Characterising Renaming within OCaml's Module System: Theory and Implementation

Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, Scott Owens
{r.n.s.rowe,h.feree,s.j.thompson,s.a.owens}@kent.ac.uk
School of Computing, University of Kent, Canterbury, UK

Abstract

We present an abstract, set-theoretic denotational semantics for a significant subset of OCaml and its module system in order to reason about the correctness of renaming value bindings. Our abstract semantics captures information about the binding structure of programs. Crucially for renaming, it also captures information about the relatedness of different declarations that is induced by the use of various different language constructs (e.g. functors, module types and module constraints). Correct renamings are precisely those that preserve this structure. We demonstrate that our semantics allows us to prove various high-level, intuitive properties of renamings. We also show that it is sound with respect to a (domain-theoretic) denotational model of the operational behaviour of programs. This formal framework has been implemented in a prototype refactoring tool for OCaml that performs renaming.

CCS Concepts • **Theory of computation → Abstraction**; **Denotational semantics**; *Program constructs*; *Functional constructs*; • **Software and its engineering → Software maintenance tools**.

Keywords Adequacy, denotational semantics, dependencies, modules, module types, OCaml, refactoring, renaming, static semantics.

ACM Reference Format:
Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, Scott Owens. 2019. Characterising Renaming within OCaml's Module System: Theory and Implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314600

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314600

1 Introduction

Refactoring is the process of changing *how* a program works without changing *what* it does, and is a necessary and ongoing process in both the development and maintenance of any codebase [12]. Whilst individual refactoring steps are often conceptually very simple, applying them in practice can be complex, involving many repeated but subtly varying changes across the entire codebase. Moreover refactorings are, by and large, context sensitive, meaning that carrying them out by hand can be error-prone and the use of general-purpose utilities (even powerful ones such as `grep` and `sed`) is only effective up to a point.

This immediately poses a challenge, but also presents an opportunity. The challenge is how to ensure, or check, a proposed refactoring does not change the behaviour of the program (or does so only in very specific ways). The opportunity is that since refactoring is fundamentally a mechanistic process it is possible to automate it. Indeed, this is desirable in order to avoid human-introduced errors. Our aim in this paper is to outline how we might begin to provide a solution to the dual problem of specifying and verifying the correctness of refactorings and building correct-by-construction automated refactoring tools for OCaml [22, 31].

Renaming is a quintessential refactoring, and so it is on this that we focus as a first step. Specifically, we look at renaming the bindings of values in modules. One might very well be tempted to claim that, since we are in a functional setting, this is simply α-conversion (as in λ-calculus) and thus trivial. This is emphatically not the case. OCaml utilises language constructs, particularly in its module system, that behave in fundamentally different ways to traditional variable binders. Thus, to carry out renaming in OCaml correctly, one must take the meaning of these constructs into account.

Some of the issues are illustrated by the example program in fig. 1 below. This program defines a functor `Pair` that takes two modules as arguments, which must conform to the `Stringable` module type. It also defines two structures `Int` and `String`. It then uses these as arguments in applications of `Pair`, the result of which is bound as the module `P`. Suppose that, for some reason, we wish to rename the `to_string` function in the module `Int`. To do so correctly, we must take the following into account.

(i) Since `Int` is used as the first argument to an application of `Pair`, the `to_string` member of `Pair`'s first parameter must be renamed.

```
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))) ;;
```

Figure 1. Example illustrating issues for renaming.

(ii) The first parameter of **Pair** is declared to be of module type **Stringable**, so `to_string` in **Stringable** must be renamed; similarly for the second parameter, since **Int** is also used as the second argument in an application of **Pair**.

(iii) **String** is also used as an argument in an application of **Pair**, thus its `to_string` member must be renamed too.

(iv) An application of **Pair** is used as an argument to another such application, meaning that we also need to rename `to_string` in the body of **Pair** itself.

(v) Since **P** is bound to the result of applying **Pair**, we must then instances of **P**.`to_string`.

Thus, renaming the binding **Int**.`to_string` actually *depends* on renaming many other bindings in the program: failing to rename any one of them would result in the program being rejected by the compiler. Moreover, this is not simply an artifact of choosing to rename this particular binding; if we were to start with, say, `to_string` in **String** or **Stringable** we would still have to rename the same set of bindings. These bindings are all *mutually* dependent on each other. Consequently, the phenomenon we observe here is distinct from the notion of a refactoring pre-condition [33]. Note that although, in this example, it seemingly suffices to simply 'find-and-replace' all occurrences of `to_string`, this is not generally the case. If the example simply used **String** as the second argument to the (outer) application of **Pair**, then we would not have to rename the binding of `to_string` in the body of the functor.

The salient point in this example is that the various definitions and declarations that must be renamed are not simply references that resolve to a single instance of some syntactic construct in the program. On the contrary, they are themselves binding constructs, which can bind occurrences of identifiers elsewhere in the program. Nevertheless, as noted above, they are connected through certain syntactic constructions, albeit in a different sense to the notion of variable binding with which we are familiar from λ-calculus. Since here names matter, one way of viewing the situation might be to see the mutually dependent declarations (and their referents) all as instances of the same 'free variable' in the program. Free variables cannot be α-renamed, and so this view highlights the gap compared with an understanding of renaming based in the λ-calculus.

One objection to the foregoing analysis might be that the wide-reaching footprint of this refactoring indicates it is not really a renaming, or that it is, in some sense, 'undesirable'. As to the former we would argue that, whilst the changes are extensive, the only syntactic operation that has occurred is to replace one identifier with another—surely, by definition, a renaming. Regarding the latter, other alternatives are indeed possible. One could, for example, localise the changes by introducing a new module expression in the applications of **Pair** that wraps the reference to the **Int** module and reintroduces a binding with the old name.

```
module P = Pair
  (struct include Int let to_string = ⟨⟨new_name⟩⟩ end)
  (Pair(String)
    (struct include Int let to_string = ⟨⟨new_name⟩⟩ end))
```

The point here is not that we are trying to dictate *which* refactoring should be applied in any particular case, but that we are able to characterise precisely which changes of name are (not) refactorings. We can therefore provide a sound foundation for a refactoring tool enabling programmers to safely modify their code.

Our Contributions

In this paper, we propose a formal framework for reasoning about renaming in a significant subset of the OCaml language. We define an abstract semantics for programs in this subset, which captures particular aspects of the structure of programs relevant for renaming value bindings. This comprises *name-invariant* information about binding structure and dependencies between value binding constructs. We then define correctness of renamings in terms of the preservation of this structure. We show that our semantics constitutes a sensible abstraction by proving that it is sound with respect a denotational semantics of the operational behaviour of programs. We use our semantics to develop a *theory of renaming*, in which we characterise correct renamings in a natural and intuitive way and prove that they enjoy desirable (de)composition properties. Finally, we have built a prototype refactoring tool for the full OCaml language based on the concepts elucidated by our framework. We have evaluated our tool on two large real-world codebases.

We have formalised our framework and the renaming theory in the Coq proof assistant [11]. This is included as supplementary material. Also included as supplementary material is an appendix containing a proof sketch of the adequacy result in section 5, and a high-level elaboration of proofs for the renaming theory.

While the paper describes the work in the context of OCaml modules, the approach can be used to understand aspects of (re)naming in other languages, such as Haskell (classes and instances), and Java (interfaces).

Paper Outline. In section 2, we present the subset of OCaml that we study, and formally define renaming. We then present our abstract renaming semantics in section 3, before developing a formal theory of renaming in section 4. Section 5 shows that our renaming semantics is sound with respect to a denotational model of the operational behaviour of our calculus. In section 6 we describe our prototype refactoring tool and experimental evaluation. Section 7 surveys related work, section 8 discusses directions for future work, and section 9 concludes.

2 An OCaml Module Calculus

The subset of OCaml for which we build our formal theory is defined in fig. 2. It extends the calculus considered in [20, 21] and consists, essentially, of a two-level lambda calculus: the 'core' level defines basic values of the language (e.g. functions), whereas the other comprises the module system. The module system contains structures, functors, and module types (with module constraints and destructive module substitutions), along with `include` statements. Since value types do not interact with the renaming that we consider, we do not include a language for defining them. Thus, in order for our calculus to count as valid actual OCaml code, we use OCaml's underscore syntax for anonymous type variables in value declarations in signatures, e.g. `sig val foo : _ end`.

Other features of OCaml's module system that we do not model, but which nonetheless interact with renaming, include: (local) open statements; recursive and first-class modules; module type extraction; and type-level module aliases. All but the first of these are extensions to the core OCaml language. We leave the treatment of these language extensions to future work.

We have assumed (disjoint) sets $\mathcal{M}$, $\mathcal{T}$, and $\mathcal{V}$ of module, module type, and value identifiers, respectively. These are ranged over by x, t, and v, respectively, and we use $\imath$ to range over the set $\mathcal{I} = \mathcal{M} + \mathcal{T} + \mathcal{V}$ of all identifiers. In real OCaml, both module identifiers and module type identifiers belong to the same lexical class. However, it will be convenient to distinguish them in our formalism. In any case it is syntactically unambiguous when such an identifier acts as a module identifier and when it acts as a module type identifier; thus we do not lose any generality in making this distinction.

2.1 Renaming Operations

To formalise the notion of carrying out renaming, we will take (fragments of) programs to be abstract syntax trees (ASTs). It will be convenient for us to consider ASTs as functions over some set $\mathcal{L}$ of locations (ranged over by ℓ) returning local syntactic information. That is, for locations denoting internal nodes of the AST the function maps to the locations of the roots of the child subtrees and indicates which compound syntactic production is applied. For locations denoting leaves the function maps to the relevant identifier or constant. We will also assume that there is some *null* location $\bot \in \mathcal{L}$ that does not denote any location in any AST. This will be used by our semantics to indicate that a reference does not resolve to anything in a program. Although ASTs impose additional hierarchical structure on locations, we leave this implicit and do not further specify their concrete nature.

Definition 1. One program (fragment) σ' is the result of renaming another such σ, when: (i) $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$; (ii) $\sigma(\ell) \in \mathcal{V} \Leftrightarrow \sigma'(\ell) \in \mathcal{V}$; and (iii) if $\sigma(\ell) \notin \mathcal{V}$ then $\sigma(\ell) = \sigma'(\ell)$. In this case, we call the pair (σ, σ') a *renaming* and write $\sigma \hookrightarrow \sigma'$.

That is, renaming is only allowed to replace value identifiers by other value identifiers, and must otherwise leave the program (fragment) unchanged.

We now define a number of syntactic concepts that will be useful in describing the action of renamings. Firstly, we consider the notion of the footprint of a renaming. This is all the locations in the program that are affected, or changed, by the renaming.

Definition 2 (Footprints)**.** The *footprint* $\varphi(\sigma, \sigma')$ of a renaming $\sigma \hookrightarrow \sigma'$ is defined to be the set of locations (necessarily in both σ and σ') that are changed by the renaming: $\varphi(\sigma, \sigma') = \{\ell \mid \ell \in \mathsf{dom}(\sigma) \wedge \sigma(\ell) \neq \sigma'(\ell)\}$. We write $\sigma \overset{\ell}{\hookrightarrow} \sigma'$ when ℓ is in the footprint of the renaming, and $\sigma \overset{v/\ell}{\hookrightarrow} \sigma'$ when moreover $\sigma'(\ell) = v$.

A general problem we are interested in is the following: given the location ℓ of some identifier in a program P and an identifier v that we wish to rename it to, can we produce a program P' such that $P \overset{v/\ell}{\hookrightarrow} P'$ is a *valid* renaming? Moreover, we are usually interested in finding such a P' that also *minimises* the footprint of the renaming. One purpose of the semantics that we define in section 3 is to enable us to provide solutions to this problem, as well as an effective abstraction of what constitutes validity for renaming.

Besides footprints, we are also interested in what we call the *dependencies* of a renaming. These are all the binding declarations modified by a renaming. In both the following definition and when presenting example syntax below, we will use subscripts on identifiers to indicate their unique position in the AST. In particular, numeric subscripts should not be taken to be part of the identifier itself.

Definition 3 (Declarations)**.** The set $\mathsf{decl}(\sigma)$ of (value) *declarations* in a program (fragment) σ is the set of all locations $\ell \in \mathsf{dom}(\sigma)$ for which there exists $\ell' \in \mathsf{dom}(\sigma)$ such that either: $\sigma(\ell') = \texttt{val}\ v_\ell\ \texttt{:}\ \_;;$, $\sigma(\ell') = \texttt{let}\ v_\ell = e;;$, $\sigma(\ell') = \texttt{let}\ v_\ell = e\ \texttt{in}\ e';;$, or $\sigma(\ell') = \texttt{fun}\ v_\ell\ \texttt{->}\ e;;$.

Module Paths $p ::= x \mid p.x$ **Extended Module Paths** $q ::= x \mid q.x \mid q(q)$ **Value Expressions** $e ::= v \mid p.v \mid \texttt{let}\ v = e\ \texttt{in}\ e \mid \texttt{fun}\ v\ \texttt{->}\ e \mid e\,e$ **Programs** $P ::= e \mid \texttt{module}\ x = m\ \texttt{;;}\ P$

Module Types $M ::= t \mid p.t \mid \texttt{sig}\ S\ \texttt{end} \mid \texttt{functor}\ (x : M)\ \texttt{->}\ M \mid M\ \texttt{with module}\ x = q \mid M\ \texttt{with module}\ x := q$

Signature Body $S ::= \varepsilon \mid D\ \texttt{;;}\ S$ **Signature Components** $D ::= \texttt{val}\ v : \_ \mid \texttt{module}\ x : M \mid \texttt{module type}\ t \mid \texttt{module type}\ t = M \mid \texttt{include}\ M$

Module Expressions $m ::= p \mid \texttt{struct}\ s\ \texttt{end} \mid \texttt{functor}\ (x : M)\ \texttt{->}\ m \mid m\,(m) \mid m : M$

Structure Body $s ::= \varepsilon \mid d\ \texttt{;;}\ s$ **Structure Components** $d ::= \texttt{let}\ v = e \mid \texttt{module}\ x = m \mid \texttt{module type}\ t = M \mid \texttt{include}\ m$

Figure 2. Syntax of a core calculus for OCaml with modules.

Definition 4 (Dependencies)**.** The *dependencies* $\delta(\sigma, \sigma')$ of $\sigma \hookrightarrow \sigma'$ are defined by $\delta(\sigma, \sigma') = \varphi(\sigma, \sigma') \cap \mathsf{decl}(\sigma)$.

Intuitively, the dependencies should be the key piece of (syntactic) information required to characterise a renaming since we expect the remaining locations in the program that must be renamed to be simply those references that resolve to one of the dependencies.

We also formally define the references of a program (fragment) as follows.

Definition 5 (References)**.** The set of (value) *references* of a program (fragment) σ is the set of locations $\ell \in \mathsf{dom}(\sigma)$ such that $\sigma(\ell) \in \mathcal{V}$ and $\ell \notin \mathsf{decl}(\sigma)$.

Notice that both the footprint and the dependencies of composite renamings are bounded by the footprints and dependencies, respectively, of their individual component renamings.

Proposition 1. *For renamings $\sigma \hookrightarrow \sigma'$ and $\sigma' \hookrightarrow \sigma''$:*
(i) $\varphi(\sigma, \sigma'') \subseteq \varphi(\sigma, \sigma') \cup \varphi(\sigma', \sigma'')$.
(ii) $\delta(\sigma, \sigma'') \subseteq \delta(\sigma, \sigma') \cup \delta(\sigma', \sigma'')$.

3 A Static Semantics for Renaming

In this section, we define a set-theoretic semantics for programs in our calculus that will allow us to reason about renaming values. The entities that comprise the meaning of a program are sets of (possibly nested) tuples of elements. Note that this allows us to also talk about functions, since these can be described by sets of ordered pairs. The semantics jointly describes binding resolution and dependency information in a name-invariant manner (using AST locations), and represents name-relevant information separately.

In the following presentation, we use standard notation for function update: i.e. $f[a \mapsto b]$ denotes the function that behaves like f except that $f(a) = b$. $f[a \mapsto b \mid a \in A]$ denotes the function that behaves like f except that $f(a) = b$ for all $a \in A$, and $f \setminus A$ the (partial) function that behaves like f but only has domain $\mathsf{dom}(f) \setminus A$.

3.1 Semantic Elements

Our abstract semantics will consist of the following entities.

Binding Resolution is a function that maps the locations of uses of identifiers to binding instances of identifiers.

Definition 6 (Binding resolution)**.** A binding resolution function $\rightarrowtail$ is a partial function between locations (we assume it does not map the null location $\bot$). We write $\ell \rightarrowtail \ell'$ instead of $\rightarrowtail(\ell) = \ell'$, and say that ℓ *resolves to* ℓ'.

The idea is that locations in the domain of the function will represent precisely the *references* in a program, and the function will describe the declaration that each reference resolves to.

Syntactic Characteristics that are captured by our semantics comprise the identifiers that are found at given locations. This allows for the locations of binding instances of like identifiers to be related (cf. section 3.2 below).

Definition 7 (Syntactic Reification)**.** A syntactic reification function $\rho : \mathcal{L} \rightharpoonup \mathcal{I}$ is a partial mapping from locations to identifiers (we assume it does not map the null location $\bot$). We write $\mathsf{dom}_{\mathcal{V}}(\rho)$ to denote the set $\{\ell \mid \rho(\ell) \in \mathcal{V}\}$.

We can view syntactic reification functions as capturing a restricted view of ASTs, giving information only about those leaves that contain identifiers. The syntactic reification function can be used to give additional information, over and above the binding resolution function, about the declarations in a program (specifically, those which are never referenced).

Value Extensions capture sets of declarations that are all different facets of the same logical concept modelled in the program. For example, a program may contain many different functions named `compare` that act on values of various different data types, which might be related through the use of different signatures declaring values named `compare`, or the application of various functors to different modules. Although the different declarations may be distributed widely throughout the program, they all model a single concept or entity in the mind of the programmer or architecture of the system. These entities are high-level abstractions encoded via the global structure of program. When we rename a declaration, we must rename all parts of the program that constitute the logical entity of which it is part. The difficulty inherent in renaming in OCaml arises since these high-level entities are not necessarily immediately evident, nor necessarily localised in the source code.

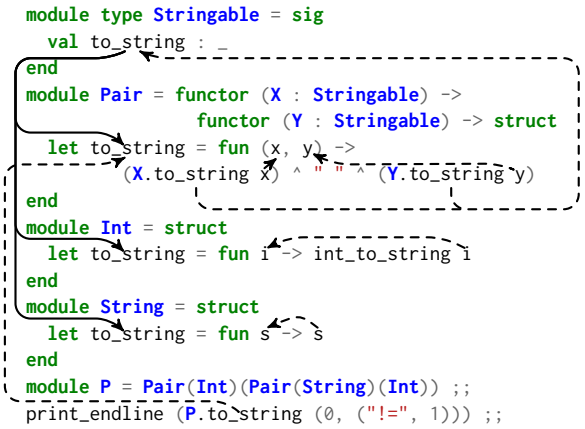

Figure 3. Graphical representation of the semantics.

We call such collections of declarations the *extension* of a high-level program abstraction. Ultimately, the extension is modelled by an equivalence class. However the structural relationships between the elements of an extension are more fine-grained and it is these that we capture, using a binary relation that we call a 'kernel'. Taking the reflexive, symmetric and transitive closure of this kernel results in the equivalence relation whose equivalence classes we take to model extensions.

Definition 8 (Value Extension Kernel)**.** A value extension kernel $\mathbb{E}$ is a binary relation on locations. $\hat{\mathbb{E}}$ denotes the reflexive, symmetric and transitive closure of $\mathbb{E}$.

We use value extensions to capture high-level, global structures present in a program, as expressed in proposition 6 below. For a location ℓ, we denote the $\hat{\mathbb{E}}$-equivalence class containing ℓ by $[\ell]_{\hat{\mathbb{E}}}$. We also denote by $\mathcal{L}_{/\hat{\mathbb{E}}}$ the quotient of $\mathcal{L}$ by $\hat{\mathbb{E}}$, i.e. the partitioning of the set of locations into $\hat{\mathbb{E}}$-equivalence classes.

To give an intuition as to how these elements are used, we show in fig. 3 a visual representation of the binding resolution function and value extension kernel that would be derived for the example of fig. 1 as expressed in our OCaml calculus (i.e. value type components are elided and functions and functors are written out 'long-hand'; for clarity, we still assume pairs and an infix string concatenation operation). Dashed arrows depict the binding resolution mappings, and solid arrows show pairs in the value extension kernel.

3.2 Semantic Descriptions

In constructing the semantics of programs, we will need to keep track of the binding structure of modules and module types. We do so using *semantic descriptions*, which capture the locations of binding instances of identifiers and the nested structure of modules and module types. We distinguish two kinds of semantic descriptions: structural descriptions describe structures and signatures, while functorial descriptions describe functors and functor types.

Definition 9 (Semantic descriptions)**.** Semantic descriptions Δ, and their constituent *components* c, are objects defined inductively by:

$$\Delta ::= \{c_1, \ldots, c_n\} \mid (\ell{:}\Delta){\to}\Delta \qquad\qquad c ::= \ell \mid (\ell, \Delta)$$

We use the meta-variable D to range over *structural* descriptions, i.e. those of the form $\{c_1, \ldots, c_n\}$, and write $\lfloor D \rfloor$ to denote the set $\{\ell \mid \ell \in D\}$. Descriptions of the form $(\ell{:}\Delta){\to}\Delta'$ are called *functorial*. We write $\mathcal{D}$ for the set of all semantic descriptions.

Basic components, comprising of simply a location, capture the locations of instances of identifiers bound to values. Components of the form (ℓ, Δ) represent sub-modules or sub-module types, comprising a description of the subcomponent along with the location of its binding, i.e. the instance of the identifier to which it is bound. Structural descriptions, which are sets of such components, thus describe the binding structure of structures and signatures. Functorial descriptions $(\ell{:}\Delta){\to}\Delta'$ capture that of functors and functor types: the left-hand member $\ell{:}\Delta$ captures the location of the parameter of the functor or functor type, along with a description of its declared module type; the right-hand member of the pair Δ' describes the body.

Example 1. Consider the **`Stringable`** module type and the **`Int`** and **`String`** modules in the program of fig. 1.

```
module type Stringable = sig val to_string₁ : _ end
module Int  = struct let to_string₅ i = int_to_string i end
module String = struct let to_string₆ s = s end
```

The numerical subscripts on identifiers indicate the abstract locations of declarations in the program. The corresponding semantic descriptions are $D_{\texttt{Stringable}} = \{1\}$, $D_{\texttt{Int}} = \{5\}$, and $D_{\texttt{String}} = \{6\}$. For the **`Pair`** functor

```
module Pair =
  functor (X₂ : Stringable) -> functor (Y₃ : Stringable) ->
    let to_string₄ = fun (x, y) -> ... end
```

the corresponding semantic description is the following.

$$\begin{aligned} D_{\texttt{Pair}} &= (2{:}D_{\texttt{Stringable}}){\to}(3{:}D_{\texttt{Stringable}}){\to}\{4\} \\ &= (2{:}\{1\}){\to}(3{:}\{1\}){\to}\{4\} \end{aligned}$$

The syntactic reification function ρ for this program has the obvious action on these declarations: $\rho(2) = \texttt{X}$, $\rho(3) = \texttt{Y}$, and $\rho(1) = \rho(4) = \rho(5) = \rho(6) = \texttt{to\_string}$.

Semantic Operations. We now describe a number of operations on semantic descriptions. Before giving the formal definition of each operation, we explain its purpose and give an example.

The value extension kernel, capturing the relationships between the declarations, is computed via a semantic join operation $\otimes_\rho$ on descriptions, which is parameterised by a syntactic reification function. This operation pairs up basic components in two semantic descriptions that syntactically

reify to the same identifier. For example, applications of functors induce dependencies between the declarations in the module type of the parameter, and corresponding bindings in the module used as the argument.

Example 2. The nested functor application in fig. 3, that is `Pair(String)(Int)`, links the declaration of `to_string` in the module type `Stringable` to those of `to_string` in the `String` and `Int` modules. Joining the description of the declared module type, `Stringable`, of `Pair`'s first parameter with the description of the first argument, `String`, gives

$$D_{\texttt{Stringable}} \otimes_\rho D_{\texttt{String}} = \{1\} \otimes_\rho \{6\} = \{(1, 6)\}$$

since $\rho(1) = \rho(6) = \texttt{to\_string}$. Similarly, joining the description of the declared module type, `Stringable`, of `Pair`'s second parameter with the description of the second argument, `Int`, gives

$$D_{\texttt{Stringable}} \otimes_\rho D_{\texttt{Int}} = \{1\} \otimes_\rho \{5\} = \{(1, 5)\}$$

for the same reason. The description of the result of a functor application is simply the description of the functor body, thus the description of `Pair(String)(Int)` is $\{4\}$. The outer functor application `Pair(Int)(Pair(String)(Int))` then relates the declaration of `to_string` in `Stringable` to that in the body of the `Pair` functor itself, via the join operation as follows.

$$D_{\texttt{Stringable}} \otimes_\rho \{4\} = \{1\} \otimes_\rho \{4\} = \{(1, 4)\}$$

The join operation is defined as follows, where writing $\rho(\ell)$ for a reification function ρ and a location ℓ asserts that ρ is defined on ℓ.

Definition 10 (Description Join)**.** For a given syntactic reification function ρ, the description join operation $\otimes_\rho$ is a binary operation on descriptions that produces a binary relation on locations and is defined inductively as follows:

$$\begin{aligned} D_1 \otimes_\rho D_2 = {} & \{(\ell_1, \ell_2) \mid \ell_1 \in D_1 \wedge \ell_2 \in D_2 \wedge \rho(\ell_1) = \rho(\ell_2)\} \\ & \cup \{(\ell_1, \ell_2) \mid \exists (\ell, \Delta_1) \in D_1, (\ell', \Delta_2) \in D_2. \\ & \qquad \rho(\ell) = \rho(\ell') \wedge (\ell_1, \ell_2) \in \Delta_1 \otimes_\rho \Delta_2\} \\ (\ell_1{:}\Delta_1){\to}\Delta_1' \otimes_\rho (\ell_2{:}\Delta_2){\to}\Delta_2' = {} & (\Delta_1 \otimes_\rho \Delta_2) \cup (\Delta_1' \otimes_\rho \Delta_2') \\ \Delta \otimes_\rho \Delta' = {} & \emptyset \qquad \text{otherwise} \end{aligned}$$

The join of two structural descriptions consists of two parts: first, basic components that reify to the same identifier are related; second, the join is applied recursively to subcomponents whose bindings reify to the same identifier. The join of two functorial descriptions is given point-wise: the join of the parameters is combined with the join of the bodies. Joins of dissimilar descriptions result in the empty relation.

To build semantic descriptions compositionally, we use a number of semantic operations that correspond to the various syntactic constructions that are used to define modules and module types.

To model the effect of `include` statements, we define a superposition operation $\oplus_\rho$ on structural descriptions that combines the elements of two descriptions, preferring those of its second argument when identifiers coincide.

Example 3. Consider the following modules.

```
module A = struct let foo₁ = ...;; let bar₂ = ...;; end
module B = struct include A let bar₃ = ...;; end
```

A semantic description of the module `A` consists of the set $D_{\texttt{A}} = \{1, 2\}$, while the remainder of the body of module `B` after the `include` statement consists of the set $D_{\text{body}} = \{3\}$. To form a description of the module `B`, we can superpose $D_{\texttt{A}}$ and D_{body} with respect to a reification function ρ' containing mappings $\rho'(1) = \texttt{foo}$, and $\rho'(2) = \rho'(3) = \texttt{bar}$. That is $D_{\texttt{B}} = D_{\texttt{A}} \oplus_{\rho'} D_{\text{body}} = \{1, 3\}$. Here, the location 3 from D_{body} is chosen over 2 from $D_{\texttt{A}}$ since ρ' maps them both to the same identifier.

Definition 11 (Description Superposition)**.** The superposition operation $\oplus_\rho$ on structural descriptions is defined by:

$$\begin{aligned} D \oplus_\rho D' = {} & D' \cup \{\ell \mid \ell \in D \wedge \forall \ell' \in D'.\, \rho(\ell) \neq \rho(\ell')\} \\ & \cup \{(\ell, \Delta) \mid (\ell, \Delta) \in D \wedge \forall (\ell', \Delta') \in D'.\, \rho(\ell) \neq \rho(\ell')\} \end{aligned}$$

Superposition $D \oplus_\rho D'$ augments D' with those components from D for which there is no corresponding component in D' whose binding reifies to the same identifier.

To model module type annotations $m : M$, we define a modulation operation $\blacktriangleright_\rho$ on semantic descriptions. This operation modifies the description of a module m according to the description of its module type M in two ways. First, it removes any (sub)components for which there does not exist a corresponding component in the module type with a similarly named binding. Second, it adds those subcomponents from the description of the module type for which there is no similarly named component in the module description.

Example 4. Consider the following module types defining a weakening of the type of the `Pair` functor.

```
module type Stringable2 =
  sig val to_string₇ : _ ;; val from_string₈ : _ ;; end
module type WeakPair = functor (X₉ : Stringable2) ->
  functor (Y₁₀ : Stringable2) -> sig end
```

$D_{\texttt{Weak}} = (9{:}D_{\texttt{Stringable2}}){\to}(10{:}D_{\texttt{Stringable2}}){\to}\emptyset$ describes the module type `WeakPair`, where $D_{\texttt{Stringable2}} = \{7, 8\}$. Assuming the syntactic reification function ρ from example 1 above also contains mappings reflecting the identifiers occurring in `Stringable2` and `WeakPair`, then `M = Pair : WeakPair` is described by the result of the applying the modulation operation, as follows.

$$D_{\texttt{M}} = D_{\texttt{Pair}} \blacktriangleright_\rho D_{\texttt{Weak}} = (2{:}\{1, 8\}){\to}(3{:}\{1, 8\}){\to}\emptyset$$

In the result, the description of the body of $D_{\texttt{Pair}}$ has been restricted, but the descriptions of the parameters have been augmented by the additional `from_string` declaration (location 8) in the module types of the parameters in $D_{\texttt{Weak}}$.

Definition 12 (Description Modulation)**.** The description modulation operation $\blacktriangleright_\rho$ is a binary operation on semantic descriptions defined inductively as follows:

$$\begin{aligned} D \blacktriangleright_\rho D' = {} & \{\ell \mid \ell \in D \wedge \exists \ell' \in D'.\, \rho(\ell) = \rho(\ell')\} \\ & \cup \{\ell' \mid \ell' \in D' \wedge \forall \ell \in D.\, \rho(\ell) \neq \rho(\ell')\} \\ & \cup \{(\ell, \Delta \blacktriangleright_\rho \Delta') \mid (\ell, \Delta) \in D \wedge \\ & \qquad \exists \ell'.\, (\ell', \Delta') \in D' \wedge \rho(\ell) = \rho(\ell')\} \\ & \cup \{(\ell', \Delta') \mid (\ell', \Delta') \in D' \wedge \forall (\ell, \Delta) \in D.\, \rho(\ell) \neq \rho(\ell')\} \\ (\ell{:}\Delta_1){\to}\Delta_2 \blacktriangleright_\rho (\ell'{:}\Delta_1'){\to}\Delta_2' = {} & (\ell{:}(\Delta_1 \blacktriangleright_\rho \Delta_1')){\to}(\Delta_2 \blacktriangleright_\rho \Delta_2') \\ \Delta \blacktriangleright_\rho \Delta' = {} & \emptyset \qquad \text{otherwise} \end{aligned}$$

Finally, to model the effects of module type constraints we define two more operations. In selective modulation, $\Delta_1 \blacktriangleleft_\rho (x{:}\Delta_2)$, only subcomponents (ℓ, Δ') of Δ_1 for which ℓ reifies to the module identifier x are modulated (by Δ_2). This models how a module constraint modifies the subcomponents of a module type. The filtering operation $\setminus_\rho$ removes subcomponents from a structural description whose binding reifies to a given identifier, corresponding to a destructive module substitution on a module type.

Example 5. Suppose we have a module type `Set` and a module `Int` defined by

```
module type Set = sig
  module Elt11 : Stringable ;; val empty12 : _ ;; end
Int2 = struct include Int;; let from_string13 = ...;; end
```

with $D_{\texttt{Set}} = \{12, (11, D_{\texttt{Stringable}})\}$ and $D_{\texttt{Int2}} = \{5, 13\}$. Again assuming ρ also contains mappings reflecting the identifiers occurring in `Set` and `Int` above, the description of `IntSet = Set with module Elt = Int2` is given by the result of selective modulation, as follows.

$$\begin{aligned} D_{\texttt{IntSet}} &= D_{\texttt{Set}} \blacktriangleleft_\rho (\texttt{Elt}{:}D_{\texttt{Int2}}) \\ &= \{12, (11, (D_{\texttt{Stringable}} \blacktriangleright_\rho D_{\texttt{Int2}}))\} \\ &= \{12, (11, \{1, 13\})\} \end{aligned}$$

To compute the description of the module type given by `IntSet2 = Set with module Elt := Int2`, we use filtering: $D_{\texttt{IntSet2}} = D_{\texttt{Set}} \setminus_\rho \texttt{Elt} = \{12\}$.

Definition 13 (Selective Modulation)**.** The selective modulation operation is a binary operation $\Delta \blacktriangleleft_\rho (x{:}\Delta')$ on semantic descriptions with respect to a module identifier, and is defined by:

$$\begin{aligned} D \blacktriangleleft_\rho (x{:}\Delta') = {} & \{\ell \mid \ell \in D\} \cup \{(\ell, \Delta) \mid (\ell, \Delta) \in D \wedge \rho(\ell) \neq x\} \\ & \cup \{(\ell, \Delta \blacktriangleright_\rho \Delta') \mid (\ell, \Delta) \in D \wedge \rho(\ell) = x\} \\ (\ell{:}\Delta_1){\to}\Delta_2 \blacktriangleleft_\rho (x{:}\Delta') = {} & \emptyset \end{aligned}$$

Definition 14 (Description Filtering)**.** The function $\setminus_\rho$ on semantic descriptions and (module) identifiers is defined by:

$$\begin{aligned} D \setminus_\rho x = {} & \{\ell \mid \ell \in D\} \cup \{(\ell, \Delta) \mid (\ell, \Delta) \in D \wedge \rho(\ell) \neq x\} \\ (\ell{:}\Delta){\to}\Delta' \setminus_\rho x = {} & \emptyset \end{aligned}$$

3.3 Semantic Environments

When constructing the semantics of programs, we will also need to keep track of the binding locations and descriptions of bound values, modules and module types. We do this using an *environment*, which is a pair $(\Gamma_\mathcal{V}, \Gamma_\mathcal{M})$ of functions $\Gamma_\mathcal{V} : \mathcal{V} \to \mathcal{L}$ and $\Gamma_\mathcal{M} : \mathcal{M} \cup \mathcal{T} \to \mathcal{D}$ that map value identifiers to the location in the program context to which they are bound, and map module and module type identifiers to semantic descriptions of the module or module type, respectively to which they are bound. We also require $\Gamma_\mathcal{V}$ to be injective on $\mathcal{L} \setminus \{\bot\}$, i.e. $\Gamma_\mathcal{V}(v) = \Gamma_\mathcal{V}(v') \neq \bot \Rightarrow v = v'$.

For notational convenience, we will write $\Gamma(v)$, $\Gamma(t)$, and $\Gamma(x)$ for $\Gamma_\mathcal{V}(v)$, $\Gamma_\mathcal{M}(t)$, and $\Gamma_\mathcal{M}(x)$, respectively. Similarly, we will write $\Gamma[v \mapsto \ell]$, $\Gamma[t \mapsto \Delta]$, and $\Gamma[x \mapsto \Delta]$ for $(\Gamma_\mathcal{V}[v \mapsto \ell], \Gamma_\mathcal{M})$, $(\Gamma_\mathcal{V}, \Gamma_\mathcal{M}[t \mapsto \Delta])$, and $(\Gamma_\mathcal{V}, \Gamma_\mathcal{M}[x \mapsto \Delta])$, respectively.

We say that a structural description D is *proper* for a reification function ρ when it satisfies: (i) $\rho(\ell) \in \mathcal{V}$ for all $\ell \in D$; (ii) $\rho(\ell) \in \mathcal{M} \cup \mathcal{T}$ for all $(\ell, \Delta) \in D$; and (iii) when $\ell, \ell' \in D$ or $(\ell, \Delta), (\ell', \Delta') \in D$ for distinct locations ℓ and ℓ', then $\rho(\ell) \neq \rho(\ell')$. That is, each location in D corresponds to a *unique* identifier under ρ. In this case, we may treat it like a partial semantic environment and combine it with an existing environment Γ (written $\Gamma +_\rho D$) as follows:

$$(\Gamma +_\rho D)(\iota) = \begin{cases} \ell & \text{if } \ell \in D \text{ and } \rho(\ell) = \iota \\ \Delta & \text{if } (\ell, \Delta) \in D \text{ and } \rho(\ell) = \iota \\ \Gamma(\iota) & \text{otherwise} \end{cases}$$

3.4 Semantics of Programs

Our static renaming semantics interprets programs as tuples $(\rightarrowtail, \mathbb{E}, \rho)$ comprising a binding resolution function, a value extension kernel, and a syntactic reification function. We use Σ to range over such tuples. We may also write $\Sigma_\rightarrowtail$, $\Sigma_\mathbb{E}$, and Σ_ρ to denote the individual respective components of Σ.

To define the semantics of programs, we use two sorts of judgement, $\Sigma; \Gamma \vdash \sigma \rightsquigarrow \Sigma'$ and $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$, which specify how a syntactic fragment σ extends the semantics Σ of a program context, described by Γ, to result in the semantics Σ'. The former sort of judgement applies when σ is a value expression, or a program (i.e. some number of module bindings followed by a value expression). The latter applies when σ is a module expression, module type expression, or the body of a structure or signature; in which case the judgement also derives a semantic description of σ.

Valid semantic judgements are defined inductively, in a 'big-step' style, by the rules in fig. 4, below. To determine the semantics Σ of a program P, we derive a valid judgement of the form $\Sigma_\bot; \Gamma_\bot \vdash P \rightsquigarrow \Sigma$, where $\Sigma_\bot$ denotes the *empty* semantics (i.e. the tuple consisting of the empty binding resolution and syntactic reification functions and empty value extension kernel), and $\Gamma_\bot$ denotes the empty environment (i.e. mapping every value identifier to the null location,

(Extended) Module Paths

$$\textsc{(ModId)}: \frac{}{\Sigma;\Gamma \vdash x \rightsquigarrow (\Gamma(x), \Sigma)} \quad \textsc{(PMod)}: \frac{\Sigma;\Gamma \vdash q \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash q\,.\,x \rightsquigarrow (\Delta, \Sigma')}\ (\exists \ell\,.\,\Sigma'_\rho(\ell) = x \wedge (\ell, \Delta) \in D) \quad \textsc{(PApp)}: \frac{\Sigma;\Gamma \vdash q_1 \rightsquigarrow ((\ell{:}\Delta_1){\to}\Delta_2, \Sigma'') \quad \Sigma'';\Gamma \vdash q_2 \rightsquigarrow (\Delta'_1, \Sigma')}{\Sigma;\Gamma \vdash q_1\ (q_2) \rightsquigarrow (\Delta_2, \Sigma'[\Delta_1 \otimes \Delta'_1])}$$

Value Expressions

$$\textsc{(ValId)}: \frac{}{\Sigma;\Gamma \vdash v_\ell \rightsquigarrow \Sigma[\ell \mapsto (v, \Gamma(v))]} \quad \textsc{(PVal}_1\textsc{)}: \frac{\Sigma;\Gamma \vdash p \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash p\,.\,v_\ell \rightsquigarrow \Sigma'[\ell \mapsto (v, \ell')]}\ (\Sigma'_\rho(\ell') = v \wedge \ell' \in D) \quad \textsc{(PVal}_2\textsc{)}: \frac{\Sigma;\Gamma \vdash p \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash p\,.\,v_\ell \rightsquigarrow \Sigma'[\ell \mapsto (v, \bot)]}\ (\forall \ell' \in D\,.\,\Sigma'_\rho(\ell') \neq v)$$

$$\textsc{(VLet)}: \frac{\Sigma;\Gamma \vdash e_1 \rightsquigarrow \Sigma'' \quad \Sigma''[\ell \mapsto v];\Gamma[v \mapsto \ell] \vdash e_2 \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \texttt{let}\ v_\ell\ =\ e_1\ \texttt{in}\ e_2 \rightsquigarrow \Sigma'} \quad \textsc{(VFun)}: \frac{\Sigma[\ell \mapsto v];\Gamma[v \mapsto \ell] \vdash e \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \texttt{fun}\ v_\ell\ \texttt{->}\ e \rightsquigarrow \Sigma'} \quad \textsc{(VApp)}: \frac{\Sigma;\Gamma \vdash e_1 \rightsquigarrow \Sigma'' \quad \Sigma'';\Gamma \vdash e_2 \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash e_1\ e_2 \rightsquigarrow \Sigma'}$$

Signature Bodies

$$\textsc{(Empty)}: \frac{}{\Sigma;\Gamma \vdash \varepsilon \rightsquigarrow (\emptyset, \Sigma)} \quad \textsc{(SigVal)}: \frac{\Sigma[\ell \mapsto v];\Gamma[v \mapsto \ell] \vdash S \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{val}\ v_\ell : \_;; S \rightsquigarrow (\{\ell\} \oplus_{\Sigma'} D, \Sigma'[\{\ell\} \otimes D])}$$

$$\textsc{(SigMod)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto x];\Gamma[x \mapsto \Delta] \vdash S \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{module}\ x_\ell : M;; S \rightsquigarrow (\{(\ell, \Delta)\} \oplus_{\Sigma'} D, \Sigma')} \quad \textsc{(SigInc)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (D, \Sigma'') \quad \Sigma'';\Gamma +_{\Sigma''_\rho} D \vdash S \rightsquigarrow (D', \Sigma')}{\Sigma;\Gamma \vdash \texttt{include}\ M;; S \rightsquigarrow (D \oplus_{\Sigma'} D', \Sigma'[\lfloor D \rfloor \otimes D'])}\ (D \text{ proper for } \Sigma''_\rho)$$

$$\textsc{(SigMty}_1\textsc{)}: \frac{\Sigma[\ell \mapsto t];\Gamma[t \mapsto \emptyset] \vdash S \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{module type}\ t_\ell;; S \rightsquigarrow (\{(\ell, \emptyset)\} \oplus_{\Sigma'} D, \Sigma')} \quad \textsc{(SigMty}_2\textsc{)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto t];\Gamma[t \mapsto \Delta] \vdash S \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{module type}\ t_\ell\ =\ M;; S \rightsquigarrow (\{(\ell, \Delta)\} \oplus_{\Sigma'} D, \Sigma')}$$

Module Types

$$\textsc{(MtyId)}: \frac{}{\Sigma;\Gamma \vdash t \rightsquigarrow (\Gamma(t), \Sigma)} \quad \textsc{(PMty)}: \frac{\Sigma;\Gamma \vdash p \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash p\,.\,t \rightsquigarrow (\Delta, \Sigma')}\ (\exists \ell\,.\,\Sigma'_\rho(\ell) = t \wedge (\ell, \Delta) \in D) \quad \textsc{(Sig)}: \frac{\Sigma;\Gamma \vdash S \rightsquigarrow (\Delta, \Sigma')}{\Sigma;\Gamma \vdash \texttt{sig}\ S\ \texttt{end} \rightsquigarrow (\Delta, \Sigma')}$$

$$\textsc{(MtyFun)}: \frac{\Sigma;\Gamma \vdash M_1 \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto x];\Gamma[x \mapsto \Delta] \vdash M_2 \rightsquigarrow (\Delta', \Sigma')}{\Sigma;\Gamma \vdash \texttt{functor}\ (x_\ell : M_1)\ \texttt{->}\ M_2 \rightsquigarrow ((\ell{:}\Delta){\to}\Delta', \Sigma')}$$

$$\textsc{(Constr)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \qquad \Sigma''[\ell \mapsto x];\Gamma \vdash q \rightsquigarrow (\Delta', \Sigma')}{\Sigma;\Gamma \vdash M\ \texttt{with module}\ x_\ell\ =\ q \rightsquigarrow (\Delta \blacktriangleleft_{\Sigma'} (x{:}\Delta'), \Sigma'[\Delta \otimes \{(\ell, \Delta')\}])} \quad \textsc{(Subst)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \qquad \Sigma''[\ell \mapsto x];\Gamma \vdash q \rightsquigarrow (\Delta', \Sigma')}{\Sigma;\Gamma \vdash M\ \texttt{with module}\ x_\ell\ :=\ q \rightsquigarrow (\Delta \setminus_{\Sigma'} x, \Sigma'[\Delta \otimes \{(\ell, \Delta')\}])}$$

Structure Bodies

$$\textsc{(StrVal)}: \frac{\Sigma;\Gamma \vdash e \rightsquigarrow \Sigma'' \quad \Sigma''[\ell \mapsto v];\Gamma[v \mapsto \ell] \vdash s \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{let}\ v_\ell\ =\ e;; s \rightsquigarrow (\{\ell\} \oplus_{\Sigma'} D, \Sigma'[\{\ell\} \otimes D])} \quad \textsc{(StrInc)}: \frac{\Sigma;\Gamma \vdash m \rightsquigarrow (D, \Sigma'') \quad \Sigma'';\Gamma +_{\Sigma''_\rho} D \vdash s \rightsquigarrow (D', \Sigma')}{\Sigma;\Gamma \vdash \texttt{include}\ m;; s \rightsquigarrow (D \oplus_{\Sigma'} D', \Sigma'[\lfloor D \rfloor \otimes D'])}\ (D \text{ proper for } \Sigma''_\rho)$$

$$\textsc{(StrMod)}: \frac{\Sigma;\Gamma \vdash m \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto x];\Gamma[x \mapsto \Delta] \vdash s \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{module}\ x_\ell\ =\ m;; s \rightsquigarrow (\{(\ell, \Delta)\} \oplus_{\Sigma'} D, \Sigma')} \quad \textsc{(StrMty)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto t];\Gamma[t \mapsto \Delta] \vdash s \rightsquigarrow (D, \Sigma')}{\Sigma;\Gamma \vdash \texttt{module type}\ t_\ell\ =\ M;; s \rightsquigarrow (\{(\ell, \Delta)\} \oplus_{\Sigma'} D, \Sigma')}$$

Module Expressions and Programs

$$\textsc{(Struct)}: \frac{\Sigma;\Gamma \vdash s \rightsquigarrow (\Delta, \Sigma')}{\Sigma;\Gamma \vdash \texttt{struct}\ s\ \texttt{end} \rightsquigarrow (\Delta, \Sigma')} \quad \textsc{(Annot)}: \frac{\Sigma;\Gamma \vdash m \rightsquigarrow (\Delta_1, \Sigma'') \quad \Sigma'';\Gamma \vdash M \rightsquigarrow (\Delta_2, \Sigma')}{\Sigma;\Gamma \vdash m : M \rightsquigarrow (\Delta_1 \blacktriangleright_{\Sigma'} \Delta_2, \Sigma'[\Delta_1 \otimes \Delta_2])} \quad \textsc{(MApp)}: \frac{\Sigma;\Gamma \vdash m_1 \rightsquigarrow ((\ell{:}\Delta_1){\to}\Delta_2, \Sigma'') \quad \Sigma'';\Gamma \vdash m_2 \rightsquigarrow (\Delta'_1, \Sigma')}{\Sigma;\Gamma \vdash m_1\ (m_2) \rightsquigarrow (\Delta_2, \Sigma'[\Delta_1 \otimes \Delta'_1])}$$

$$\textsc{(MFun)}: \frac{\Sigma;\Gamma \vdash M \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto x];\Gamma[x \mapsto \Delta] \vdash m \rightsquigarrow (\Delta', \Sigma')}{\Sigma;\Gamma \vdash \texttt{functor}\ (x_\ell : M)\ \texttt{->}\ m \rightsquigarrow ((\ell{:}\Delta){\to}\Delta', \Sigma')} \quad \textsc{(PMod)}: \frac{\Sigma;\Gamma \vdash m \rightsquigarrow (\Delta, \Sigma'') \quad \Sigma''[\ell \mapsto x];\Gamma[x \mapsto \Delta] \vdash P \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \texttt{module}\ x_\ell\ =\ m;; P \rightsquigarrow \Sigma'}$$

Figure 4. The abstract renaming semantics of the OCaml calculus.

and every module and module type identifier to the empty structural description, viz. the empty set).

We use some shorthand notation for specifying updates to $\Sigma = (\rightarrowtail, \mathbb{E}, \rho)$: (1) $\Sigma[\ell \mapsto \iota]$ stands for $(\rightarrowtail, \mathbb{E}, \rho[\ell \mapsto \iota])$; (2) $\Sigma[\ell \mapsto (\iota, \ell')]$ stands for $(\rightarrowtail[\ell \mapsto \ell'], \mathbb{E}, \rho[\ell \mapsto \iota])$; and (3) $\Sigma[\Delta_1 \otimes \Delta_2]$ stands for $(\rightarrowtail, \mathbb{E} \cup (\Delta_1 \otimes_\rho \Delta_2), \rho)$. To minimise notation, we also write: (4) $D \oplus_\Sigma D'$ for $D \oplus_\rho D'$; (5) $\Delta \blacktriangleright_\Sigma \Delta'$ for $\Delta \blacktriangleright_\rho \Delta'$; (6) $\Delta \blacktriangleleft_\Sigma (x{:}\Delta')$ for $\Delta \blacktriangleleft_\rho (x{:}\Delta')$; and (7) $\Delta \setminus_\Sigma x$ for $\Delta \setminus_\rho x$.

Figure 4 groups the rules according to the different kinds of syntax: value expressions, module types (including signature components), modules (including structure components), and module paths. We only give rules for extended module paths, as standard module paths are a strict subset of these.

The meaning of identifiers is given by looking them up in the semantic environment Γ, as specified in the rules (ValId), (ModId), and (MtyId). The rule (ValId), for value identifiers, also updates the binding resolution function. Thus, a reference v_ℓ (i.e. the occurrence of a value identifier v at location ℓ) resolves to the location $\Gamma(v)$ of the declaration of v currently in scope. When $\Gamma(v) = \bot$, this signifies no matching declaration is in scope and so the binding resolution function indicates that the reference is unresolved. Identifers qualified by module paths are handled by rules (PVal_1), (PVal_2), (PMod), and (PMty). The premises of these rules derive a description D of the module path, and the meaning is given by the component of D whose binding ℓ reifies to the identifier. When there is no such component, then qualified module and module type identifiers are meaningless (i.e. have no valid judgements), but in this case (PVal_2) indicates that a value identifier is an unresolved reference.

The derivation rules process syntactic elements left-to-right, bringing each declaration into the scope of the remaining program. For example the (PMod) rule derives the semantics for a program beginning with a module binding, `module` x_ℓ = m, in the context of a semantics Σ and a semantic environment Γ. The left-hand premise of the rule adds to Σ the abstract renaming information (i.e. binding structure, value extension, and syntactic reification information) from the module expression m to produce an updated semantics Σ'', as well as a semantic description Δ of m. The right-hand premise then derives the semantics Σ' of the remainder of the program, P. Note that it adds the module binding to the scope, by updating the semantic environment Γ to include a mapping of the module identifier x to the description Δ. It also adds a mapping $\ell \mapsto x$ for the module binding to the syntactic reification function of Σ''. Similarly, the (MFun) rule derives a semantic description of a functor parameter in the left-hand premise, and adds a corresponding mapping in the environment for the right-hand premise to derive the semantics of the functor body.

To derive the description of a module m annotated with a module type M, the (Annot) rule applies the modulation operation (cf. definition 12) to the descriptions Δ_1 and Δ_2 derived for m and M, respectively. This results in a modified version of Δ_1, in which bindings not also appearing in Δ_2 are removed and any new bindings from Δ_2 are added. The rules (Constr) and (Subst) give meanings to module types modified by module constraints and destructive module substitutions, respectively, using the selective modulation and filtering operations (cf. definitions 13 and 14).

The rules for signature bodies build descriptions by using the superposition operation (cf. definition 11) to combine the descriptions derived for the initial component and the remainder of the body. The rules for structure bodies work analogously. For example, the (SigInc) rule handles a signature body starting with `include` M;; by building the description $D \oplus D'$, where D and D' are the descriptions derived for the included module type M and the remainder of the signature body, S, respectively. The environment Γ (i.e. containing scope) for the remainder of body is updated with the description D of the included module type (cf. section 3.3). Thus the rule requires that the description D is proper. Using superposition means that if the remainder of the signature body S contains a redeclaration (i.e. a shadowing) of any binding in M, then the resulting description contains information only about the new declaration.

Notice that in the rules for signature and structure bodies, the value extension in the abstract semantics is also updated with the *join* of the derived descriptions (cf. definition 10). The result is that any shadowed declarations or bindings are related by the value extension. This is necessary to be able to construct valid renamings (cf. proposition 8 below).

Example 6 (Shadowing)**.** In the type of the module

```
module M : sig val foo : bool;; val foo : int;; end =
  struct let foo = 42 end
```

`val foo : int;;` shadows `val foo : bool;;`. To rename `foo` correctly, both declarations in the module type (as well as the binding `let foo = 42`) must be renamed, else the compiler will reject the resulting program as ill-typed.

Although the semantics allows us to preserve well-typedness during renaming (since it captures the information required for renaming to occur within module types), notice that the semantic rules do *not* perform any sort of type checking nor guarantee the well-typedness of programs. We consider this a feature rather than a bug since we see issues of renaming as orthogonal to type safety. Indeed, it is often desirable to be able to carry out renaming on incomplete (ill-typed) programs, and our semantics facilitates this.

The semantics also allows us to properly reason about renaming with respect to encapsulation, which is a key feature of the use of module types annotations.

Example 7 (Encapsulation)**.** In the following modules

```
module A = struct let foo = ... ;; let bar = ... ;; end
module B = struct
  include (A : sig val foo : _ end);; let bar = ... ;;
end
```

the `include` of module `A` in `B` is restricted by a module type. This serves to hide the fact that `A` contains a binding of `bar`. Thus, the binding of `bar` given in module `B` does not introduce any shadowing and so we can rename `A.bar` and `B.bar` independently.

Under certain conditions (which we elide, but elaborate in the appendix), the semantics of fig. 4 are deterministic. That is, for a given syntactic fragment σ, Semantics Σ and environment Γ, there is at most one description Δ and semantics Σ' such that $\Sigma; \Gamma \vdash \sigma \rightsquigarrow \Sigma'$ or $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ is valid. Thus, assuming the conditions that imply determinism, the rules compute a (partial) semantic *function*. As such, they allow us to interpret programs.

Definition 15 (Semantics of programs)**.** We define families of (partial) interpretation functions $[\![\sigma]\!]_{\Sigma;\Gamma}$ and $\mathcal{D}_{\Sigma;\Gamma}(\sigma)$, indexed by pairs of semantics Σ and environments Γ, that return (when they exist) the unique Σ' and Δ, respectively, such that $\Sigma;\Gamma \vdash \sigma \rightsquigarrow \Sigma'$ or $\Sigma;\Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ is valid.

We write $[\![\sigma]\!]$ to mean $[\![\sigma]\!]_{\Sigma_\perp;\Gamma_\perp}$. For a program P with $[\![P]\!] = \Sigma$, we will write $\rightarrowtail_P$, $\mathbb{E}_P$, and ρ_P to mean $\Sigma_{\rightarrowtail}$, $\Sigma_{\mathbb{E}}$, and Σ_ρ respectively.

The semantics naturally captures the syntactic information in a program pertaining to value identifiers.

Proposition 2. *If $[\![P]\!]$ is defined then* $\mathsf{ref}(P) = \mathit{dom}(\rightarrowtail_P)$ *and* $\mathsf{decl}(P) = \mathit{dom}_{\mathcal{V}}(\rho_P) \setminus \mathit{dom}(\rightarrowtail_P)$.

4 Characterising Renaming

The primary purpose of our semantics is to distinguish 'correct' renamings from 'incorrect' ones. For example, given some declaration ℓ in program P and a new identifier v, it might seem that $P' = P[\ell' \mapsto v \mid \ell' = \ell \vee \ell' \rightarrowtail_P \ell]$ would be a good candidate for forming a minimal, valid renaming. That is, rename the identifier at location ℓ to v, as well as the identifiers at all the locations ℓ' that resolve to ℓ. As discussed in section 1 this is not always sufficient, and in general we find that we should modify multiple declarations and their associated references.

The first step, therefore, is to specify which renamings preserve meaning as captured by our semantics. The meaning that we are interested in is *name-invariant* binding structure, which we capture at the semantic level via the following equivalence relations.

Definition 16 (Semantic Equivalence)**.** We define the following equivalences on semantics and environments:

- $\Sigma \sim \Sigma'$ iff $\Sigma_{\rightarrowtail} = \Sigma'_{\rightarrowtail}$, $\Sigma_{\mathbb{E}} = \Sigma'_{\mathbb{E}}$, $\mathsf{dom}(\Sigma_\rho) = \mathsf{dom}(\Sigma'_\rho)$, $\Sigma_\rho(\ell) \in \mathcal{V} \Leftrightarrow \Sigma'_\rho(\ell) \in \mathcal{V}$, and if $\Sigma_\rho(\ell) \notin \mathcal{V}$ then $\Sigma_\rho(\ell) = \Sigma'_\rho(\ell)$.
- $\Gamma \sim \Gamma'$ iff $\Gamma_{\mathcal{M}} = \Gamma'_{\mathcal{M}}$, and $\mathsf{ran}(\Gamma_{\mathcal{V}}) = \mathsf{ran}(\Gamma'_{\mathcal{V}})$.

When $\Sigma \sim \Sigma'$ and $\Gamma \sim \Gamma'$ hold, we write $(\Sigma, \Gamma) \sim (\Sigma', \Gamma')$.

Intuitively, this equivalence relation captures when two pairs of semantics and environments represent program contexts having the same binding structure regardless of the particular value identifiers that have been used. Notice that the equivalence relation on semantics comprises the same conditions on the syntactic reification function as are used to define renamings. With this equivalence we define what it means for a renaming to be valid.

Definition 17 (Valid Renamings)**.** We say that a renaming $\sigma \hookrightarrow \sigma'$ is *valid with respect to* $\Sigma;\Gamma$, and write $\Sigma;\Gamma \vdash \sigma \hookrightarrow \sigma'$, when $[\![\sigma]\!]_{\Sigma;\Gamma}$ is defined, and there exists a semantics Σ' and environment Γ' with $(\Sigma', \Gamma') \sim (\Sigma, \Gamma)$ such that $[\![\sigma']\!]_{\Sigma';\Gamma'}$ is defined and $[\![\sigma]\!]_{\Sigma;\Gamma} \sim [\![\sigma']\!]_{\Sigma';\Gamma'}$. When $\Sigma_\perp;\Gamma_\perp \vdash \sigma \hookrightarrow \sigma'$ holds, then we simply say that the renaming $\sigma \hookrightarrow \sigma'$ is *valid*.

For whole programs, validity of renamings collapses to the following statement.

Proposition 3. *$P \hookrightarrow P'$ is valid iff $[\![P]\!]$ and $[\![P']\!]$ are defined and $[\![P]\!] \sim [\![P']\!]$.*

Thus, to check whether a renaming is valid, it suffices to compute the semantics of the original and renamed programs and then check that they are equivalent. We now proceed to explore some of the properties of valid renamings. That is to say, we begin to outline a theory of renaming for our OCaml calculus.

Firstly, as a basic sanity check, we note that renamings induce an equivalence relation on programs.

Proposition 4 (Equivalences)**.** *The following properties hold:*

i) $P \hookrightarrow P$ is a (valid) renaming (when $[\![P]\!]$ defined).
ii) If $P \hookrightarrow P'$ is a (valid) renaming, then so is $P' \hookrightarrow P$.
iii) If $P \hookrightarrow P'$ and $P' \hookrightarrow P''$ are (valid) renamings, then so is $P \hookrightarrow P''$.

A main objective for defining the semantics is to characterise renamings semantically. The following property shows that (up to unresolved references) a renaming is described by its dependencies and the binding resolution function.

Proposition 5. *Suppose $P \hookrightarrow P'$ is a valid renaming, and let $L = \{\ell \mid \ell \in \delta(P, P') \vee \exists \ell' \in \delta(P, P').\, \ell \rightarrowtail_P \ell'\}$; then $L \subseteq \varphi(P, P')$ and $\ell \rightarrowtail_P \perp$ for all $\ell \in \varphi(P, P') \setminus L$.*

This also means checking whether a renaming is invalid is cheaper than checking its validity, since we need only compute the semantics of the original program. Furthermore, the dependencies of a renaming are themselves characterised by the extension kernel.

Proposition 6. *If $P \hookrightarrow P'$ is valid, then $\delta(P, P')$ has a partitioning that is a subset of $\mathcal{L}_{/\hat{\mathbb{E}}_P}$.*

The value extension kernel thus captures the dependencies inherent in a renaming: for a program P, all declarations belonging to an $\hat{\mathbb{E}}_P$-equivalence class must be renamed together (along with their associated references), or none at all. In other words, *dependencies are value extensions*. This provides an alternative check for invalidity of renamings.

Given a declaration in a semantically meaningful program, it then follows from propositions 5 and 6 that we can uniquely identify a lower bound for the footprint of any valid renaming containing the given declaration.

Proposition 7. *If $P \overset{\ell}{\hookrightarrow} P'$ is valid and $\ell \in \mathsf{decl}(P)$, then $\varphi(P, P') \supseteq \{\ell' \mid \ell' \in [\ell]_{\hat{\mathbb{E}}_P} \vee \exists \ell'' \in [\ell]_{\hat{\mathbb{E}}_P}.\, \ell' \rightarrowtail_P \ell''\}$.*

This is, in fact, a tight bound since we can construct a valid renaming with exactly this footprint.

Proposition 8. *Suppose $[\![P]\!]$ is defined, $\ell \in \mathsf{decl}(P)$, and $v \in \mathcal{V}$ does not occur in P, then $P \hookrightarrow P'$ is a valid renaming, where $P' = P[\ell' \mapsto v \mid \ell' \in [\ell]_{\hat{\mathbb{E}}_P} \vee \exists \ell'' \in [\ell]_{\hat{\mathbb{E}}_P}.\, \ell' \rightarrowtail_P \ell'']$.*

Moreover, when a valid renaming does not have a minimal footprint it is always possible to decompose it into two strictly smaller valid renamings, provided the renaming involves a fresh identifier.

Proposition 9 (Factorisation). *Suppose $P \hookrightarrow P'$ is valid, and let ℓ and ℓ' be two distinct declarations in $\delta(P, P')$ such that $(\ell, \ell') \notin \hat{\mathbb{E}}_P$ and $\rho_{P'}(\ell)$ does not occur in P; then there exists P'' such that both $P \hookrightarrow P''$ and $P'' \hookrightarrow P'$ are valid, with $\varphi(P, P'') \subset \varphi(P, P')$ and $\varphi(P'', P') \subset \varphi(P, P')$.*

5 Adequacy of the Semantics

The renaming semantics defined in section 3 leads to an intuitive theory for characterising renaming. However, it is also important that it constitutes a sensible abstraction of what we understand programs really to be. That is, the abstract semantics should be *adequate*, in the sense that it is a sound abstraction of the behavioural meaning of programs. We now show that our renaming semantics is indeed adequate in this sense, by proving that if two renaming-related programs have equivalent abstract semantics then they have the same behaviour.

The model of program behaviour we consider is a denotational semantics that extends the model considered by Leroy in [21]. Our extensions cover the additional features of the module system incorporated by our OCaml calculus (i.e. **include** statements, module types as members of structures and signatures, and **module with** constraints on module types). However, we depart from that model in another important way: our model gives a denotational meaning to module types, which contribute towards the meaning of programs. This is because, as discussed in section 3 above, module types have meaning in the context of renaming. In contrast, the model of [21] simply ignores all module types in programs. For lack of space, we only describe the essential differences of our denotational model compared with [21]. Full definitions can be found in the appendix.

We assume an interpretation, using standard results, of value expressions (viz. lambda terms) in some domain $\mathbb{F}$ containing an element **wrong** denoting run-time errors. We interpret modules in a domain $\mathbb{M}$ satisfying:

$$\mathbb{M} = \mathbb{D} + (\mathbb{M} \to \mathbb{M}) + \mathbf{wrong}$$
$$\mathbb{D} = (\mathcal{V} \rightharpoonup_{\mathsf{fin}} \mathbb{F}) \times (\mathcal{T} \rightharpoonup_{\mathsf{fin}} \mathbb{T}) \times (\mathcal{M} \rightharpoonup_{\mathsf{fin}} \mathbb{M})$$

where $\mathbb{T}$ is the set in which we interpret module types, defined inductively as the set X satisfying the following:

$$X = D + (\mathcal{M} \times X) \times X + \mathbf{wrong}$$
$$D = \wp_{\mathsf{fin}}(\mathcal{V}) \times (\mathcal{T} \rightharpoonup_{\mathsf{fin}} X) \times (\mathcal{M} \rightharpoonup_{\mathsf{fin}} X)$$

The denotational semantics of programs is given by a function $(\!|\cdot|\!)_\theta$, which interprets syntactic elements in their appropriate domains. As usual, it is parameterised by a denotational environment θ mapping identifiers to elements of the appropriate domain.

The interpretation of module types mirrors the way descriptions of module types are constructed by our abstract semantics. The main difference, then, between our denotational semantics and that of [21] is that module type denotations affect the meaning of modules. This happens in two ways. Firstly, the denotation of a module is modified by the denotation of a module type with which it is annotated.

$$(\!|m : M|\!)_\theta = \text{let } d = (\!|m|\!)_\theta \text{ in let } \tau = (\!|M|\!)_\theta \text{ in } d : \tau$$

Here, we utilise a semantic operation $d : \tau$ on denotations d and τ, which essentially inserts 'dynamic' type checks. For example, if d denotes a structure containing some binding of v but τ denotes a signature not containing a declaration of v, then v will not be in the domain of $d : \tau$. In the reverse situation, v will be in the domain of $d : \tau$, but it will return **wrong** on being applied to v. This is analogous to the approach taken in gradual typing frameworks [39, 40], which insert casts that perform such dynamic checks.

Secondly, this operation is used to insert checks on the argument to a functor according to the module type declared for the corresponding parameter.

$$(\!|\texttt{functor } (x : M) \texttt{ -> } m|\!)_\theta = \text{let } \tau = (\!|M|\!)_\theta \text{ in } \lambda d.(\!|m|\!)_{\theta[x \mapsto d:\tau]}$$

We note that, for well-typed programs, this approach should be equivalent to the one ignoring all type annotations. Notwithstanding, by considering a 'dynamically typed' model we do not have to separately consider well-typedness.

Our abstract renaming semantics is sound with respect to the denotational semantics defined above. We write $(\!|P|\!)$ to mean $(\!|P|\!)_{\theta_\perp}$, where $\theta_\perp$ is the environment that maps everything to **wrong**.

Proposition 10 (Adequacy). $(\!|P|\!) = (\!|P'|\!)$ *if* $P \hookrightarrow P'$ *is valid.*

The converse result, completeness, does not hold. That is, there are renamings that preserve the operational meaning of programs, but which result in different abstract semantics. This is due to the fact that, according to our semantics, valid renamings must preserve the shadowing structure.

Example 8. Consider the following variation of example 6.

```
module M = (struct let foo = true let foo = 42 end
  : sig val foo : bool val foo : int end) ;;
M.foo ;;
```

Here there is shadowing in both the module expression and the module type. According to our semantics, the only valid renaming is the one that renames all instances of the identifier `foo`. However, it would be sufficient (in the sense that the result is denotationally equivalent) to rename both instances in the module type, but only the latter one in the module expression. It seems plausible that our semantics could be refined in order to reason about those cases in which (un)shadowing is allowed to occur, thus facilitating a completeness result. We leave this for future work.

6 Rotor: A Refactoring Tool for OCaml

We have built a prototype refactoring tool for the OCaml language, called Rotor (Reliable OCaml Tool for OCaml Refactoring), that carries out renaming based on the analysis modelled in our abstract semantics. The source code and a pre-compiled executable are available online [5, 6].

6.1 Implementation

The aim of our implementation was to produce a tool embodying proposition 8 above. That is, given a particular declaration in the input source code, the tool should produce a patch consisting of the minimal number of changes needed to correctly enact the renaming. In handling the OCaml language as a whole, we faced a number of challenges.

– In order to avoid having to build basic language processing functionality from scratch, we implemented Rotor in OCaml itself. This allowed us to reuse the compiler as a library, providing an abstract representation of the input source code directly. OCaml's abstract syntax data type contains source code location information, which we used to produce accurate patches describing how to apply the renaming. We also relied on the recently developed `visitors` library [35] to automatically generate boilerplate code for traversing and processing the abstract syntax trees. This library provides similar functionality to that found in Haskell's SYB [18] and Strafunski [19] libraries, or the Stratego/XT framework [10].

– For complex, real-world codebases the wider ecosystem and build pipeline of OCaml becomes relevant, as it introduces extra layers not present in the basic language itself. Two aspects of this were particularly relevant in implementing Rotor. Firstly, OCaml has a preprocessor infrastructure called `PPX` [13]. This means that, in general, the abstract syntax that is processed by Rotor may contain elements that do not correspond to actual source code. Moreover it is not always straightforward to determine when this is and is not the case, and our analysis must work on the post-processed code in order to fully compute the information it needs. Secondly, some build systems (e.g. `dune` [4]), in order to implement packaging and namespace separation, utilise custom mappings between the names of source files and the names of compiled modules, cf. [22, §8.12]. Rotor must be aware of these custom mappings to be able to produce accurate patch information.

– The primary difficulty in implementing our analysis was computing the binding resolution and dependency information on which our analysis is built. Since it was not feasible to reimplement an entire binding analysis for the full language, we again relied on the OCaml compiler as much as possible. During type inference the compiler performs a binding analysis, assigning each binding a unique stamp. However, it only computes a partial view of the binding resolution function of our analysis. For value identifiers qualified by a module path (i.e. that refer to a binding inside another module), the compiler only provides the stamp of the outermost containing module whereas our binding resolution function provides the 'stamp' of the value binding itself.

For this reason, Rotor approximates the abstract locations of our semantics using these logical paths. In fact, we had to extend the notion of paths implemented by the compiler, since they cannot refer to subcomponents of module types, or those of functors and their parameters. For each reference in the program, Rotor can rely on information provided by the compiler to determine which logical path it resolves to. For each path, Rotor must then compute the other paths it depends upon, i.e. which other declarations are in its value extension. It does this by comparing path prefixes whenever it encounters an `include` statement, module type annotation, module type constraint, or functor application. For example if, in analysing the dependencies of the path `M.N.foo` (representing the `foo` value binding in the `N` submodule of module `M`), Rotor encounters the module binding `module P = M : T`, it would generate dependencies on the paths `P.N.foo` and `T.N.foo`. An important point here is that, in our semantics, the logical paths `M.N.foo` and `P.N.foo` would denote the same (abstract) location, since module `P` is bound to module `M`. However, according to the information we can extract from the compiler, references might resolve to either of the paths. Thus, Rotor must treat them as (logically) distinct dependencies.

Rotor computes dependency information using a worklist algorithm, beginning with a working set containing just the path of the declaration to be renamed. For each dependency, it analyses the codebase to compute which other paths it depends upon, adding ones it has not previously processed to the working set. As each dependency is processed, Rotor also identifies all of its references and builds up the final patch that can be applied to enact the renaming. At each point in the analysis, Rotor checks to ensure that the new name does not introduce shadowing, or modify any shadowing that already occurs. If this is the case, Rotor fails with a warning to the user. The renaming might also fail if Rotor detects a declaration must be renamed that is not part of the input source code (e.g. a library function).

6.2 Rotor in Practice

The aim of Rotor is to provide a practical tool for refactoring "real world" OCaml code, but in doing this we have made a number of tradeoffs between the cost of handling certain features and the benefits that that would bring. We chose not to support modules that use PPX, because this can give rise to function declarations being automatically generated during PPX preprocessing; extending Rotor to handle these cases would be very hard, as we would need to enable it to reason about meta-programming. Other aspects include module type extraction, which lies outside of core OCaml; our choice

here has been to concentrate on a set of language features that cover all essential aspects of the module system, such that other aspects could be treated using similar techniques.

We evaluated Rotor on two substantial, real-world codebases. Firstly, Jane Street's standard library overlay [16], comprising 869 source files in 77 libraries. Secondly, part of the OCaml (4.04.0) compiler itself [3] consisting of 502 source files. We analysed each codebase to extract its set of value bindings, which we used as test cases. For each case, we asked Rotor to rename the binding to a fresh name not occurring in the codebase and tested the result by attempting to re-compile.

Setting aside the cases that we do not handle, and the cases which fail because they generate a requirement to rename an (external) library function, at the point of writing around 70% of the tests pass; of the remainder, some are doubtlessly due to bugs, but others are due to the presence of features of the language so far unhandled by the system.

Data for the successful test cases are given in table 1. This comprises the number of source files requiring changes and the number of hunks in the diff patch produced by Rotor. These measures constitute a good proxy for the theoretical notion of 'footprint' we have considered in our formalism. We also show the number of renaming dependencies. For each metric, we give maximum, mean, and mode values. Our evaluation shows that while renamings usually require only a small number of changes (both commonly, as well as on average), they can be surprisingly complex. The largest footprint for the successful Jane Street test cases consists of 128 changes in 50 individual files. For the OCaml compiler the largest footprint is 59 changes across 19 files. The metrics for the Jane Street testbed have significantly higher values than for the OCaml compiler, showing that the former codebase is more complex. Indeed, an examination of the source code shows much heavier use of module types, module type constraints, and functors in the Jane Street codebase.

As well as providing test data, this exercise has demonstrated the value of the dependency concept in practice. Among the test cases for the OCaml compiler, more than thirty generate sets of dependencies of size at least 10, and over a hundred have non-trivial sets of dependencies. For the Jane Street testbed, over eight hundred cases generate 10 or more dependencies; over a thousand have more than 1.

Table 1. Results of experimental evaluation.

	Jane Street			OCaml Compiler		
	Max	Mean	Mode	Max	Mean	Mode
Files	50	5.0	3	19	3.8	3
Hunks	128	7.5	3	59	5.9	3
Dependencies	1127	24.0	19	35	1.6	1

7 Related Work

A general survey of refactoring research up until 2004 has been given by Mens and Tourwé [30]. Much work on refactoring has been carried out within the object-oriented programming paradigm; a standard reference is [12]. Thompson and Li have carried out a survey of refactoring tools for functional languages [42] including the tools Wrangler [23, 24] (for Erlang [8]) and HaRe [25] (for Haskell [34]). Renaming, and perhaps refactoring generally, seems to be more difficult in a language like OCaml with its powerful module system. Erlang is dynamically typed, but has a flat module system, and Haskell, whilst possessing a powerful multi-feature type system, also does not support complex modules. Object-oriented features overlap somewhat with those of OCaml's module system, since the use of inheritance and interfaces can lead to analogous dependencies across a program. Schäfer et al. use an inversion of attribute-grammar lookup rules to help identify entities within object-oriented programs [38]. This handles classes and interfaces, but not the full complexity needed for the OCaml module system (e.g. functor applications, `include` statements, and module type inference); it also requires an existing attribute-based formalism, which does not exist for OCaml.

It has long been recognised that, for correctness, refactorings generally require certain preconditions to hold [14]. As we have already noted, the notion of dependency that we describe in this paper is something other than a precondition and seems not to have been studied before. Our approach of constructing a semantic abstraction specifically to support refactoring in a general purpose programming language is also novel, as far as we know. Whiteside et al. have considered a similar approach for refactoring formal proofs scripts [44] including, in particular, renaming lemmas. However, in this setting, lemma names are global free identifiers and so renaming is simply a matter of replacing uses of the name, which are readily identified. Our semantic abstraction also bears some similarity to work on program analysis via fact extraction. This is the approach behind the CodeQuest tool [15] and, more recently, the QL language [9] and Semmle platform [1]. The JunGL tool [43] uses this technique in the context of refactoring to check preconditions. However, these tools do not consider this technique as a semantic abstraction in a formal sense as we do. Lin and Holt consider an abstract formalization of fact extraction [27], and consider different notions of semantic completeness [28], but this is not tied to any language in particular and cannot obviously be applied to refactoring. Separately, Lin has also devised a (relational) algebraic procedure for binding resolution in various (imperative) languages, based on fact extraction [26]. Related to this is the recent work on scope graphs for name resolution [32] and static type checking [7]. This is a generic framework for specifying (and checking) static semantics of languages (including binding resolution), but it does not contain anything that supports or directly corresponds to our notion of value extension. Poulsen et al. show that scope graphs represent an abstraction of a generic memory model

based on frames, and thus allow interpreters to be derived from scope graphs [36]. However it is not shown that scope graphs directly abstract any existing operational models, as our semantics does. Menarini et al. take a semantic approach to code review, but do not address how semantics may guide automatic construction of refactorings [29].

We have formally shown our renaming semantics to be an abstraction of an operational model of our OCaml calculus, which is an extension of the model considered in [20, 21] by Leroy. Rossberg et al. have also given a semantics for a large subset of OCaml and its module system via a translation to System F_ω [37]. However, since this translation requires programs to be well-typed, we did not follow this approach. The CakeML project [17] is a compiler stack for a large subset of OCaml that is formalised and fully verified in the HOL4 theorem prover [41]. However, it currently contains only the most basic form of the module system.

8 Future Work

One direction for future work is to extend our calculus and abstract semantics to cover the extended features of OCaml's module system, such as first class and recursive modules, module type extraction, and type-level module aliases. The first three should only require straightforward extensions of the approach we describe in this paper. Modelling type-level aliases is more challenging, as they interact non-trivially with module type constraints [2]. In particular, it would involve updating semantic descriptions in a non-local manner.

We would like to further extend our approach in order to rename identifiers within OCaml's other namespaces. These include value types, modules, module types, record fields, object methods, and data constructors and polymorphic variants. Again, we anticipate that this should be largely straightforward. Module and module type identifiers behave in a similar way to value identifiers. The relationship between object methods and object types is analogous to that between value bindings and module types, and so could be handled by the semantic structures we have already defined. The case of value type identifiers is even simpler since type definitions cannot shadow each other in the same module. The more difficult cases are those of (polymorphic) data constructors and record field identifiers, because they can be overloaded and are resolved by means of a type-based analysis.

As mentioned in the introduction, the notion of renaming we have focussed on is not the only one that might be sensibly applied. Our motivation was to effect renaming by applying only the simplest of syntactic transformations, whilst ensuring the operational meaning of programs was preserved. This necessitated an unrestricted, whole-program, scope for changes. However the scope of changes can be limited by allowing more complex syntactic transformations. For example, we can limit changes to within a given module by transforming references to that module into an 'adapter' module which client code can treat as if it were the original version. This would preserve the operational meaning of programs and could be supported by extending our semantics to: (1) include binding information for module identifiers; and (2) restricting the value extension kernel to relate declarations only within the specified module. Alternatively we could extend the semantics to keep track of a containment relation between value declarations and module bindings. Support for renaming of methods in object-oriented programs in, e.g., Eclipse or Visual Studio allows users to simply restrict the scope of renaming but without introducing any mitigation outside of this scope. Although generally unsound, this is another approach we could support and would require simply restricting construction of the value extension kernel.

We note that our current notion of value extension kernel can already directly support other kinds of refactorings. For example, generalising a function to accept an additional argument requires identifying the value extension that it belongs to, since we would also need to generalise the other functions in the extension. Furthermore, although our renaming theory currently only utilises the equivalence relations induced by value extension kernels, it is interesting to consider whether there is useful information in the particular structure of the kernel relation itself. One possibility is to define a complexity measure for programs based on the 'distance' of the value extension kernel from its equivalence closure.

Lastly, our prototype tool, Rotor, needs further development. It is our hope that it can become an industrially useful tool to the OCaml community. Furthermore, we would like to investigate whether our approach can be integrated into a mechanised formal framework, such as CakeML [17].

9 Conclusion

In this paper we have presented a framework based on an abstract denotational semantics that allows us to reason about the correctness of renaming value bindings within OCaml modules. We have formally modelled a significant subset of the OCaml core language and its module system. Our abstract semantics allows us to characterise renamings which do not change the operational meaning of programs, and describe how they compose. A key concept that arose from our analysis was that of the *extension* of a value binding, this being the collection of bindings in the program that are related via the name-aware structures of the language. To the best of our knowledge, this is a novel concept not previously identified in the literature. We implemented our framework in a prototype tool called Rotor, which is able to automatically carry out renaming on real-world OCaml code with a significant degree of success.

Acknowledgments

This work was supported by EPSRC grant no. EP/N028759/1.

References

[1] [n. d.]. Semmle™. https://www.semmle.com/ (accessed 11th November 2018).

[2] 2012. OCaml Bug Report 5514: "with module" semantics seem broken. https://github.com/ocaml/ocaml/issues/5514 Last accessed 22nd March 2018. Communicated to us by Leo White.

[3] 2016. The Core OCaml System: Compilers, Runtime System, Base Libraries (version 4.04.0). https://github.com/ocaml/ocaml/tree/4.04.0

[4] 2018. Dune: A Composable Build System. https://github.com/ocaml/dune

[5] 2019. A Prototype Refactoring Tool for OCaml. https://gitlab.com/trustworthy-refactoring/refactorer/

[6] 2019. A Prototype Refactoring Tool for OCaml (Docker Image). https://hub.docker.com/r/reubenrowe/ocaml-rotor/

[7] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes As Types. *PACMPL* 2, OOPSLA (2018), 114:1–114:30. https://doi.org/10.1145/3276484

[8] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG* (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK.

[9] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72 (2008), 52–70. Issue 1–2. https://doi.org/10.1016/j.scico.2007.11.003

[11] The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. https://doi.org/10.5281/zenodo.1219885

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Alain Frisch. 2014. PPX and Extension Points. https://lexifi.com/blog/ppx-and-extension-points (blog post).

[14] William G. Griswold and William F. Opdyke. 2015. The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research. *IEEE Software* 32, 6 (2015), 30–38. https://doi.org/10.1109/MS.2015.107

[15] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. 2005. CodeQuest: Querying Source Code with Datalog. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, New York, NY, USA, 102–103. https://doi.org/10.1145/1094855.1094884

[16] Jane Street. 2018. Standard Library Overlay. https://github.com/janestreet/core

[17] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/2535838.2535841

[18] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179

[19] Ralf Lämmel and Joost Visser. 2003. A Strafunski Application Letter. In *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*. Springer-Verlag, Heidelberg Berlin, Germany, 357–375. https://doi.org/10.1007/3-540-36388-2_24

[20] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17–21, 1994*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

[21] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/199448.199476

[22] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. The OCaml System Release 4.07 Documentation and User's Manual. http://caml.inria.fr/pub/docs/manual-ocaml/

[23] Huiqing Li and Simon J. Thompson. 2012. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*. 501–515. https://doi.org/10.1007/978-3-642-28872-2_34

[24] Huiqing Li, Simon J. Thompson, George Orösz, and Melinda Tóth. 2008. Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*. 61–72. https://doi.org/10.1145/1411273.1411283

[25] Huiqing Li, Simon J. Thompson, and Claus Reinke. 2005. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.* 141, 4 (2005), 29–34. https://doi.org/10.1016/j.entcs.2005.02.053

[26] Yuan Lin. 2008. *Completeness of Fact Extractors and a New Approach to Extraction with Emphasis on the Refers-to Relation*. Ph.D. Dissertation. http://hdl.handle.net/10012/3865

[27] Yuan Lin and Richard C. Holt. 2004. Formalizing Fact Extraction. *Electr. Notes Theor. Comput. Sci.* 94 (2004), 93–102. https://doi.org/10.1016/j.entcs.2004.01.001

[28] Yuan Lin, Richard C. Holt, and Andrew J. Malton. 2003. Completeness of a Fact Extractor. In *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*. 196–205. https://doi.org/10.1109/WCRE.2003.1287250

[29] Massimiliano Menarini, Yan Yan, and William G. Griswold. 2017. Semantics-assisted Code Review: An Efficient Toolchain and a User Study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 – November 03, 2017*. IEEE Computer Society, 554–565. https://doi.org/10.1109/ASE.2017.8115666

[30] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30 (2004), 126–139. Issue 2. https://doi.org/10.1109/TSE.2004.1265817

[31] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, Sebastopol, CA, USA.

[32] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[33] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[34] Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries: Revised Report*. Cambridge University Press, Cambridge, UK. https://haskell.org/onlinereport

[35] François Pottier. 2017. Visitors Unchained. *PACMPL* 1, ICFP (2017), 28:1–28:28. https://doi.org/10.1145/3110272

[36] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 20:1–20:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.20

[37] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing Modules. *J. Funct. Program.* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

[38] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and Extensible Renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 277–294. https://doi.org/10.1145/1449764.1449787

[39] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming 2006 - Proceedings of the 2006 Workshop on Scheme and Functional Programming, Portland, Oregon, Sunday September 17, 2006*, Robert Bruce Findler (Ed.). University of Chicago, 1100 East 58th Street, Chicago, IL 60637, 81–92. https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2006-06 Technical Report TR-2006-06.

[40] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[41] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

[42] Simon Thompson and Huiqing Li. 2013. Refactoring Tools for Functional Languages. *Journal of Functional Programming* 23, 3 (2013), 293–350. https://doi.org/10.1017/S0956796813000117

[43] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. 2006. JunGL: A Scripting Language for Refactoring. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/1134311

[44] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. 2011. Towards Formal Proof Script Refactoring. In *Intelligent Computer Mathematics*, James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe (Eds.). Springer, Berlin/Heidelberg, Germany, 260–275. https://doi.org/10.1007/978-3-642-22673-1_18