Characterising Renaming within OCaml's Module System

Reuben Rowe, Hugo Férée, Simon Thompson, and Scott Owens

Cornell University Department of Computer Science
Seminar in Programming Languages
Wednesday 28th November 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

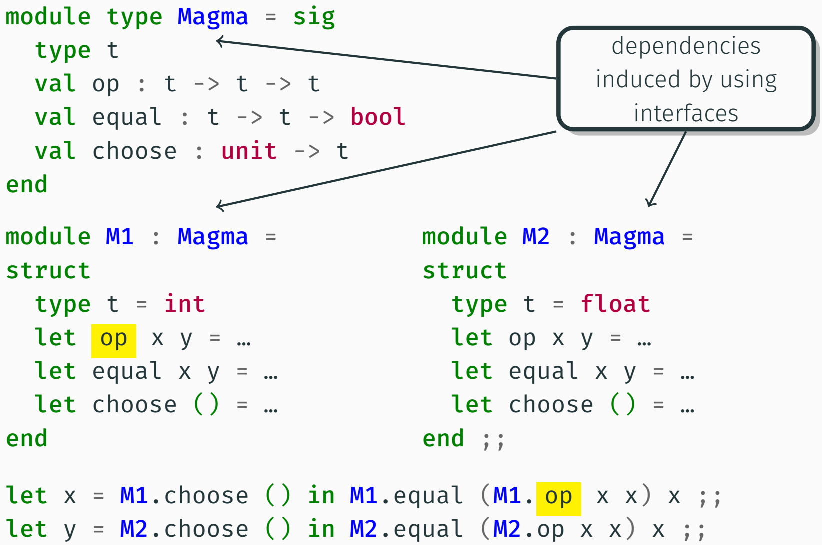
```
module type Magma = sig
  type t
  val op : t -> t -> t
  val equal : t -> t -> bool
  val choose : unit -> t
end
```

```
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 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
```

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 = 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 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 = 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 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 = 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 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 = 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 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 = 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 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 = 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 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 = 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

Renamings as Operations on ASTs

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

Renamings as Operations on ASTs

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

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

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

($\mathcal{V} \subseteq \mathcal{S}yn$ is the set of value identifiers)

Renamings as Operations on ASTs

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

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

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

($\mathcal{V} \subseteq \mathcal{S}yn$ is the set of value identifiers)

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

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

Two Important Questions

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

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

An Abstract Semantic Structure

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

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

An Abstract Semantic Structure

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

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

$[\![\sigma]\!]_{\langle \Gamma, \Sigma \rangle}$ returns the semantics Σ' of the AST σ

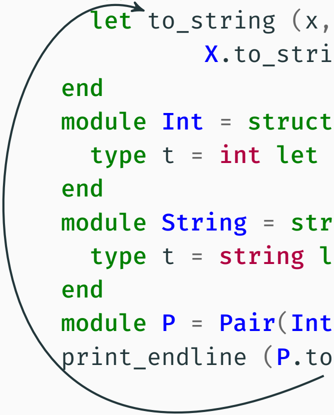
$[\![\cdot]\!]$ parameterized by the 'context' semantics $\langle \Gamma, \Sigma \rangle$

The environment Γ gives meaning to identifiers 'in scope'

Interpreting Programs: Binding Resolution

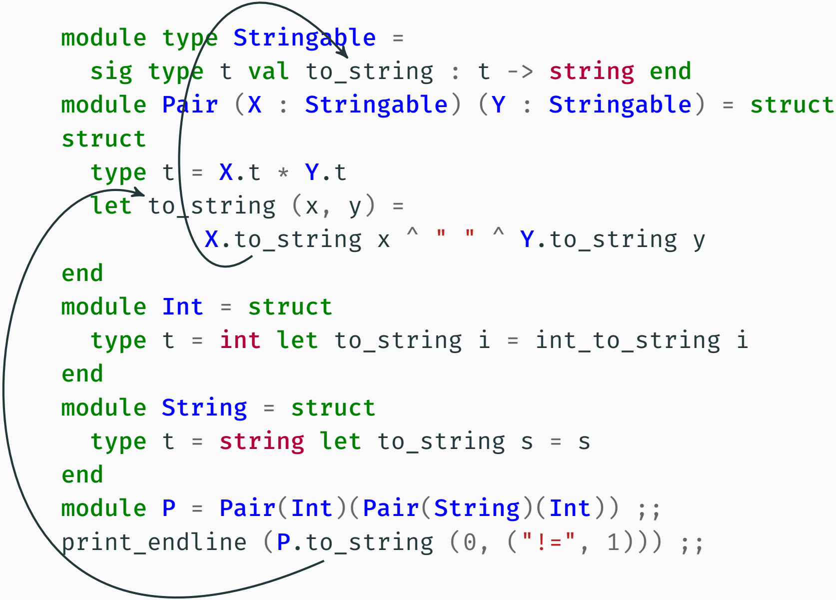
```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
struct
  type t = X.t * Y.t
  let to_string (x, y) =
          X.to_string x ^ " " ^ Y.to_string y
end
module Int = struct
  type t = int let to_string i = int_to_string i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

Interpreting Programs: Binding Resolution



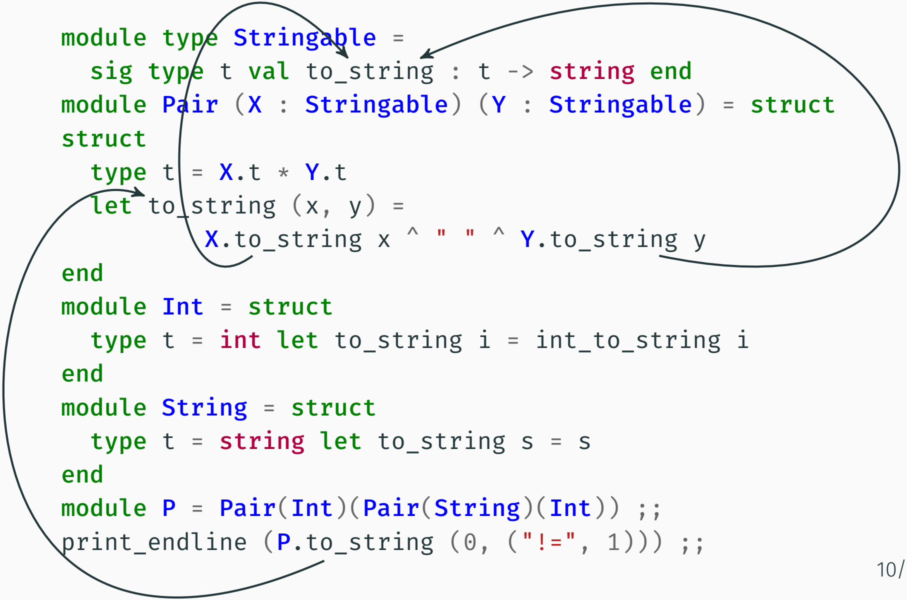
```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
struct
  type t = X.t * Y.t
  let to_string (x, y) =
          X.to_string x ^ " " ^ Y.to_string y
end
module Int = struct
  type t = int let to_string i = int_to_string i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

Interpreting Programs: Binding Resolution



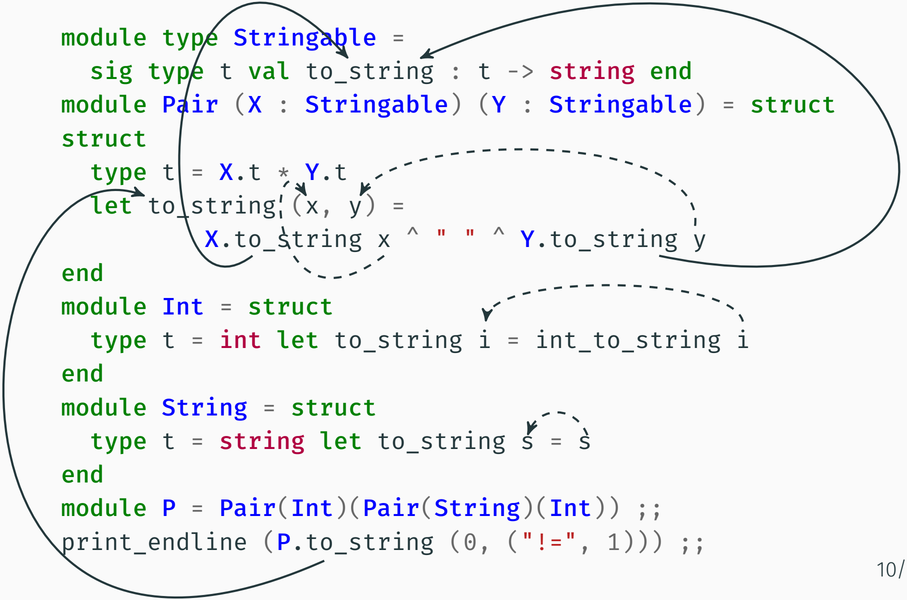
```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
struct
  type t = X.t * Y.t
  let to_string (x, y) =
          X.to_string x ^ " " ^ Y.to_string y
end
module Int = struct
  type t = int let to_string i = int_to_string i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

Interpreting Programs: Binding Resolution



```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
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))) ;;
```

Interpreting Programs: Binding Resolution

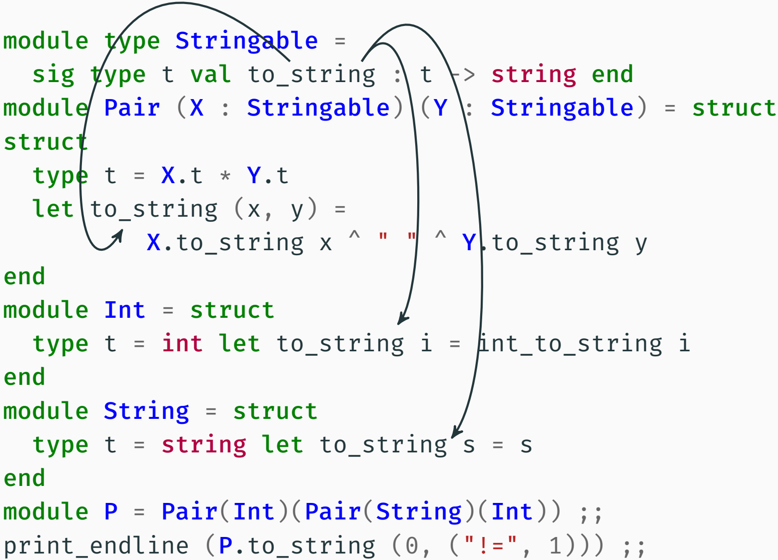


```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
struct
  type t = X.t * Y.t
  let to_string (x, y) =
        X.to_string x ^ " " ^ Y.to_string y
end
module Int = struct
  type t = int let to_string i = int_to_string i
end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

Interpreting Programs: Value Extension Relation

```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
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))) ;;
```

Interpreting Programs: Value Extension Relation



```
module type Stringable =
  sig type t val to_string : t -> string end
module Pair (X : Stringable) (Y : Stringable) = struct
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))) ;;
```

Semantic Equivalence

We define an ‘up-to-renaming’ equivalence

- $(\rightarrowtail_1, \mathbb{E}_1, \rho_1) \sim (\rightarrowtail_2, \mathbb{E}_2, \rho_2)$ iff
    - $\rightarrowtail_1 = \rightarrowtail_2$
    - $\mathbb{E}_1 = \mathbb{E}_2$
    - $\mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$
    - $\rho_1(\ell) \in \mathcal{V} \Leftrightarrow \rho_2(\ell) \in \mathcal{V}$
    - $\rho_1(\ell) \notin \mathcal{V} \Rightarrow \rho_1(\ell) = \rho_2(\ell)$

- $\Gamma \sim \Gamma'$ iff
    - $(\exists v\ \Gamma(v) = \ell) \Leftrightarrow (\exists v\ \Gamma'(v) = \ell)$
    - $\Gamma(x) = \Gamma'(x)$ (*x* is a module or module type identifier)

Distinguishing Valid Renamings

A valid renaming is one that preserve the equivalence:

- $\sigma \twoheadrightarrow \sigma'$ valid w.r.t. $\langle \Sigma, \Gamma \rangle$ when $\exists\, \Sigma' \sim \Sigma, \Gamma' \sim \Gamma$ such that $[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} \sim [\![\sigma']\!]_{\langle \Sigma', \Gamma' \rangle}$

Distinguishing Valid Renamings

A valid renaming is one that preserve the equivalence:

- $\sigma \twoheadrightarrow \sigma'$ valid w.r.t. $\langle \Sigma, \Gamma \rangle$ when $\exists\, \Sigma' \sim \Sigma, \Gamma' \sim \Gamma$ such that $[\![\sigma]\!]_{\langle \Sigma, \Gamma \rangle} \sim [\![\sigma']\!]_{\langle \Sigma', \Gamma' \rangle}$

This does indeed induce an equivalence relation on programs

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 $P \twoheadrightarrow P''$

Constructing Valid Renamings

$\ell \in \mathrm{dom}(\sigma)$ is a declaration of σ when it is a value identifier (i.e. $\sigma(\ell) \in \mathcal{V}$) and there is $\ell' \in \mathrm{dom}(\sigma)$ such that

$$\sigma(\ell') = \texttt{val}\ v_\ell\ \texttt{:}\ \dots \qquad \sigma(\ell') = \texttt{let}\ v_\ell\ \texttt{=}\ \dots\ \texttt{in}\ \dots$$

$$\sigma(\ell') = \texttt{let}\ v_\ell\ \texttt{=}\ \dots \qquad \sigma(\ell') = \texttt{fun}\ v_\ell\ \texttt{->}\ \dots$$

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

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

Characterising Valid Renamings

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

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

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

1. $L \subseteq \mathrm{foot}(P, P')$
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 $\mathcal{L}oc_{/\mathbb{E}}$

Adequacy of the Semantics

We define a denotational semantics $(\!|P|\!)$ 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 $(\!|P|\!)$ 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

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 (68%)			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 (32%)			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