Gamma\rangle} = (\Sigma', \Delta)$$

Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} = (\Sigma', \Delta)$$

semantic structures

$$[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} = (\Sigma', \Delta)$$

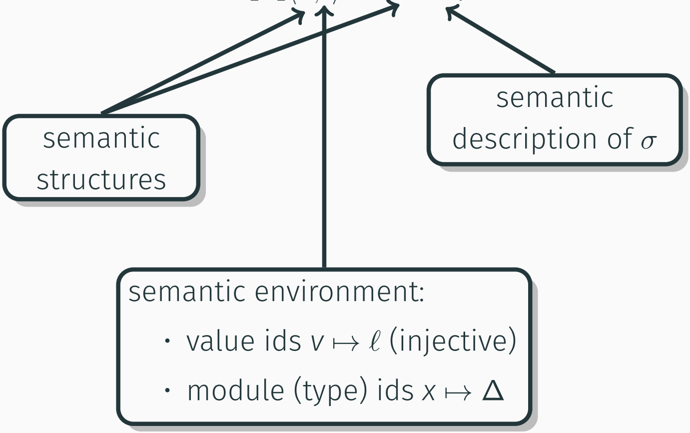
semantic structures

semantic environment:

- value ids $v \mapsto \ell$ (injective)
- module (type) ids $x \mapsto \Delta$

Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} = (\Sigma', \Delta)$$



Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle\Sigma,\Gamma\rangle} = (\Sigma', \Delta)$$

$$[\![\texttt{module}\ x_\ell\ \texttt{=}\ m\ \texttt{;;}\ \sigma]\!]_{\langle\Sigma,\Gamma\rangle} =$$
$$\text{let } (\Sigma', \Delta_1) = [\![m]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } (\Sigma'', \Delta_2) = [\![\sigma]\!]_{\langle\Sigma'[\ell\mapsto x],\Gamma[x\mapsto\Delta_1]\rangle} \text{ in}$$
$$(\Sigma'', (\emptyset, [\ell \mapsto \Delta_1]) + \Delta_2)$$

Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle\Sigma,\Gamma\rangle} = (\Sigma', \Delta)$$

$$[\![\texttt{module } x_\ell \texttt{ = } m \texttt{ ;; } \sigma]\!]_{\langle\Sigma,\Gamma\rangle} =$$
$$\text{let } (\Sigma', \Delta_1) = [\![m]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } (\Sigma'', \Delta_2) = [\![\sigma]\!]_{\langle\Sigma'[\ell\mapsto x],\Gamma[x\mapsto\Delta_1]\rangle} \text{ in}$$
$$(\Sigma'', (\emptyset, [\ell \mapsto \Delta_1]) + \Delta_2)$$

$$[\![\texttt{functor } (x_\ell : M) \texttt{ -> } m]\!]_{\langle\Sigma,\Gamma\rangle} =$$
$$\text{let } (\Sigma', \Delta_1) = [\![M]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } (\Sigma'', \Delta_2) = [\![\sigma]\!]_{\langle\Sigma'[\ell\mapsto x],\Gamma[x\mapsto\Delta_1]\rangle} \text{ in}$$
$$(\Sigma'', ((\ell, \Delta_1), \Delta_2))$$

Constructing the Semantics of Programs

$$[\![\sigma]\!]_{\langle\Sigma,\Gamma\rangle} = (\Sigma', \Delta)$$

$$[\![\texttt{module } x_\ell \texttt{ = } m \texttt{ ;; } \sigma]\!]_{\langle\Sigma,\Gamma\rangle} = \text{let } (\Sigma', \Delta_1) = [\![m]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } (\Sigma'', \Delta_2) = [\![\sigma]\!]_{\langle\Sigma'[\ell\mapsto x],\Gamma[x\mapsto\Delta_1]\rangle} \text{ in } (\Sigma'', (\emptyset, [\ell \mapsto \Delta_1]) + \Delta_2)$$

$$[\![\texttt{functor } (x_\ell : M) \texttt{ -> } m]\!]_{\langle\Sigma,\Gamma\rangle} = \text{let } (\Sigma', \Delta_1) = [\![M]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } (\Sigma'', \Delta_2) = [\![\sigma]\!]_{\langle\Sigma'[\ell\mapsto x],\Gamma[x\mapsto\Delta_1]\rangle} \text{ in } (\Sigma'', ((\ell, \Delta_1), \Delta_2))$$

$$[\![m_1 (m_2)]\!]_{\langle\Sigma,\Gamma\rangle} = \text{let } (\Sigma', ((\ell, \Delta_1), \Delta_2)) = [\![m_1]\!]_{\langle\Sigma,\Gamma\rangle} \text{ in let } ((\rightarrowtail, \mathbb{E}, \rho), \Delta_1') = [\![m_2]\!]_{\langle\Sigma',\Gamma\rangle} \text{ in } ((\rightarrowtail, \mathbb{E} \cup (\Delta_1 \otimes_\rho \Delta_1'), \rho), \Delta_2)$$

Distinguishing Valid Renamings

We define up-to-renaming equivalences on environments and semantic structures

- $\Gamma \sim \Gamma'$ iff $\Gamma(x) = \Gamma'(x)$ and $(\exists v\, \Gamma(v) = \ell) \Leftrightarrow (\exists v\, \Gamma(v) = \ell)$
- $(\rightarrowtail_1, \mathbb{E}_1, \rho_1) \sim (\rightarrowtail_2, \mathbb{E}_2, \rho_2)$ iff $\rightarrowtail_1 = \rightarrowtail_2$, $\mathbb{E}_1 = \mathbb{E}_2$, $\mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$,
  $\rho_1(\ell) \in \mathcal{V} \Leftrightarrow \rho_2(\ell) \in \mathcal{V}$, and if $\rho_1(\ell) \notin \mathcal{V}$ then $\rho_1(\ell) = \rho_2(\ell)$

A valid renaming is one that preserve the equivalence

- $\sigma \twoheadrightarrow \sigma'$ valid w.r.t. $\langle \Sigma, \Gamma \rangle$ iff $\exists\, \Sigma' \sim \Sigma, \Gamma' \sim \Gamma$ such that $\Sigma_1 \sim \Sigma_2$, where $[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} = (\Sigma_1, \Delta_1)$ and $[\![\sigma']\!]_{\langle \Sigma', \Gamma' \rangle} = (\Sigma_2, \Delta_2)$

Distinguishing Valid Renamings

We define up-to-renaming equivalences on environments and semantic structures

- $\Gamma \sim \Gamma'$ iff $\Gamma(x) = \Gamma'(x)$ and $(\exists v\ \Gamma(v) = \ell) \Leftrightarrow (\exists v\ \Gamma(v) = \ell)$
- $(\rightarrowtail_1, \mathbb{E}_1, \rho_1) \sim (\rightarrowtail_2, \mathbb{E}_2, \rho_2)$ iff $\rightarrowtail_1 = \rightarrowtail_2$, $\mathbb{E}_1 = \mathbb{E}_2$, $\mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$,
  $\rho_1(\ell) \in \mathcal{V} \Leftrightarrow \rho_2(\ell) \in \mathcal{V}$, and if $\rho_1(\ell) \notin \mathcal{V}$ then $\rho_1(\ell) = \rho_2(\ell)$

A valid renaming is one that preserve the equivalence

- $\sigma \twoheadrightarrow \sigma'$ valid w.r.t. $\langle \Sigma, \Gamma \rangle$ iff $\exists\, \Sigma' \sim \Sigma, \Gamma' \sim \Gamma$ such that $\Sigma_1 \sim \Sigma_2$, where $[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} = (\Sigma_1, \Delta_1)$ and $[\![\sigma']\!]_{\langle \Sigma', \Gamma' \rangle} = (\Sigma_2, \Delta_2)$
- For whole programs (interpreted w.r.t. $\Sigma_\perp$ and $\Gamma_\perp$), we say
  $P \twoheadrightarrow P'$ valid iff $[\![P]\!] \sim [\![P']\!]$ (when $[\![P]\!]$ and $[\![P']\!]$ defined)

Formal Properties of Valid Renamings

Formal Properties of Valid Renamings

Theorem (1) $P \twoheadrightarrow P$ is valid (when $[\![P]\!]$ defined)
(2) if $P \twoheadrightarrow P'$ is valid then so is $P' \twoheadrightarrow P$
(3) if $P \twoheadrightarrow P'$ and $P' \twoheadrightarrow P''$ are valid then so is $P \twoheadrightarrow P''$

Formal Properties of Valid Renamings

Theorem (1) $P \twoheadrightarrow P$ is valid (when $[\![P]\!]$ defined)
(2) if $P \twoheadrightarrow P'$ is valid then so is $P' \twoheadrightarrow P$
(3) if $P \twoheadrightarrow P'$ and $P' \twoheadrightarrow P''$ are valid then so is $P \twoheadrightarrow P''$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then:
(1) $L = \{\ell \mid \ell \in \mathrm{deps}(P, P') \vee \exists \ell' \in \mathrm{deps}(P, P').\, \ell \rightarrowtail_P \ell'\} \subseteq \mathrm{foot}(P, P')$
(2) $\ell \rightarrowtail \bot$ for all $\ell \in \mathrm{foot}(P, P') \setminus L$

Formal Properties of Valid Renamings

Theorem (1) $P \twoheadrightarrow P$ is valid (when $[\![P]\!]$ defined)
(2) if $P \twoheadrightarrow P'$ is valid then so is $P' \twoheadrightarrow P$
(3) if $P \twoheadrightarrow P'$ and $P' \twoheadrightarrow P''$ are valid then so is $P \twoheadrightarrow P''$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then:
(1) $L = \{\ell \mid \ell \in \mathrm{deps}(P, P') \vee \exists \ell' \in \mathrm{deps}(P, P').\, \ell \rightarrowtail_P \ell'\} \subseteq \mathrm{foot}(P, P')$
(2) $\ell \rightarrowtail \bot$ for all $\ell \in \mathrm{foot}(P, P') \setminus L$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then $\mathrm{deps}(P, P')$ has a partitioning that is a subset of $\mathcal{L}oc_{/\mathbb{E}}$

Formal Properties of Valid Renamings

Theorem (1) $P \twoheadrightarrow P$ is valid (when $[\![P]\!]$ defined)
(2) if $P \twoheadrightarrow P'$ is valid then so is $P' \twoheadrightarrow P$
(3) if $P \twoheadrightarrow P'$ and $P' \twoheadrightarrow P''$ are valid then so is $P \twoheadrightarrow P''$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then:
(1) $L = \{\ell \mid \ell \in \mathrm{deps}(P, P') \vee \exists \ell' \in \mathrm{deps}(P, P').\, \ell \rightarrowtail_P \ell'\} \subseteq \mathrm{foot}(P, P')$
(2) $\ell \rightarrowtail \bot$ for all $\ell \in \mathrm{foot}(P, P') \setminus L$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then $\mathrm{deps}(P, P')$ has a partitioning that is a subset of $\mathcal{L}oc_{/\mathbb{E}}$

Corollary Let $P \twoheadrightarrow P'$ be valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$ and $\ell \in \mathrm{deps}(P, P')$, then $\mathrm{foot}(P, P') \supseteq \{\ell' \mid \ell' \in [\ell]_{\mathbb{E}} \vee \exists \ell'' \in [\ell]_{\mathbb{E}}.\, \ell' \rightarrowtail_P \ell''\}$

Formal Properties of Valid Renamings

Theorem (1) $P \twoheadrightarrow P$ is valid (when $[\![P]\!]$ defined)
(2) if $P \twoheadrightarrow P'$ is valid then so is $P' \twoheadrightarrow P$
(3) if $P \twoheadrightarrow P'$ and $P' \twoheadrightarrow P''$ are valid then so is $P \twoheadrightarrow P''$

Theorem If $P \twoheadrightarrow P'$ is valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$, then:
(1) $L = \{\ell \mid \ell \in \text{deps}(P, P') \vee \exists \ell' \in \text{deps}(P, P').\, \ell \rightarrowtail_P \ell'\} \subseteq \text{foot}(P, P')$
(2) $\ell \rightarrowtail \bot$ for all $\ell \in \text{foot}(P, P') \setminus L$

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

Corollary Let $P \twoheadrightarrow P'$ be valid, with $[\![P]\!] = (\rightarrowtail, \mathbb{E}, \rho)$ and $\ell \in \text{deps}(P, P')$, then $\text{foot}(P, P') \supseteq \{\ell' \mid \ell' \in [\ell]_{\mathbb{E}} \vee \exists \ell'' \in [\ell]_{\mathbb{E}}.\, \ell' \rightarrowtail_P \ell''\}$

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

ROTOR: A Prototype Renaming Tool

- Developed in OCaml itself
  - Allows reuse of the compiler infrastructure
- Approximates the approach discussed
  - Only intra-file binding information provided by compiler
  - Inter-file binding information remains as logical paths
- Tested on 2 large codebases
  - Jane Street public libraries (~900 files, ~3000 test cases)
  - OCaml compiler (~500 files, ~2650 test cases)

Experimental Results: Jane Street Codebase

	Rebuild Succeeded (37%)			Avg.
	Files	Hunks	Deps	Hunks/File
Max	50	128	1127	5.7
Mean	5.0	7.5	24.0	1.3
Mode	3	3	19	1.0

	Rebuild Failed (63%)			Avg.
	Files	Hunks	Deps	Hunks/File
Max	66	305	3365	8
Mean	7.0	12.0	133.4	1.4
Mode	3	3	1	1.0

Experimental Results: OCaml Compiler Codebase

	Rebuild Succeeded (65%)			Avg.
	Files	Hunks	Deps	Hunks/File
Max	19	59	35	15.0
Mean	3.8	5.9	1.6	1.5
Mode	3	3	1	1.0

	Rebuild Failed (31%)			Avg.
	Files	Hunks	Deps	Hunks/File
Max	83	544	56	14.2
Mean	10.2	23.3	10.8	1.7
Mode	4	4	1	1.0

Conclusions

- We have developed a framework for formally describing and reasoning about renaming in OCaml
- Based on a compositional, denotational semantics for a core calculus
- Enables precise statements describing relevant concepts at the right abstraction level
- Implemented a prototype renaming tool based on this approach