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

Anonymous Author(s)

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.

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

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 [10]. 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 [21, 30].

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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 following example.

```
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))) ;;
```

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.

(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 [32]. 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 some of the renaming theory in the Coq proof assistant [38]. This is included as supplementary material with our submission. Results which have not yet been proved are marked as conjectures. We have also included as supplementary material 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 describe the subset of OCaml that we study, and formally define operations that carry out 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 and section 8 concludes.

2 An OCaml Module Calculus

The subset of OCaml for which we build our formal theory is defined in fig. 1. It extends the calculus considered in [19, 20] 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

Module Paths $p ::= x \mid p.x$

Extended Module Paths $q ::= x \mid q.x \mid q(q)$

Value Expressions $e ::= v \mid p.v \mid \texttt{let } v \texttt{ = } e \texttt{ in } e \mid \texttt{fun } v \texttt{ -> } e \mid e\ e$

Programs $P ::= e \mid \texttt{module } x \texttt{ = } m \texttt{ ;; } P$

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

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

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** $d ::= \texttt{let } v \texttt{ = } e \mid \texttt{module } x \texttt{ = } m \mid \texttt{module type } t \texttt{ = } M \mid \texttt{include } m$

Figure 1. Syntax of a core calculus for OCaml with modules.

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. The first three should only require straightforward extensions of the approach we describe in this paper. Modelling type-level aliases correctly is more challenging, as they interact non-trivially with module type constraints [2].

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 ι 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 $\perp \in \mathcal{L}$ 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) $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$; (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: $\varphi(\sigma, \sigma') = \{\ell \mid \ell \in \mathrm{dom}(\sigma) \wedge \sigma(\ell) \neq \sigma'(\ell)\}$. 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 $\mathrm{decl}(\sigma)$ of (value) *declarations* in a program (fragment) σ is the set of all locations $\ell \in \mathrm{dom}(\sigma)$ for which there exists $\ell' \in \mathrm{dom}(\sigma)$ such that either: $\sigma(\ell') = \texttt{val } v_\ell : \_ \texttt{;;}$, $\sigma(\ell') = \texttt{let } v_\ell \texttt{ = } e \texttt{;;}$, $\sigma(\ell') = \texttt{let } v_\ell \texttt{ = } e \texttt{ in } e'\texttt{;;}$, or $\sigma(\ell') = \texttt{fun } v_\ell \texttt{ -> } e\texttt{;;}$.

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

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 $\perp$). 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. A syntactic reification function $\rho : \mathcal{L} \rightharpoonup \mathcal{I}$ is a partial mapping from locations to identifiers (and we assume that ρ does not map the null location $\perp$). 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.

We call such collections of declarations the *extension*[1] 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. A value extension kernel $\mathbb{E}$ is any binary relation on locations. $\hat{\mathbb{E}}$ denotes the reflexive, symmetric and transitive closure of $\mathbb{E}$.

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.

The notion of value extension will allow us to carry out renaming correctly by capturing the high-level, global structures present in a program. This is expressed in conjecture 2 below.

[1]This is by analogy with Frege's development and use of this term within the Logicist philosophical programme.

```
module type Stringable = sig
  val to_string : _
end
module Pair(X : Stringable)(Y : Stringable) = struct
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end
module Int = struct
  let to_string i = int_to_string i
end
module String = struct
  let to_string s = s
end
module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
```

Figure 2. Graphical representation of the semantics.

To give an intuition as to how these elements are used, we show in fig. 2 a visual representation of the binding resolution function and value extension kernel that would be derived for the example in section 1. The binding resolution mappings are depicted using dashed arrows, and pairs in the value extension kernel by solid arrows.

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, ranged over by Δ, are objects defined inductively as follows:

- A *component* is either: (i) a location ℓ; or (ii) a pair of the form (ℓ, Δ).
- A semantic description is either: (i) a set of components, in which case we call it *structural*; or (ii) a tuple of the form $((\ell, \Delta), \Delta')$, in which case we call it *functorial*.

We write $\mathcal{D}$ for the set of all semantic descriptions. Additionally, we use D to range over structural descriptions only and will usually write functorial descriptions $((\ell, \Delta), \Delta')$ as $(\ell{:}\Delta){\to}\Delta'$. We write $\lfloor D \rfloor$ to denote the set $\{\ell \mid \ell \in D\}$.

Basic components, comprising of simply a location, capture the locations of instances of identifiers bound to values. Components of the form (ℓ, Δ) represent subcomponents with further structure (i.e. sub-modules and sub-module types), along with the location of the instance of the identifier that they are bound to. 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 type; the right-hand member of the pair, Δ', describes the body.

We now define some operations on semantic descriptions that will be used to define the semantics of programs. In the following definitions, when we write $\rho(\ell)$ for a reification function ρ and a location ℓ, we mean this to also assert that ρ is defined on ℓ.

Superposition. We define a family of superposition operations $\oplus_\rho$ on structural descriptions, parameterised by syntactic reification functions, that selectively combine the elements of the two descriptions based on syntactic information about locations contained in the reification function. The purpose of this is to sequentially combine the semantic description of two structure or signature fragments. In particular, it is used to model the effect of **`include`** statements.

Definition 10 (Description Superposition)**.** The superposition operation $\oplus_\rho$ on structural descriptions is defined by:

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

For example, 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 $D_{\mathrm{A}} = \{1, 2\}$, while the remainder of the body of module **B** after the **`include`** statement consists of the set $D_{\mathrm{body}} = \{3\}$. To form a description of the module **B**, we can superpose D_{A} and D_{body} with respect to the obvious reification function ρ that maps location 1 to `foo`, and locations 2 and 3 to `bar`. That is $D_{\mathrm{B}} = D_{\mathrm{A}} \oplus_\rho D_{\mathrm{body}} = \{1, 3\}$. Here, the location 3 from D_{body} is chosen over 2 from D_{A} since ρ maps them both to the same identifier.

Join. We define a family of join operations $\otimes_\rho$ on semantic descriptions, parameterised by syntactic reification functions, that each produce a value extension kernel from their input descriptions. The purpose of this operation is to extract the information about value extensions that is induced by the association of a module with a module type, either through an explicit module type annotation $(m : M)$ or via a functor application $(m_1(m_2))$.

Definition 11 (Description Join)**.** The description join operation $\otimes_\rho$ is a binary operation on descriptions producing a value extension relation and is defined inductively as follows:

$$\begin{aligned} D_1 \otimes_\rho D_2 = &\{(\ell_1, \ell_2) \mid \ell_1 \in D_1 \land \ell_2 \in D_2 \land \rho(\ell_1) = \rho(\ell_2)\} \\ &\cup \{(\ell_1, \ell_2) \mid \exists (\ell, \Delta_1) \in D_1, (\ell', \Delta_2) \in D_2. \\ &\qquad \rho(\ell) = \rho(\ell') \land (\ell_1, \ell_2) \in \Delta_1 \otimes_\rho \Delta_2\} \end{aligned}$$

$$(\ell_1{:}\Delta_1){\to}\Delta_1' \otimes_\rho (\ell_2{:}\Delta_2){\to}\Delta_2' = (\Delta_1 \otimes_\rho \Delta_2) \cup (\Delta_1' \otimes_\rho \Delta_2')$$
$$\Delta \otimes_\rho \Delta' = \emptyset \qquad \text{otherwise}$$

To illustrate this, consider the functor from the example in section 1.

```
module type Stringable = sig val to_string1 : _ ;; end
module Pair = functor (X2 : Stringable) ->
  functor (Y3 : Stringable) ->
    struct let to_string4 = fun ...;; end
```

The **Stringable** module type is described by $D_{\mathtt{Stringable}} = \{1\}$. For the **Pair** functor, our semantics constructs the description $D_{\mathtt{Pair}} = (2{:}D_{\mathtt{Stringable}}){\to}((3{:}D_{\mathtt{Stringable}}){\to}\{5\})$. Applications of functors induce dependencies between the declarations in the type of the parameter, and corresponding bindings in the module used as the argument. These dependecies are computed by the join operation. Thus, for modules

```
module Int = struct let to_string5 = fun ...;; end
module String = struct let to_string6 = fun ...;; end
```

with descriptions $D_{\mathtt{Int}} = \{5\}$ and $D_{\mathtt{String}} = \{6\}$, respectively, an application `Pair(String)(Int)` induces dependencies $D_{\mathtt{Stringable}} \otimes_\rho D_{\mathtt{String}} = \{(1, 6)\}$ and $D_{\mathtt{Stringable}} \otimes_\rho D_{\mathtt{Int}} = \{(1, 5)\}$. Here, again, ρ is the obvious reification function (mapping 1 to `to_string`, 2 to `X`, etc.).

Modulation. We define a family of operations $\blacktriangleright_\rho$ on semantics descriptions, also parameterised by a syntactic reification function, that produce another semantic description. This operation will be used to model how the description of a module is updated by a module type annotation.

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

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

For example, consider the following module type, which is a weakening of the type of the **Pair** functor considered above.

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

$D_{\mathtt{Weak}} = (9{:}D_{\mathtt{Stringable2}}){\to}((10{:}D_{\mathtt{Stringable2}}){\to}\emptyset)$ describes the module type **WeakPair**, where $D_{\mathtt{Stringable2}} = \{7, 8\}$. To describe the module `M = Pair : WeakPair`, we use the result of the applying the modulation operation.

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

Notice that the result type has been restricted, but the types of the functor parameters in the original $D_{\mathtt{Pair}}$ description have been augmented by the additional `from_string` declarations (location 8) in the types of the parameters in $D_{\mathtt{Weak}}$. Here, we intend that ρ has been updated with new mappings reflecting the identifiers occurring in **Stringable2** and **WeakPair** above.

We also define a family of selective modulation operations that modulate only certain elements of a structural description. This will be used to model the effect of a module constraint on a module type

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 inductively as follows:

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

For example, suppose we have the following module type.

```
module type Set = sig
  module Elt11 : Stringable ;;
  val empty12 : _ ;;
end
```

Consider also the following module.

```
Int2 = struct include Int;; let from_string13 = ...;; end
```

These can be described by $D_{\mathtt{Set}} = \{12, (11, D_{\mathtt{Stringable}})\}$ and $D_{\mathtt{Int2}} = \{5, 13\}$. To compute the description of the module type given by `IntSet = Set with module Elt = Int2` we use selective modulation:

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

Filtering. Lastly, we define an operation that removes elements from a structural description corresponding to a particular name, according to a given syntactic reification function. This will be used to model the effect of a destructive module substitution on a module type.

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

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

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

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. $\Gamma_\bot$ will denote the environment consisting of the functions that map every value identifier to the null location, and every module and module type identifier to the empty structural description (i.e. the empty set).

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

We will interpret programs as tuples $(\rightarrowtail, \mathbb{E}, \rho)$ comprising a binding resolution function, a value extension kernel, and a syntactic reification function. We will 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 the semantics Σ.

To give the semantics of programs, we define two sorts of judgement, $\Sigma; \Gamma \vdash \sigma \rightsquigarrow \Sigma'$ and $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$, which specify how the 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.

The semantic judgements are defined inductively using the rules in fig. 3 below. They employ the following shorthand notation for specifying updates to $\Sigma = (\rightarrowtail, \mathbb{E}, \rho)$.

- $\Sigma[\ell \mapsto \iota]$ stands for $(\rightarrowtail, \mathbb{E}, \rho[\ell \mapsto \iota])$.
- $\Sigma[\ell \mapsto (\iota, \ell')]$ stands for $(\rightarrowtail[\ell \mapsto \ell'], \mathbb{E}, \rho[\ell \mapsto \iota])$.
- $\Sigma[\Delta_1 \otimes \Delta_2]$ stands for $(\rightarrowtail, \mathbb{E} \cup (\Delta_1 \otimes_\rho \Delta_2), \rho)$.
- $D \oplus_\Sigma D'$ stands for $D \oplus_\rho D'$.
- $\Delta \blacktriangleright_\Sigma \Delta'$ stands for $\Delta \blacktriangleright_\rho \Delta'$, and $\Delta \blacktriangleleft_\Sigma (x{:}\Delta')$ stands for $\Delta \blacktriangleleft_\rho (x{:}\Delta')$.
- $\Delta \setminus_\Sigma x$ stands for $\Delta \setminus_\rho x$.

Figure 3 elides the rules for standard module paths, since extended module paths are a strict superset of these. Moreover, for standard module paths, the judgement $\Sigma; \Gamma \vdash p \rightsquigarrow \Delta$ is used as a shorthand since, as can be straightforwardly determined, standard module paths do not update the semantics (although extended module paths, i.e. containing functor applications, do update the value extension kernel). We denote by $\Sigma_\bot$ the *empty* semantics, i.e. the tuple consisting of the empty binding resolution function and syntactic reification functions and empty value extension kernel.

Under certain conditions (which we elide, but elaborate in the appendix), the semantics of fig. 3 are deterministic. Thus 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 $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ holds. We write $[\![\sigma]\!]$ to mean $[\![\sigma]\!]_{\Sigma_\bot;\Gamma_\bot}$.

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* $\mathsf{decl}(P) = \mathit{dom}_{\mathcal{V}}(\rho_P) \setminus \mathit{dom}(\rightarrowtail_P)$.

An important point to note is that, even assuming an untyped value language, our semantics does *not* 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. On the other hand, we can preserve well-typedness during renaming since the semantics captures the information required for renaming to also occur within module types. This also allows us to properly reason about renaming with respect to encapsulation, as illustrated by the following example.

```
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, which 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. The result is that we can rename **A**.`bar` and **B**.`bar` independently, whereas otherwise

(Extended) Module Paths

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

Value Expressions

$$\frac{}{\Sigma;\Gamma\vdash v_\ell \leadsto \Sigma[\ell\mapsto(v,\Gamma(v))]} \qquad \frac{\Sigma;\Gamma\vdash p\leadsto D}{\Sigma;\Gamma\vdash p\,.\,v_\ell \leadsto \Sigma[\ell\mapsto(v,\ell')]}\,(\Sigma_\rho(\ell')=v\wedge\ell'\in D) \qquad \frac{\Sigma;\Gamma\vdash p\leadsto D}{\Sigma;\Gamma\vdash p\,.\,v_\ell\leadsto\Sigma[\ell\mapsto(v,\bot)]}\,(\forall\ell'\in D.\,\Sigma_\rho(\ell')\neq v)$$

$$\frac{\Sigma;\Gamma\vdash e_1\leadsto\Sigma'' \quad \Sigma''[\ell\mapsto v];\Gamma[v\mapsto\ell]\vdash e_2\leadsto\Sigma'}{\Sigma;\Gamma\vdash \texttt{let}\ v_\ell\ =\ e_1\ \texttt{in}\ e_2\leadsto\Sigma'} \qquad \frac{\Sigma[\ell\mapsto v];\Gamma[v\mapsto\ell]\vdash e\leadsto\Sigma'}{\Sigma;\Gamma\vdash\texttt{fun}\ v_\ell\ \texttt{->}\ e\leadsto\Sigma'} \qquad \frac{\Sigma;\Gamma\vdash e_1\leadsto\Sigma'' \quad \Sigma'';\Gamma\vdash e_2\leadsto\Sigma'}{\Sigma;\Gamma\vdash e_1\,e_2\leadsto\Sigma'}$$

Signature Bodies

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

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

$$\frac{\Sigma[\ell\mapsto t];\Gamma[t\mapsto\emptyset]\vdash S\leadsto(D,\Sigma')}{\Sigma;\Gamma\vdash\texttt{module type}\ t_\ell\,;;\,S\leadsto(\{(\ell,\emptyset)\}\oplus_{\Sigma'}D,\Sigma')} \qquad \frac{\Sigma;\Gamma\vdash M\leadsto(\Delta,\Sigma'') \quad \Sigma''[\ell\mapsto t];\Gamma[t\mapsto\Delta]\vdash S\leadsto(D,\Sigma')}{\Sigma;\Gamma\vdash\texttt{module type}\ t_\ell\ =\ M\,;;\,S\leadsto(\{(\ell,\Delta)\}\oplus_{\Sigma'}D,\Sigma')}$$

Module Types

$$\frac{}{\Sigma;\Gamma\vdash t\leadsto(\Gamma(t),\Sigma)} \qquad \frac{\Sigma;\Gamma\vdash p\leadsto D}{\Sigma;\Gamma\vdash p\,.\,t\leadsto(\Delta,\Sigma)}\begin{pmatrix}\exists\ell.\,\Sigma_\rho(\ell)=t\\ \wedge\,(\ell,\Delta)\in D\end{pmatrix} \qquad \frac{\Sigma;\Gamma\vdash S\leadsto(\Delta,\Sigma')}{\Sigma;\Gamma\vdash\texttt{sig}\ S\ \texttt{end}\leadsto(\Delta,\Sigma')}$$

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

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

Structure Bodies

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

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

Module Expressions and Programs

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

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

Figure 3. The abstract renaming semantics of the OCaml calculus.

we would consider the latter to shadow the former and thus have to rename both together to preserve binding structure. A key feature of (module) types is that they should express such encapsulation properties.

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 $\Sigma_{\rightarrowtail} = \Sigma'_{\rightarrowtail}$, $\Sigma_{\mathbb{E}} = \Sigma'_{\mathbb{E}}$, $\mathrm{dom}(\Sigma_\rho) = \mathrm{dom}(\Sigma'_\rho)$, $\Sigma_\rho(\ell) \in \mathcal{V} \Leftrightarrow \Sigma'_\rho(\ell) \in \mathcal{V}$, 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 $\mathrm{ran}(\Gamma_{\mathcal{V}}) = \mathrm{ran}(\Gamma'_{\mathcal{V}})$.

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.

Conjecture 1. *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.

Conjecture 2. *Let $P \hookrightarrow P'$ be a valid renaming, 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 conjectures 1 and 2 that we can uniquely identify a lower bound for the footprint of any valid renaming containing the given declaration.

Conjecture 3. *For $P \overset{\ell}{\hookrightarrow} P'$ a valid renaming and $\ell \in \mathrm{decl}(P)$, $\varphi(P, P') \supseteq \{\ell' \mid \ell' \in [\ell]_{\hat{\mathbb{E}}_P} \vee \exists \ell'' \in [\ell]_{\hat{\mathbb{E}}_P}. \ell' \rightarrowtail_P \ell''\}$.*

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

Proposition 5. *Suppose $[\![P]\!]$ is defined, $\ell \in \mathrm{decl}(P)$, 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 possible to decompose it into two, strictly smaller valid renamings.

Conjecture 4 (Factorisation)**.** *Suppose $P \hookrightarrow P'$ is a valid renaming, and let ℓ and ℓ' be two distinct locations such that $\ell \in \varphi(P, P')$ and $\ell' \in \varphi(P, P')$, with $(\ell, \ell') \notin \hat{\mathbb{E}}_P$; then there exists a P'' such that both $P \hookrightarrow P''$ and $P'' \hookrightarrow P'$ are valid, with $\varphi(P, P'') \subset \varphi(P, P')$ and $\varphi(P'', P') \subset \varphi(P, P')$.*

The reader may notice that our theory of renaming only utilises the equivalence relations induced by value extension kernels, rather than making any direct use of the structure of the value extension kernel itself. Nevertheless, we propose that our renaming theory could potentially make use of this detailed structure. One possibility is to define a complexity measure based on the 'distance' of the value extension kernel from its equivalence closure. We leave such investigations to future work.

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 [20]. 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 [20] simply ignores all module types in programs. For lack of space, we only describe the essential differences of our denotational model compared with [20]. The appendix contains the full definitions.

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 $\llparenthesis \cdot \rrparenthesis_\theta$, 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 [20] 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.

$$\llparenthesis m : M \rrparenthesis_\theta = \text{let } d = \llparenthesis m \rrparenthesis_\theta \text{ in let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in } d : \tau$$

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 [36, 37], 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.

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

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 $\llparenthesis P \rrparenthesis$ to mean $\llparenthesis P \rrparenthesis_{\theta_\perp}$, where $\theta_\perp$ is the environment that maps everything to **wrong**.

Proposition 6 (Adequacy). *$\llparenthesis P \rrparenthesis = \llparenthesis P' \rrparenthesis$ 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 all shadowing that occurs in programs. For example, consider the following contrived but nonetheless valid OCaml program.

```
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 5 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 [34] 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 [17] and Strafunski [18] libraries, or the Stratego/XT framework [9].

– 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` [11]. 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. [21, §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 – which lie outside core OCaml – include module type extraction; 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 [15], 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 more than 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.

As well as providing test data, this exercise has demonstrated the value of the dependency concept in practice. Among the refactorings for the OCaml compiler, more than thirty generate sets of dependencies of size at least 24, and over a hundred have non-trivial sets of dependencies. These more complex refactorings typically span multiple files, and generate multiple patches. In summary for the compiler, in the successful cases we have these data.

	Max	Mean	Mode
Files	19	3.8	3
Hunks	59	5.9	3
Dependencies	35	1.6	1
Avg. Hunks/File	15.0	1.5	1.0

7 Related Work

A general survey of refactoring research up until 2004 has been given by Mens and Tourwé [29]. Much work on refactoring has been carried out within the object-oriented programming paradigm; a standard reference is [10]. Thompson and Li have carried out a survey of refactoring tools for functional languages [39] including the tools Wrangler [22, 23] (for Erlang [7]) and HaRe [24] (for Haskell [33]). 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.

It has long been recognised that, for correctness, refactorings generally require certain preconditions to hold [12]. 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 for the purpose of refactoring, as far as we know, is also novel. It bears some similarity to work on program analysis via fact extraction. This is the approach behind the codeQuest tool [13] and, more recently, the QL language [8] and Semmle platform [1]. The JunGL tool [41] 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 [26], and consider different notions of semantic completeness [27], 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 [25]. Related to this is the recent work on scope graphs for name resolution [31] and static type checking [40]. This is a generic framework for specifying (and checking) static semantics of languages (including binding resolution), but does not present scope graphs as abstractions of operational models. Menarini et al. take a semantic approach to code review, but do not address how semantics may guide automatic construction of refactorings [28].

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 [19, 20] 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 [35]. However, since this translation requires programs to be well-typed, we did not follow this approach. The CakeML project [16] is a compiler stack for a large subset of OCaml that is formalised and fully verified in the HOL4 theorem prover [14]. However, it currently contains only the most basic form of the module system.

8 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.

Future Work. We would like to extend our approach to cover other features of OCaml's module system, such as first class and recursive modules, module type extraction, and type-level module aliases. We would also like to consider renaming module and module type bindings, as well as other kinds of refactoring. It will be interesting to see if our notion of value extension is flexible enough to capture other language features and more complex refactorings. 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.

References

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

[2] 2012. Mantis Bug Report 5514: "with module" semantics seem broken. https://caml.inria.fr/mantis/view.php?id=5514 Last accessed 9th Nov. 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 Compsable Build System. https://github.com/ocaml/dune

[5] 2018. A Prototype Refactoring Tool for OCaml. link removed for double blind review.

[6] 2018. A Prototype Refactoring Tool for OCaml (Docker Image). link removed for double blind review.

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

[8] 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

[9] 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

[10] 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.

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

[12] 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

[13] 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

[14] HOL4 2017. Interactive Theorem Prover. hol-theorem-prover.org

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

[16] 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

[17] 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

[18] 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

[19] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of POPL'94: 21^{st} 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

[20] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Conference Record of POPL'95: 22^{nd} 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

[21] 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/

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

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

[31] 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

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

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

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

[35] 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

[36] 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.

[37] 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

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

[39] 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

[40] 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

[41] 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

A Proofs

Here we elaborate on the results stated in the main body of the paper, and provide proofs of those results that are not included in the Coq formalisation.

A.1 The Abstract Renaming Semantics

It was stated in section 3 that, under certain conditions, the semantics are deterministic. Here, we give the formal statement of this property.

We first have to define a notion of well-behavedness for semantic descriptions and environments. Given an interpretation of locations as identifiers (i.e. a syntactic reification function), a semantic description is well-behaved when each location in a (possibly nested) structural description corresponds to an identifier that is *unique* within that description.

Definition 18 (Well-behaved Descriptions)**.** We define the subset of semantic descriptions that are *well-behaved* with respect to a given syntactic reification function ρ as the smallest set satisfying the following.

- A structural description D is well-behaved w.r.t. ρ when:
  (i) $\ell \in D$ implies $\ell \in \mathrm{dom}(\rho)$ and $\rho(\ell) \in \mathcal{V}$;
  (ii) $(\ell, \Delta) \in D$ implies $\ell \in \mathrm{dom}(\rho)$, $\rho(\ell) \in \mathcal{M} \cup \mathcal{T}$ and Δ is well-behaved w.r.t. ρ; and
  (iii) if $\rho(\ell) = \rho(\ell')$ for $\ell, \ell' \in D$ or $(\ell, \Delta), (\ell', \Delta') \in D$, then also $\ell = \ell'$.
- A functorial description $(\ell{:}\Delta) \rightarrow \Delta'$ is well-behaved w.r.t. ρ when both Δ and Δ' are well-behaved w.r.t. ρ.

That is, a semantic description that is well-behaved for ρ is proper for ρ 'all the way down'.

We say that an environment Γ is well-behaved for a syntactic reification function ρ when $\Gamma(v) = \ell$ implies $\rho(\ell) = v$ for every $\ell \neq \bot$, and each Δ_ι such that $\Gamma(\iota) = \Delta_\iota$ ($\iota \in \mathcal{M} \cup \mathcal{T}$) is well-behaved w.r.t. ρ. We say that an environment Γ or semantic description Δ is well-behaved for a *semantics* Σ when it is well-behaved w.r.t. the reification function ρ for which $\rho(\iota) = \ell$ if and only if $\Sigma_\rho(\iota) = \ell$ and $\ell \notin \mathrm{dom}(\Sigma_\rightarrowtail)$. We denoted by $\mathrm{ran}_{\mathcal{I}}(\Gamma)$ the set $\mathrm{ran}(\Gamma_{\mathcal{V}}) \cup \{\ell \mid \exists \Delta. (\ell, \Delta) \in \mathrm{ran}(\Gamma_{\mathcal{M}})\}$.

Lemma 5 (Determinism)**.** *For any program fragment σ, semantics Σ, and environment Γ that is well-behaved for Σ and satisfies $(dom(\Sigma_\rho) \cup ran_{\mathcal{I}}(\Gamma)) \cap dom(\sigma) = \emptyset$, there is at most one Σ' and one Δ such that $\Sigma; \Gamma \vdash \sigma \rightsquigarrow \Sigma'$ or $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$.*

Proof. Given in the Coq formalisation. By induction on the definition of the semantics. In fact, we need to use a stronger hypothesis involving the following additional invariants:
1. Σ' contains only locations in $\mathrm{dom}(\Sigma_\rho)$ and $\mathrm{dom}(\sigma)$;
2. Γ is well-behaved also for Σ';
3. for judgements $\Sigma; \Gamma \vdash \sigma : (\Delta, \Sigma')$, then Δ is well-behaved for Σ'; and
4. Δ well-behaved w.r.t. Σ implies Δ well-behaved w.r.t. Σ', for all Δ. □

Thus, we specify that $[\![\sigma]\!]_{\Sigma;\Gamma}$ and $\mathcal{D}_{\Sigma;\Gamma}(\sigma)$ are only defined when Γ is well-behaved for Σ and $(\mathrm{dom}(\Sigma_\rho) \cup \mathrm{ran}_{\mathcal{I}}(\Gamma)) \cap \mathrm{dom}(\sigma) = \emptyset$. A consequence of lemma 5 is that (when defined) $\mathcal{D}_{\Sigma;\Gamma}(\sigma)$ is well-behaved w.r.t. ρ, where $[\![\sigma]\!]_{\Sigma;\Gamma} = (\rightarrowtail, \mathbb{E}, \rho)$.

The following property is necessary for a semantics to correspond to an actual program fragment.

Definition 19 (Properness)**.** A semantics $\Sigma = (\rightarrowtail, \mathbb{E}, \rho)$ is called *proper* when it satisfies the following conditions.
(i) $\mathrm{dom}(\rightarrowtail) \cap \mathrm{ran}(\rightarrowtail) = \emptyset$.
(ii) $\ell \rightarrowtail \ell'$ and $\ell' \neq \bot$ implies $\rho(\ell) = \rho(\ell')$.
(iii) $\rho(\ell) \in \mathcal{V}$, for all $l \in \mathrm{dom}(\rightarrowtail) \cup \mathrm{ran}(\rightarrowtail)$ with $\ell \neq \bot$.
(iv) $\rho(\ell) = \rho(\ell') \in \mathcal{V}$, $\ell \notin \mathrm{dom}(\rightarrowtail)$ and $\ell' \notin \mathrm{dom}(\rightarrowtail)$, for all (ℓ, ℓ') in $\mathbb{E}$.

Note that the empty semantics is trivially proper. We can show that properness is preserved by the semantics.

Lemma 6. *Let Σ be proper, and environment Γ be well-behaved for Σ; if $\Sigma; \Gamma \vdash \sigma \rightsquigarrow \Sigma'$ or $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ holds then Σ' is proper.*

Proof. By induction on the semantic rules. Given in the Coq formalisation. □

The semantic characterisation of the syntactically defined references and declarations given in proposition 2 is a special case of the following lemma. We write $\mathrm{decl}(\Sigma)$ to denote the set $\mathrm{dom}_{\mathcal{V}}(\Sigma_\rho) \setminus \mathrm{dom}(\Sigma_\rightarrowtail)$.

Proposition 7. *If $[\![\sigma]\!]_{\Sigma;\Gamma} = \Sigma'$ then:*
(i) $\mathrm{ref}(\sigma) = dom(\Sigma'_\rightarrowtail) \setminus dom(\Sigma_\rightarrowtail)$.
(ii) $\mathrm{decl}(\sigma) = \mathrm{decl}(\Sigma') \setminus \mathrm{decl}(\Sigma)$.

Proof. By induction on the semantic rules. Given in the Coq formalisation. □

We now justify the statement of validity for whole program renamings.

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

Proof. Notice that trivially $\Gamma_\bot$ is well-behaved for $\Sigma_\bot$ and, when restricting to pairs (Σ, Γ) such that Γ is well-behaved for Σ, we have $[(\Sigma_\bot, \Gamma_\bot)]_\sim = \{(\Sigma_\bot, \Gamma_\bot)\}$, whence the statement follows directly from definition 17. □

We now consider some properties pertaining to the structure of the semantics and descriptions synthesised by the semantic rules. In an abuse of notation, we will write $\mathcal{L}(\Delta)$ to denote the set of all locations appearing in (a subcomponent) of Δ. For an environment Γ and identifier $\iota \in \mathcal{M} \cup \mathcal{T}$, we then write $\Gamma_{\mathcal{D}}(\iota)$ for the description Δ such that there exists ℓ with $\Gamma(\iota) = (\ell, \Delta)$, and $\mathrm{ran}_{\mathcal{D}}(\Gamma)$ for the set $\bigcup_{\iota \in \mathcal{M} \cup \mathcal{T}} \mathcal{L}(\Gamma_{\mathcal{D}}(\iota))$.

Lemma 7. *If $\Sigma; \Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ then $\mathcal{L}(\Delta) \subseteq dom(\sigma) \cup ran_{\mathcal{D}}(\Gamma)$.*

Proof. By induction on the semantic rules. Included in the Coq formalisation. □

Lemma 8. *If $\Sigma;\Gamma \vdash \sigma \rightsquigarrow (\Delta, \Sigma')$ then $\mathbb{E}' \setminus \mathbb{E} \subseteq L \times L$, for $L = dom(\sigma) \cup ran_{\mathcal{D}}(\Gamma)$, where $\mathbb{E}$ and $\mathbb{E}'$ are the extension kernels of Σ and Σ', respectively.*

Proof. By induction on the semantic rules. Included in the Coq formalisation. □

The following concepts of inclusion, relevance and freshness for semantics are central to proving many of the results in this paper. To express these properties, we use the following notation for partial functions f and g, and (binary) relation R:

- $f \subseteq g$ denotes that, for all $x \in \mathrm{dom}(f)$, if $f(x) = y$ then $g(x) = y$;
- $f \setminus g$ denotes the function defined by $(f \setminus g)(x) = y$ if and only if $f(x) = y$ and either $g(x)$ undefined or $g(x) \neq y$;
- $f \subseteq R$ denotes that $f(x) = y$ only if $(x, y) \in R$.

The use of set-theoretic notation here is justified by the view of (partial) functions as sets of mappings (i.e. pairs). Notice that the following property holds.

Proposition 8. *Suppose that $f(x) = y$ but $x \notin dom(f \setminus g)$, then $g(x) = y$.*

Proof. Suppose, for contradiction, that in fact $g(x)$ is undefined or else $g(x) \neq y$. But then from the assumption that $f(x) = y$ we have, by definition, that $(f \setminus g)(x) = y$, which contradicts the assumption that $x \notin \mathrm{dom}(f \setminus g)$. □

The definitions of inclusion, relevance and freshness are as follows.

Definition 20 (Inclusion)**.** We say that Σ' *includes* Σ, and write $\Sigma \subseteq \Sigma'$, when the following hold: (1) $\Sigma_{\rightarrowtail} \subseteq \Sigma'_{\rightarrowtail}$; (2) $\Sigma_{\mathbb{E}} \subseteq \Sigma'_{\mathbb{E}}$; and (3) $\Sigma_{\rho} \subseteq \Sigma'_{\rho}$. When we additionally have $\ell \in \mathrm{dom}(\Sigma_\rho) \setminus \mathrm{dom}(\Sigma_{\rightarrowtail})$ implies $\ell \in \mathrm{dom}(\Sigma'_\rho) \setminus \mathrm{dom}(\Sigma'_{\rightarrowtail})$ for all locations $\ell \in \mathcal{L}$, we say that Σ' *properly* includes Σ.

Definition 21 (Relevance)**.** For semantics Σ and Σ', and a set of locations $L \subseteq \mathcal{L}$, we say Σ' is *relevant for L over* Σ, and write $\Sigma' \setminus \Sigma \subseteq L$, when the following hold:

(1) $\Sigma'_{\rightarrowtail} \setminus \Sigma_{\rightarrowtail} \subseteq L \times (L \cup \mathrm{dom}(\Sigma_\rho))$
(2) $\Sigma'_\rho \setminus \Sigma_\rho \subseteq L \times \mathcal{I}$
(3) $\Sigma'_{\mathbb{E}} \setminus \Sigma_{\mathbb{E}} \subseteq (L \cup \mathrm{dom}(\Sigma_\rho))^2 \setminus \mathrm{dom}(\Sigma_\rho)^2$

Definition 22 (Freshness)**.** We say that a set $L \subseteq \mathcal{L}$ of locations is *fresh* for a semantics $\Sigma = (\rightarrowtail, \mathbb{E}, \rho)$ when the following properties hold for all locations $\ell \in \mathcal{L}$:

(1) $\ell \in \mathrm{dom}(\rightarrowtail) \cup \mathrm{ran}(\rightarrowtail) \Rightarrow \ell \notin L$
(2) $(\exists \ell'. (\ell, \ell') \in \mathbb{E} \vee (\ell', \ell) \in \mathbb{E}) \Rightarrow \ell \notin L$
(3) $\ell \in \mathrm{dom}(\rho) \Rightarrow \ell \notin L$

For proper semantics, these properties are guaranteed by the interpretation function.

Lemma 9. *If $[\![\sigma]\!]_{\Sigma;\Gamma} = \Sigma'$ with Σ proper then: (1) Σ' properly includes Σ; (2) Σ' is relevant for $dom(\sigma)$ over Σ; and (3) $dom(\sigma)$ is fresh for Σ.*

Proof. Given in the Coq formalisation. The freshness property follows from properness and the preconditions for $[\![\sigma]\!]_{\Sigma;\Gamma}$ to be defined (cf. definition 15)—namely that $\mathrm{dom}(\Sigma_\rho) \cap \mathrm{dom}(\sigma) = \emptyset$. The other properties are shown by induction on syntactic structure. □

Thus, the major utility of definitions 20 to 22 lies in the following result.

Lemma 10. *Take semantics Σ_1, Σ_2, Σ'_1 and Σ'_2, with a set of locations $L \subseteq \mathcal{L}$ such that the following conditions hold:*

- *$\Sigma_1 \subseteq \Sigma'_1$, and $\Sigma_2 \subseteq \Sigma'_2$;*
- *$\Sigma'_1 \setminus \Sigma_1 \subseteq L$ and $\Sigma'_2 \setminus \Sigma_2 \subseteq L$; and*
- *L is fresh for both Σ_1 and Σ_2.*

Then $\Sigma'_1 \sim \Sigma'_2$ implies that $\Sigma_1 \sim \Sigma_2$.

Proof. Let $\Sigma_1 = (\rightarrowtail_1, \mathbb{E}_1, \rho_1)$, $\Sigma_2 = (\rightarrowtail_2, \mathbb{E}_2, \rho_2)$, with $\Sigma'_1 = (\rightarrowtail'_1, \mathbb{E}'_1, \rho'_1)$, and $\Sigma'_2 = (\rightarrowtail'_2, \mathbb{E}'_2, \rho'_2)$. Since $\Sigma'_1 \sim \Sigma'_2$, we have by definition 16 that $\rightarrowtail'_1 = \rightarrowtail'_2$, $\mathbb{E}'_1 = \mathbb{E}'_2$, $\mathrm{dom}(\rho'_1) = \mathrm{dom}(\rho'_2)$, $\rho'_1(\ell) \in \mathcal{V} \Leftrightarrow \rho'_2(\ell) \in \mathcal{V}$, and $\rho'_1(\ell) = \rho'_2(\ell)$ if $\rho'_1(\ell) \notin \mathcal{V}$. We must show the following:

($\rightarrowtail_1 = \rightarrowtail_2$): To see that $\rightarrowtail_1 \subseteq \rightarrowtail_2$, take $(\ell, \ell') \in \rightarrowtail_1$. Since $\Sigma_1 \subseteq \Sigma'_1$ it follows that $\rightarrowtail_1 \subseteq \rightarrowtail'_1$, and thus that $(\ell, \ell') \in \rightarrowtail'_1$. Moreover, since $\rightarrowtail'_1 = \rightarrowtail'_2$ it then follows that $(\ell, \ell') \in \rightarrowtail'_2$. Now, since L is fresh for Σ_1, we have that $\ell \notin L$ and therefore, since Σ'_2 is relevant for L over Σ_2, it follows that $\ell \notin \mathrm{dom}(\rightarrowtail'_2 \setminus \rightarrowtail_2)$. However, since we have that $(\ell, \ell') \in \rightarrowtail'_2$, by proposition 8 it must be that $(\ell, \ell') \in \rightarrowtail_2$ as required. A symmetric chain of reasoning shows that $\rightarrowtail_2 \subseteq \rightarrowtail_1$, hence we conclude.

($\mathbb{E}_1 = \mathbb{E}_2$): To see that $\mathbb{E}_1 \subseteq \mathbb{E}_2$, take $(\ell, \ell') \in \mathbb{E}_1$ and reason as above that $(\ell, \ell') \in \mathbb{E}'_2$. Since, L is fresh for Σ_1, it follows that neither $\ell \in L$ nor $\ell' \in L$. Then, since Σ'_2 is relevant for L over Σ_2, we have by clause (3) of definition 21 that for any $(\ell_1, \ell_2) \in \mathbb{E}'_2 \setminus \mathbb{E}_2$ it must be that either $\ell_1 \in L$ or $\ell_2 \in L$. Thus, $(\ell, \ell') \notin \mathbb{E}'_2 \setminus \mathbb{E}_2$. Therefore, since $(\ell, \ell') \in \mathbb{E}'_2$, it then follows by simple set-theoretic reasons that $(\ell, \ell') \in \mathbb{E}_2$ as required. Again, a symmetric chain of reasoning demonstrates that $\mathbb{E}_2 \subseteq \mathbb{E}_1$, hence we conclude.

($\mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$): To see $\mathrm{dom}(\rho_1) \subseteq \mathrm{dom}(\rho_2)$, take $\ell \in \mathrm{dom}(\rho_1)$. Since $\Sigma_1 \subseteq \Sigma'_1$, we have $\rho_1 \subseteq \rho'_1$ and thus that $\ell \in \mathrm{dom}(\rho'_1)$. Then, since $\mathrm{dom}(\rho'_1) = \mathrm{dom}(\rho'_2)$, it follows that $\ell \in \mathrm{dom}(\rho'_2)$. Also, $\ell \notin L$ by clause (3) of definition 22 since L is fresh for Σ_1. Thus, since Σ'_2 is relevant for L over Σ_2, we have by clause (2) of definition 21 that $\ell \notin \mathrm{dom}(\rho'_2 \setminus \rho_2)$. However, since we have that $\ell \in \mathrm{dom}(\rho'_2)$, by proposition 8 it must be that $\ell \in \mathrm{dom}(\rho_2)$ as required. A symmetric chain of reasoning shows that $\mathrm{dom}(\rho_2) \subseteq \mathrm{dom}(\rho_1)$, hence we conclude.

($\rho_1(\ell) \in \mathcal{V} \Leftrightarrow \rho_2(\ell) \in \mathcal{V}$): Assume $(\ell, v) \in \rho_1$ for some $v \in \mathcal{V}$; we show that there is some $v' \in \mathcal{V}$ with $(\ell, v') \in \rho_2$.

Since $\Sigma_1 \subseteq \Sigma'_1$, we have $\rho_1 \subseteq \rho'_1$ and thus that $(\ell, v) \in \rho'_1$. Then, since $\rho'_1(\ell) \in \mathcal{V} \Leftrightarrow \rho'_2(\ell) \in \mathcal{V}$, it follows that there is some $v' \in \mathcal{V}$ such that $(\ell, v') \in \rho'_2$. Also, $\ell \notin L$ since L is fresh for Σ_1. Therefore, since Σ'_2 is relevant for L over Σ_2, we have that $\ell \notin \mathrm{dom}(\rho'_2 \setminus \rho_2)$. However, since we have that $(\ell, v') \in \rho'_2$, by proposition 8 it must be that $(\ell, v') \in \rho_2$ as required. A symmetric chain of reasoning shows that the converse direction holds, hence we conclude.

($\rho_1(\ell) = \rho_2(\ell)$ if $\rho_1(\ell) \notin \mathcal{V}$): Assume $\rho_1(\ell) = \iota$ and $\iota \notin \mathcal{V}$. Since $\Sigma_1 \subseteq \Sigma'_1$, we have $\rho_1 \subseteq \rho'_1$ and thus that $\rho'_1(\ell) = \iota$. Then, since $\rho'_1(\ell) \notin \mathcal{V}$ implies $\rho'_1(\ell) = \rho'_2(\ell)$, it follows that $\rho'_2(\ell) = \iota$. Also, $\ell \notin L$ since $\ell \in \mathrm{dom}(\rho_1)$ (by assumption) and L is fresh for Σ_1. Thus, since Σ'_2 is relevant for L over Σ_2, we have by clause (2) of definition 21 that $\ell \notin \mathrm{dom}(\rho'_2 \setminus \rho_2)$. However, since we have that $\rho'_2(\ell) = \iota$, by proposition 8 it must be that $\rho_2(\ell) = \iota$ as required. □

This is used in the proofs of conjectures 14 and 15 and theorem 20 in order to infer the necessary conditions for applying the inductive hypothesis, namely relatedness of the semantics for corresponding sub-fragments of programs.

We now show some simple properties to do with preservation of properness and equivalence of semantics.

Lemma 11. *Suppose $\{\ell\}$ is fresh for Σ, with $\ell \neq \bot$; then Σ is proper if and only if $\Sigma[\ell \mapsto v]$ is.*

Proof. Immediate, by definition 19, since the only difference between the two semantics is the mapping of ℓ to v in the reification functions, and the freshness constraint entails that ℓ does not occur in the binding resolution function or the extension. □

Lemma 12. *Let $v, v' \in \mathcal{V}$ with $\ell \notin dom(\Sigma_\rho)$, $\ell \notin dom(\Sigma'_\rho)$, $\ell \notin ran(\Gamma_1)$, and $\ell \notin ran(\Gamma_2)$ for $\ell \neq \bot$; then:*

1. *$\Sigma \sim \Sigma'$ if and only if $\Sigma[\ell \mapsto v] \sim \Sigma'[\ell \mapsto v']$.*
2. *$\Gamma_1 \sim \Gamma_2$ only if $\Gamma_1[v \mapsto \ell] \sim \Gamma_2[v' \mapsto \ell]$.*

Proof. Immediate, by definition 16. For the case of semantics, the result obtains because we have only updated the reification functions with mappings to value identifiers in both cases. For environments, we have only updated the value identifier mappings, in each case to the same location thus preserving the equality of the ranges. □

Lemma 13. *let Σ and Σ' be semantics and ℓ a location such that there is no ℓ' such that $(\ell, \ell') \in \Sigma_{\mathbb{E}}$ or $(\ell, \ell') \in \Sigma'_{\mathbb{E}}$; then $\Sigma[\{\ell\} \otimes \Delta] \sim \Sigma'[\{\ell\} \otimes \Delta']$ implies $\Sigma \sim \Sigma'$, for all Δ, Δ'.*

Proof. The reification and binding resolution functions are not updated by the join operation. Thus it remains to show that $\Sigma_{\mathbb{E}} = \Sigma'_{\mathbb{E}}$. We show one direction of the inclusion; the other is symmetric. Let $\mathbb{E}_+$ and $\mathbb{E}'_+$ be the extension kernels of $\Sigma[\{\ell\} \otimes \Delta]$ and $\Sigma'[\{\ell\} \otimes \Delta']$, respectively. Suppose $(\ell_1, \ell_2) \in \Sigma_{\mathbb{E}}$. Since $\mathbb{E}_+ = \Sigma_{\mathbb{E}} \cup (\{\ell\} \otimes_{\Sigma_\rho} \Delta)$, thus also $(\ell_1, \ell_2) \in \mathbb{E}_+$. Since $\Sigma[\{\ell\} \otimes \Delta] \sim \Sigma'[\{\ell\} \otimes \Delta']$, it follows that $\mathbb{E}_+ = \mathbb{E}'_+$. Therefore $(\ell_1, \ell_2) \in \mathbb{E}'_+$. Notice that $\ell_1 \neq \ell$ since there is no ℓ' such that $(\ell, \ell') \in \Sigma_{\mathbb{E}}$. Moreover, by definition 11, all pairs in $\{\ell\} \otimes_{\Sigma'_\rho} \Delta'$ are of the form (ℓ, ℓ') for some ℓ'. Thus $(\ell_1, \ell_2) \notin \{\ell\} \otimes_{\Sigma'_\rho} \Delta'$. Since $\mathbb{E}'_+ = \Sigma'_{\mathbb{E}} \cup (\{\ell\} \otimes_{\Sigma'_\rho} \Delta')$ it follows that we must have $(\ell_1, \ell_2) \in \Sigma'_{\mathbb{E}}$. □

We now turn attention to the results of the renaming theory. Conjecture 1 is a corollary of the following property that we conjecture holds of our semantics. It should be possible to prove by induction on syntactic structure.

Conjecture 14. *If $\Sigma; \Gamma \vdash \sigma \hookrightarrow \sigma'$, with $[\![\sigma]\!]_{\Sigma;\Gamma} = (\rightarrowtail', \mathbb{E}', \rho')$, then $\varphi(\sigma, \sigma') = U \cup L \cup C$, where:*

- *$U \subseteq \{\ell \mid \ell \rightarrowtail' \bot\}$,*
- *$L = \{\ell \mid \ell \in \delta(\sigma, \sigma') \vee \exists \ell' \in \delta(\sigma, \sigma').\, \ell \rightarrowtail' \ell'\}$, and*
- *$C \subseteq \{\ell \mid \exists \ell' \neq \bot.\, \ell' \in \mathrm{decl}(\Sigma) \wedge \ell \rightarrowtail' \ell'\}$.*

From this we can immediately derive conjecture 1 by straightforwardly instantiating it with $\sigma \equiv P$ and $\sigma' \equiv P'$, and interpreting with respect to $\Sigma = \Sigma_\bot$ and $\Gamma = \Gamma_\bot$. In this case, notice that $C = \emptyset$.

Conjecture 1. *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 $\ell \rightarrowtail_P \bot$ for all $\ell \in \varphi(P, P') \setminus L$.*

Conjecture 2 is a corollary of the following property that we conjecture to hold of our semantics. Again, it should be possible to prove by induction on syntactic structure.

Conjecture 15. *Let $\Sigma_1 = (\rightarrowtail_1, \mathbb{E}_1, \rho_1)$, $\Sigma_2 = (\rightarrowtail_2, \mathbb{E}_2, \rho_2)$, such that both $[\![\sigma]\!]_{\Sigma_1;\Gamma_1}$ and $[\![\sigma']\!]_{\Sigma_2;\Gamma_2}$ are defined and, moreover, $[\![\sigma]\!]_{\Sigma_1;\Gamma_1} \sim [\![\sigma']\!]_{\Sigma_2;\Gamma_2}$; if D has a partitioning $P \subseteq \mathcal{L}_{/\hat{\mathbb{E}}_1}$, where $D = \{\ell \mid \ell \in (dom_{\mathcal{V}}(\rho_1) \setminus dom(\rightarrowtail_1)) \wedge \rho_1(\ell) \neq \rho_2(\ell)\}$, then also $D \cup \delta(\sigma, \sigma')$ has a partitioning $P' \subseteq \mathcal{L}_{/\hat{\mathbb{E}}'}$, where $[\![\sigma]\!]_{\Sigma_1;\Gamma_1} = (\rightarrowtail', \mathbb{E}', \rho')$.*

Deriving conjecture 2 from this is done by straightforwardly instantiating it with $\sigma \equiv P$ and $\sigma' \equiv P'$, and interpreting with respect to $\Sigma_1 = \Sigma_2 = \Sigma_\bot$ and $\Gamma_1 = \Gamma_2 = \Gamma_\bot$; in this case, notice that $D = \emptyset$.

Conjecture 2. *Let $P \hookrightarrow P'$ be a valid renaming, then $\delta(P, P')$ has a partitioning that is a subset of $\mathcal{L}_{/\hat{\mathbb{E}}_P}$.*

Proposition 5 is a corollary of the following result.

Lemma 16. *Let $[\![\sigma]\!]_{\Sigma;\Gamma} \subseteq \Sigma'$, where for $\Sigma = (\rightarrowtail_\Sigma, \mathbb{E}_\Sigma, \rho_\Sigma)$ and $\Sigma' = (\rightarrowtail, \mathbb{E}, \rho)$, with Σ and Σ' proper; then for some given $\ell \in \mathrm{decl}(\Sigma')$ and $v \in \mathcal{V}$ not occurring in σ or Σ', define the following:*

- *$L = \{\ell' \mid \ell' \in [\ell]_{\hat{\mathbb{E}}} \vee \exists \ell'' \in [\ell]_{\hat{\mathbb{E}}}.\, \ell' \rightarrowtail \ell''\}$;*
- *$\sigma' = \sigma[\ell' \mapsto v \mid \ell' \in L \cap dom(\sigma)]$; and*
- *$\Sigma' = (\rightarrowtail_\Sigma, \mathbb{E}_\Sigma, \rho_\Sigma[\ell' \mapsto v \mid \ell' \in L \cap dom(\rho_\Sigma)])$.*

Furthermore, define Γ' as follows: if there is (a necessarily unique) v' such that $\Gamma(v') \in [\ell]_{\hat{\mathbb{E}}}$ then Γ' behaves as Γ except $\Gamma'(v) = \Gamma(v')$ and $\Gamma'(v') = \Gamma(v)$; otherwise, $\Gamma' = \Gamma$. Then $(\Sigma, \Gamma) \sim (\Sigma', \Gamma')$, $[\![\sigma']\!]_{\Sigma';\Gamma'}$ is defined, and $[\![\sigma]\!]_{\Sigma;\Gamma} \sim [\![\sigma']\!]_{\Sigma';\Gamma'}$.

Proof. By induction on syntactic structure. Given in the Coq formalisation. □

Proposition 5. *Suppose* $[\![P]\!]$ *is defined,* $\ell \in \mathrm{decl}(P)$*, and* $v \in \mathcal{V}$ *does not occur in* P*, then* $P \hookrightarrow P'$ *is a valid renaming, where* $P' = P[\ell' \mapsto v \mid \ell' \in [\ell]_{\hat{\mathbb{E}}_P} \vee \exists \ell'' \in [\ell]_{\hat{\mathbb{E}}_P}.\, \ell' \rightarrowtail_P \ell'']$.

Proof. By straightforward instantiation of lemma 16 with $\sigma \equiv P$, interpreted with respect to $\Sigma = \Sigma_\perp$ and $\Gamma = \Gamma_\perp$. In this case, the definition of P' arises because we have by lemma 9 that $[\![P]\!]$ is relevant for $\mathrm{dom}(P)$ over $\Sigma_\perp$ and thus, by definition 21, it follows that $L \subseteq \mathrm{dom}(P)$. ◻

A.2 Adequacy

Here we give the full definition of our denotational model of behaviour for the OCaml module calculus. We first reiterate the definition of the denotational domain in which we interpret programs.

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:

$$\begin{aligned} \mathbb{M} &= \mathbb{D} + (\mathbb{M} \to \mathbb{M}) + \textbf{wrong} \\ \mathbb{D} &= (\mathcal{V} \rightharpoonup_{\mathrm{fin}} \mathbb{F}) \times (\mathcal{T} \rightharpoonup_{\mathrm{fin}} \mathbb{T}) \times (\mathcal{M} \rightharpoonup_{\mathrm{fin}} \mathbb{M}) \end{aligned}$$

where $\mathbb{T}$ is the domain (defined below) in which we interpret module types. For $d \in \mathbb{D}$ we will write $\iota \in \mathrm{dom}(d)$ to mean that ι is in the domain of the appropriate component of d, and $d(\iota)$ to mean the application of the appropriate component of d to ι. For $d, d' \in \mathbb{D}$, we write $d + d'$ for the module denotation (also in $\mathbb{D}$) for which $(d + d')(\iota) = d'(\iota)$ if $\iota \in \mathrm{dom}(d')$, $(d + d')(\iota) = d(\iota)$ if $\iota \in \mathrm{dom}(d) \setminus \mathrm{dom}(d')$, and undefined otherwise. We define $d + \textbf{wrong}$ to denote **wrong**. We will also sometimes describe an element $d \in \mathbb{D}$ as a (finite) set of pairs of the appropriate sorts of elements.

We interpret module types as elements in the initial algebra $\mathbb{T}$ of the following functor F (in the category of sets):

$$\begin{aligned} F(X) &= D(X) + (\mathcal{M} \times X) \times X + \textbf{wrong} \\ D(X) &= \wp_{\mathrm{fin}}(\mathcal{V}) \times (\mathcal{T} \rightharpoonup_{\mathrm{fin}} X) \times (\mathcal{M} \rightharpoonup_{\mathrm{fin}} X) \end{aligned}$$

For $\tau \in D(\mathbb{T})$ we abuse notation and write $\mathcal{V}(\tau)$ for the first component of τ; we also write $\tau(\iota)$ to mean the application of the appropriate component of τ to ι, and $\mathrm{dom}(\tau)$ to mean the combined domains of the second and third components of τ. For $\tau, \tau \in D(\mathbb{T})$ we write $\tau + \tau'$ for the module type denotation (also in $D(\mathbb{T})$) for which $\mathrm{dom}_{\mathcal{V}}(\tau + \tau') = \mathrm{dom}_{\mathcal{V}}(\tau) \cup \mathrm{dom}_{\mathcal{V}}(\tau')$, with $(\tau + \tau')(\iota) = \tau'(\iota)$ if $\iota \in \mathrm{dom}(\tau')$, $(\tau + \tau')(\iota) = \tau(\iota)$ if $\iota \in \mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')$, and undefined otherwise. We define $\tau + \textbf{wrong}$ to denote **wrong**. We will also sometimes describe an element $\tau \in D(\mathbb{T})$ as a (finite) set of value identifiers and pairs of appropriate elements.

The denotational interpretation function $(\!|\cdot|\!)_\theta$ is defined in fig. 4. It is parameterised by a denotational environment θ mapping value identifiers to elements of $\mathbb{F}$, module type identifiers to elements of $\mathbb{T}$, and module identifiers to pairs consisting of an element of $\mathbb{M}$ and an element of $\mathbb{T}$. This function interprets value expressions in $\mathbb{F}$, module types in $\mathbb{T}$, and module expressions as a pair of an element in $\mathbb{M}$ and an element in $\mathbb{T}$. Thus, for a module expression, $(\!|\cdot|\!)_\theta$ also synthesizes (the meaning of) its corresponding module type. We write $(\!|\sigma|\!)$ to mean $(\!|\sigma|\!)_{\theta_\perp}$, where $\theta_\perp$ is the environment that maps value and module type identifiers to **wrong** and module identifiers to the pair (**wrong**, **wrong**).

As a notational convenience, for $d \in \mathbb{D}$ and $\tau \in D(\mathbb{T})$ we write $\theta + (d, \tau)$ to denote the environment θ updated by the mappings in d, with mappings of module identifiers in d augmented by the corresponding module types in τ. That is, if $x \in \mathrm{dom}(d)$, then $(\theta + (d, \tau))(x) = (d(x), \tau')$ where $\tau' = \tau(x)$ if $x \in \mathrm{dom}(\tau)$ and $\tau' = \textbf{wrong}$ otherwise.

The following coercion operation is used to give meaning to functors and module type annotations.

Definition 23 (Denotational Coercion)**.** The (infix) operator (:), of type $\mathbb{M} \times \mathbb{T} \to \mathbb{M}$, is defined inductively on the structure of module type denotations as follows.

$$d{:}\tau = \begin{cases} V \cup M \cup T & \text{if } d \in \mathbb{D} \wedge \tau \in D(\mathbb{T}) \\ \lambda d'.(d(d'{:}\tau_1)){:}\tau_2 & \text{if } d \in \mathbb{M} \to \mathbb{M} \wedge \tau = ((x, \tau_1), \tau_2) \\ \textbf{wrong} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where } V &= \{(v, d(v)) \mid v \in \mathrm{dom}(d) \wedge v \in \mathcal{V}(\tau)\} \\ &\quad \cup \{(v, \textbf{wrong}) \mid v \notin \mathrm{dom}(d) \wedge v \in \mathcal{V}(\tau)\} \\ M &= \{(x, d(x){:}\tau(x)) \mid x \in \mathrm{dom}(d) \wedge x \in \mathrm{dom}(\tau)\} \\ &\quad \cup \{(x, \textbf{wrong}) \mid x \notin \mathrm{dom}(d) \wedge x \in \mathrm{dom}(\tau)\} \\ T &= \{(t, \tau(t)) \mid t \in \mathrm{dom}(\tau)\} \end{aligned}$$

We also define an operation to 'promote' a module type denotation to a module denotation. This operation reifies the structure of the module type denotation, building constant-valued functors for (sub)modules having a functor type. It is used to define the meaning of module types in various cases.

Definition 24 (Promotion)**.** We define $(\cdot)^* : \mathbb{T} \to \mathbb{M}$ by induction on the structure of module type denotations.

$$\begin{aligned} \textbf{wrong}^* &= \textbf{wrong} \\ \tau^* &= \{(v, \textbf{wrong}) \mid v \in \mathcal{V}(\tau)\} \qquad \text{if } \tau \in D(\mathbb{T}) \\ &\quad \cup \{(t, \tau(t) \mid t \in \mathrm{dom}(\tau)\} \\ &\quad \cup \{(x, \tau(x)^*) \mid x \in \mathrm{dom}(\tau)\} \\ ((x, \tau_1), \tau_2)^* &= \lambda\_.\tau_2{}^* \end{aligned}$$

To prove the adequacy result, we must define how the elements of the set-theoretic semantics of section 3 relate to those of the denotational semantics defined in section 5. We first consider how the meanings of module types in the two semantics are related.

Definition 25. The relation $\tau \models_\rho \Delta$, for a module type denotation $\tau \in \mathbb{T}$ and a semantic description $\Delta \in \mathcal{D}$ w.r.t. a reification function ρ, is defined inductively as follows.

(1) **wrong** $\models_\rho \Delta$ for all Δ, ρ.
(2) $\tau \models_\rho D$, for $\tau \in D(\mathbb{T})$, if:

$$
\begin{aligned}
\llparenthesis x \rrparenthesis_\theta &= \theta(x)\\
\llparenthesis q_1(q_2) \rrparenthesis_\theta &= \text{let } (d', \_) = \llparenthesis q_2 \rrparenthesis_\theta \text{ in match } \llparenthesis q_1 \rrparenthesis_\theta \text{ with}\\
&\quad \mid (d \in \mathbb{M} \to \mathbb{M}, ((x, \tau_1), \tau_2)) \to (d(d'), \tau_2)\\
&\quad \mid (d \in \mathbb{M} \to \mathbb{M}, \_) \to (d(d'), \textbf{wrong})\\
&\quad \mid \_ \to (\textbf{wrong}, \textbf{wrong})
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis q\,.\,x \rrparenthesis_\theta &= \text{match } \llparenthesis q \rrparenthesis_\theta \text{ with}\\
&\quad \mid (d \in \mathbb{D}, \tau \in D(\mathbb{T})) \text{ when } x \in \text{dom}(\tau) \to\\
&\qquad \text{if } x \in \text{dom}(d) \text{ then } (d(x), \tau(x)) \text{ else } (\textbf{wrong}, \tau(x))\\
&\quad \mid (d \in \mathbb{D}, \tau \in D(\mathbb{T})) \text{ when } x \in \text{dom}(d) \to (d(x), \textbf{wrong})\\
&\quad \mid \_ \to (\textbf{wrong}, \textbf{wrong})
\end{aligned}
$$

(a) Semantics of (extended) module paths.

$$
\begin{aligned}
\llparenthesis t \rrparenthesis_\theta &= \theta(t)\\
\llparenthesis p\,.\,t \rrparenthesis_\theta &= \text{let } (\_, \tau) = \llparenthesis p \rrparenthesis_\theta \text{ in if } \tau \in D(\mathbb{T}) \text{ and } t \in \text{dom}(\tau)\\
&\qquad \text{then } \tau(t) \text{ else } \textbf{wrong}\\
\llparenthesis \texttt{functor (}x : M_1\texttt{) -> } M_2 \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M_1 \rrparenthesis_\theta \text{ in}\\
&\qquad \text{let } \tau' = \llparenthesis M_2 \rrparenthesis_{\theta[x \mapsto (\tau^*, \tau)]} \text{ in } ((x, \tau), \tau')\\
\llparenthesis M \texttt{ with module } x \texttt{ = } q \rrparenthesis_\theta &= \text{let } (\_, \tau') = \llparenthesis q \rrparenthesis_\theta \text{ in let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in}\\
&\qquad \text{if } \tau \in D(\mathbb{T}) \text{ then } \llparenthesis M \rrparenthesis_\theta[x \mapsto \tau'] \text{ else } \textbf{wrong}\\
\llparenthesis M \texttt{ with module } x \texttt{ := } q \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in}\\
&\qquad \text{if } \tau \in D(\mathbb{T}) \text{ then } \tau \setminus x \text{ else } \textbf{wrong}\\
\llparenthesis \texttt{sig } S \texttt{ end} \rrparenthesis_\theta &= \llparenthesis S \rrparenthesis_\theta
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis \varepsilon \rrparenthesis_\theta &= \emptyset\\
\llparenthesis \texttt{val } v : \_ \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis s \rrparenthesis_{\theta[v \mapsto \textbf{wrong}]} \text{ in } \{v\} + \tau\\
\llparenthesis \texttt{module } x : M \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in}\\
&\qquad \text{let } \tau' = \llparenthesis s \rrparenthesis_{\theta[x \mapsto (\tau^*, \tau)]} \text{ in } \{(x, \tau)\} + \tau'\\
\llparenthesis \texttt{module type } t \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis s \rrparenthesis_{\theta[t \mapsto \emptyset]} \text{ in } \{(t, \emptyset)\} + \tau\\
\llparenthesis \texttt{module type } t \texttt{ = } M \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in}\\
&\qquad \text{let } \tau' = \llparenthesis s \rrparenthesis_{\theta[t \mapsto \tau]} \text{ in } \{(t, \tau)\} + \tau'\\
\llparenthesis \texttt{include } M \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in if } \tau \in D(\mathbb{T}) \text{ then}\\
&\qquad \text{let } \tau' = \llparenthesis s \rrparenthesis_{\theta + (\tau^*, \tau)} \text{ in } \tau + \tau'\\
&\qquad \text{else } \textbf{wrong}
\end{aligned}
$$

(b) Semantics of module types.

$$
\begin{aligned}
\llparenthesis \texttt{struct } s \texttt{ end} \rrparenthesis_\theta &= \llparenthesis s \rrparenthesis_\theta\\
\llparenthesis \texttt{functor (}x : M\texttt{) -> } m \rrparenthesis_\theta &= \lambda d.\, \text{if } d = \textbf{wrong} \text{ then } \textbf{wrong} \text{ else}\\
&\qquad \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in } \llparenthesis m \rrparenthesis_{\theta[x \mapsto (d:\tau, \tau)]}\\
\llparenthesis m_1(m_2) \rrparenthesis_\theta &= \text{let } (d', \_) = \llparenthesis m_2 \rrparenthesis_\theta \text{ in match } \llparenthesis m_1 \rrparenthesis_\theta \text{ with}\\
&\quad \mid (d \in \mathbb{M} \to \mathbb{M}, ((x, \tau_1), \tau_2)) \to (d(d'), \tau_2)\\
&\quad \mid (d \in \mathbb{M} \to \mathbb{M}, \_) \to (d(d'), \textbf{wrong})\\
&\quad \mid \_ \to (\textbf{wrong}, \textbf{wrong})\\
\llparenthesis m : M \rrparenthesis_\theta &= \text{let } d = \llparenthesis m \rrparenthesis_\theta \text{ in let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in } (d{:}\tau, \tau)\\
\llparenthesis \varepsilon \rrparenthesis_\theta &= (\emptyset, \emptyset)
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis \texttt{let } v \texttt{ = } e \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } f = \llparenthesis e \rrparenthesis_\theta \text{ in let } (d, \tau) = \llparenthesis s \rrparenthesis_{\theta[v \mapsto f]} \text{ in}\\
&\qquad (\{(v, f)\} + d, \{v\} + \tau)\\
\llparenthesis \texttt{module } x \texttt{ = } m \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } (d, \tau) = \llparenthesis m \rrparenthesis_\theta \text{ in}\\
&\qquad \text{let } (d', \tau') = \llparenthesis s \rrparenthesis_{\theta[x \mapsto (d, \tau)]} \text{ in}\\
&\qquad (\{(x, d)\} + d', \{(x, \tau)\} + \tau')\\
\llparenthesis \texttt{module type } t \texttt{ = } M \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } \tau = \llparenthesis M \rrparenthesis_\theta \text{ in let } (d, \tau') = \llparenthesis s \rrparenthesis_{\theta[t \mapsto \tau]} \text{ in}\\
&\qquad (\{(t, \tau)\} + d, \{(t, \tau)\} + \tau')\\
\llparenthesis \texttt{include } m \texttt{ ;; } s \rrparenthesis_\theta &= \text{let } (d, \tau) = \llparenthesis m \rrparenthesis_\theta \text{ in if } d \in \mathbb{D} \text{ then}\\
&\qquad \text{let } (d', \tau') = \llparenthesis s \rrparenthesis_{\theta + (d, \tau)} \text{ in } (d + d', \tau + \tau')\\
&\qquad \text{else } (\textbf{wrong}, \textbf{wrong})
\end{aligned}
$$

(c) Semantics of module expressions.

$$
\llparenthesis \texttt{module } x \texttt{ = } m \texttt{ ;; } P \rrparenthesis_\theta = \text{let } (d, \tau) = \llparenthesis m \rrparenthesis_\theta \text{ in } \llparenthesis P \rrparenthesis_{\theta[x \mapsto (d, \tau)]}
\qquad
\llparenthesis p\,.\,v \rrparenthesis_\theta = \text{let } (d, \_) = \llparenthesis p \rrparenthesis_\theta \text{ in if } v \in \text{dom}(d) \text{ then } d(v) \text{ else } \textbf{wrong}
$$

(d) Semantics of programs and module paths in value expressions.

Figure 4. The denotational semantics of the OCaml calculus.

(i) $\forall \ell \in D$: $\ell \in \text{dom}(\rho)$ and $\rho(\ell) \in \mathcal{V}(\tau)$; and
(ii) $\forall (\ell, \Delta) \in D$: $\ell \in \text{dom}(\rho)$, $\rho(\ell) \in \text{dom}(\tau)$ and $\tau(\rho(\ell)) \models_\rho \Delta$.

(3) $((x, \tau), \tau') \models_\rho ((\ell, \Delta), \Delta')$ if:
$\ell \in \text{dom}(\rho)$, $\rho(\ell) = x$, $\tau \models_\rho \Delta$, and $\tau' \models_\rho \Delta'$.

When $\tau \models_\rho \Delta$ holds, we say that the module type denotation τ models the semantic description Δ (w.r.t. ρ).

This relation satisfies a monotonicity property.

Lemma 17. *If $\tau \models_\rho \Delta$ and $\rho \subseteq \rho'$ then $\tau \models_{\rho'} \Delta$.*

Proof. Straightforward induction on the definition of $\models_\rho$. □

The heart of the refinement result that we show below, from which adequacy follows, is a logical relation asserting that two module denotations both constitute the same 'implementation' of a module description in the set-theoretic semantics with respect to two given reification functions.

Definition 26. For $\Delta \in \mathcal{D}$, $d, d' \in \mathbb{M}$, and reification functions ρ, ρ', the logical relation $\Delta \vdash (\rho, d) \sim (\rho', d')$ is defined inductively on the structure of descriptions as follows.

1. $\Delta \vdash (\rho, \textbf{wrong}) \sim (\rho', \textbf{wrong})$ for all Δ, ρ, and ρ'.
2. $D \vdash (\rho, d) \sim (\rho', d')$, for $d, d' \in \mathbb{D}$, if:
   (a) $\iota \in \text{dom}(d) \Leftrightarrow \exists \ell.\, (\ell \in D \vee \exists \Delta.\, (\ell, \Delta) \in D) \wedge \rho(\ell) = \iota$
   (b) $\iota \in \text{dom}(d') \Leftrightarrow \exists \ell.\, (\ell \in D \vee \exists \Delta.\, (\ell, \Delta) \in D) \wedge \rho'(\ell) = \iota$
   (c) $\forall \ell \in D.\, \rho(\ell) \in \mathcal{V} \wedge \rho'(\ell) \in \mathcal{V} \Rightarrow d(\rho(\ell)) = d'(\rho'(\ell))$
   (d) $\forall (\ell, \Delta) \in D$:
      i. $\rho(\ell) \in \mathcal{T} \Rightarrow d(\rho(\ell)) \models_\rho \Delta$
      ii. $\rho'(\ell) \in \mathcal{T} \Rightarrow d'(\rho'(\ell)) \models_{\rho'} \Delta$

iii. $\rho(\ell) \in \mathcal{M} \wedge \rho'(\ell) \in \mathcal{M} \Rightarrow$
$\Delta \vdash (\rho, d(\rho(\ell))) \sim (\rho', d'(\rho'(\ell)))$

3. $(\ell{:}\Delta_1)\to\Delta_2 \vdash (\rho, d) \sim (\rho', d')$, for $d, d' \in \mathbb{M} \to \mathbb{M}$, if:
$\forall d_1, d_2 \in \mathbb{M}, \rho_1 \supseteq \rho, \rho_2 \supseteq \rho'.$
$\Delta_1 \vdash (\rho_1, d_1) \sim (\rho_2, d_2) \Rightarrow$
$\Delta_2 \vdash (\rho_1, d(d_1)) \sim (\rho_2, d'(d_2))$

This logical relation is also monotone with respect to reification functions.

Lemma 18. *Suppose $\Delta \vdash (\rho_1, d_1) \sim (\rho_2, d_2)$, with $\rho_1 \vdash \Delta$ and $\rho_2 \vdash \Delta$; if $\rho_1 \subseteq \rho_1'$ and $\rho_2 \subseteq \rho_2'$ then $\Delta \vdash (\rho_1', d_1) \sim (\rho_2', d_2)$.*

Proof. Straightforward, by induction. □

Using these relations we can define when two sets of semantics with a corresponding (set-theoretic) semantic and denotational environment, constitute the same context up to renaming.

Definition 27. We define a relation on tuples of semantics and environments by $(\Sigma, \Gamma, \theta) \sim (\Sigma', \Gamma', \theta')$ if and only if:

(1) $(\Sigma, \Gamma) \sim (\Sigma', \Gamma')$;
(2) for all $v \in \mathcal{V}$,
  (i) $\Gamma(v) = \bot \Rightarrow \theta(v) = \mathbf{wrong}$
  (ii) $\Gamma'(v) = \bot \Rightarrow \theta'(v) = \mathbf{wrong}$
  (iii) for all $v' \in \mathcal{V}$, $\Gamma(v) = \Gamma'(v') \Rightarrow \theta(v) = \theta'(v')$
(3) for all $t \in \mathcal{T}$, $\theta(t) \models_{\Sigma_\rho} \Gamma(t)$ and $\theta'(t) \models_{\Sigma'_\rho} \Gamma'(t)$
(4) for all $x \in \mathcal{M}$ with $\theta(x) = (d, \tau)$ and $\theta'(x) = (d', \tau')$,
  (i) $\tau \models_{\Sigma_\rho} \Gamma(x)$ and $\tau' \models_{\Sigma'_\rho} \Gamma'(x)$
  (ii) $\Delta \vdash (\Sigma_\rho, d) \sim (\Sigma'_\rho, d')$, where $\Delta = \Gamma(x) = \Gamma'(x)$

The following property holds.

Lemma 19. *Let $v, v' \in \mathcal{V}$ be value identifiers, $d \in \mathbb{F}$ a value denotation, and $\ell \neq \bot$ a location such that $\ell \notin dom(\Sigma_\rho)$, $\ell \notin dom(\Sigma'_\rho)$, $\ell \notin ran(\Gamma_1)$, and $\ell \notin ran(\Gamma_2)$ for $\ell \neq \bot$; then*

$$(\Sigma, \Gamma, \theta) \sim (\Sigma', \Gamma', \theta') \Rightarrow (\Sigma[\ell \mapsto v], \Gamma[v \mapsto \ell], \theta[v \mapsto d]) \sim (\Sigma'[\ell \mapsto v'], \Gamma'[v' \mapsto \ell], \theta'[v' \mapsto d])$$

Proof. Suppose $(\Sigma, \Gamma, \theta) \sim (\Sigma', \Gamma', \theta')$. By lemma 12 it follows that $\Sigma[\ell \mapsto v] \sim \Sigma'[\ell \mapsto v']$ and $\Gamma[v \mapsto \ell] \sim \Gamma'[v' \mapsto \ell]$. By definition 27, the only extra condition we need to check is clause 2(iii) for v and v', since the additional mapping in each case gives $\Gamma[v \mapsto \ell](v) = \Gamma'[v' \mapsto \ell](v') = \ell$. Notice that we have $\theta[v \mapsto d](v) = \theta'[v' \mapsto d](v') = d$, and so the condition is met. □

We can now show that the set-theoretic semantics refines the denotational semantics.

Theorem 20 (Refinement). *Suppose $\sigma_1 \hookrightarrow \sigma_2$ is a renaming, $[\![\sigma_1]\!]_{\Sigma_1;\Gamma_1} = \Sigma' \sim \Sigma'' = [\![\sigma_2]\!]_{\Sigma_2;\Gamma_2}$ with Σ_1 and Σ_2 proper, and there are θ_1, θ_2 with $(\Sigma_1, \Gamma_1, \theta_1) \sim (\Sigma_2, \Gamma_2, \theta_2)$; then:*

1. *if σ_1, σ_2 are module types, then $\mathcal{D}_{\Sigma_1;\Gamma_1}(\sigma_1) = \mathcal{D}_{\Sigma_2;\Gamma_2}(\sigma_2) = \Delta$ with $(\!|\sigma_1|\!)_{\theta_1} \models_{\Sigma'_\rho} \Delta$ and $(\!|\sigma_2|\!)_{\theta_2} \models_{\Sigma''_\rho} \Delta$;*
2. *if σ_1, σ_2 are module expressions, where $(\!|\sigma_1|\!)_{\theta_1} = (d_1, \tau_1)$ and $(\!|\sigma_2|\!)_{\theta_2} = (d_2, \tau_2)$, then $\mathcal{D}_{\Sigma_1;\Gamma_1}(\sigma_1) = \mathcal{D}_{\Sigma_2;\Gamma_2}(\sigma_2) = \Delta$ with $\tau_1 \models_{\Sigma'_\rho} \Delta$, $\tau_2 \models_{\Sigma''_\rho} \Delta$, and $\Delta \vdash (\Sigma'_\rho, d_1) \sim (\Sigma''_\rho, d_2)$;*
3. *if σ_1 and σ_2 are both value expressions or both programs, then $(\!|\sigma_1|\!)_{\theta_1} = (\!|\sigma_2|\!)_{\theta_2}$.*

Proof. By induction on syntactic structure. We show some of the important cases in detail.

Value Expressions. For value expressions, the result follows straightforwardly by induction using the standard denotational constructions of lambda calculus; we need only to show that (qualified) value identifiers have the same denotation. Let $\sigma \equiv p\,.\,v_\ell$ and $\sigma' \equiv p\,.\,v'_\ell$. Then $\Sigma' = \Sigma_3[\ell \mapsto (v, \ell')]$ and $\Sigma'' = \Sigma_4[\ell \mapsto (v', \ell')]$ for some ℓ', where $[\![p]\!]_{\Sigma_1;\Gamma_1} = \Sigma_3 = (\rightarrowtail, \mathbb{E}, \rho)$ and $[\![p]\!]_{\Sigma_2;\Gamma_2} = \Sigma_4 = (\rightarrowtail, \mathbb{E}, \rho)$, with $\Sigma_3 \sim \Sigma_4$. Moreover, by lemma 6, Σ_3 and Σ_4 are proper. Thus by the inductive hypothesis we have that there is some $D = \mathcal{D}_{\Sigma_1;\Gamma_1}(p) = \mathcal{D}_{\Sigma_2;\Gamma_2}(p)$ with $D \vdash (\rho, d_1) \sim (\rho', d_2)$, where $(\!|p|\!)_{\theta_1} = (d_1, \tau_1)$ and $(\!|p|\!)_{\theta_2} = (d_2, \tau_2)$. There are now two cases to consider, from the definition of the set-theoretic semantics (cf. fig. 3):

($\ell' = \bot$): Then we have $\rho(\ell'') \neq v$ and $\rho'(\ell'') \neq v'$ for all $\ell'' \in D$. Thus it follows from clauses (2a) and (2b) of definition 26 that $v \notin \mathrm{dom}(d_1)$ and $v' \notin \mathrm{dom}(d_2)$. Therefore, by definition (cf. fig. 4), $(\!|p\,.\,v_\ell|\!)_{\theta_1} = (\!|p\,.\,v'_\ell|\!)_{\theta_2} = \mathbf{wrong}$, as required.

($\ell' \neq \bot$): Then we have that $\ell' \in D$ with $\rho(\ell') = v$ and $\rho'(\ell') = v'$. It thus follows from clauses (2a) and (2b) of definition 26, respectively, that $v \in \mathrm{dom}(d_1)$ and $v' \in \mathrm{dom}(d_2)$, and from clause (2c) that $d_1(v) = d_1(\rho(\ell')) = d_2(\rho'(\ell')) = d_2(v')$. Therefore $(\!|p\,.\,v_\ell|\!)_{\theta_1} = (\!|p\,.\,v'_\ell|\!)_{\theta_2}$, as required.

Programs. If σ_1 and σ_2 are value expressions, then the result follows immediately from that for value expressions. When $\sigma_1 \equiv \mathtt{let}\ x_\ell = m_1\ ;;\ P_1$ and $\sigma_2 \equiv \mathtt{let}\ x_\ell = m_2\ ;;\ P_2$, then there are semantics $\Sigma_3 = [\![m_1]\!]_{\Sigma_1;\Gamma_1}$ and $\Sigma_4 = [\![m_2]\!]_{\Sigma_2;\Gamma_2}$ and descriptions $\Delta_1 = \mathcal{D}_{\Sigma_1;\Gamma_1}(m_1)$ and $\Delta_2 = \mathcal{D}_{\Sigma_2;\Gamma_2}(m_2)$ such that $\Sigma' = [\![P_1]\!]_{\Sigma_3[\ell\mapsto x];\Gamma_1[x\mapsto\Delta_1]}$ and $\Sigma'' = [\![P_2]\!]_{\Sigma_4[\ell\mapsto x];\Gamma_2[x\mapsto\Delta_2]}$. By lemma 6, both Σ_3 and Σ_4 are proper. It thus follows trivially from definition 19 that $\Sigma_3[\ell \mapsto x]$ and $\Sigma_4[\ell \mapsto x]$ are proper, since the only difference in the updated semantics is in the reification function. Therefore, by lemma 9, we have that $\Sigma_3[\ell \mapsto x] \subseteq \Sigma'$ with $\Sigma' \setminus \Sigma_3[\ell \mapsto x] \subseteq \mathrm{dom}(P_1)$ and $\mathrm{dom}(P_1)$ fresh for $\Sigma_3[\ell \mapsto x]$, as well as $\Sigma_4[\ell \mapsto x] \subseteq \Sigma''$ with $\Sigma'' \setminus \Sigma_4[\ell \mapsto x] \subseteq \mathrm{dom}(P_2)$ and $\mathrm{dom}(P_2)$ fresh for $\Sigma_4[\ell \mapsto x]$. Hence, by lemma 10, $\Sigma_3[\ell \mapsto x] \sim \Sigma_4[\ell \mapsto x]$. Moreover notice that, by lemma 9, we have that $\Sigma_1 \subseteq \Sigma_3$ and $\Sigma_3 \setminus \Sigma_1 \subseteq \mathrm{dom}(m_1)$ with $\mathrm{dom}(m_1)$ fresh for Σ_1, and also $\Sigma_2 \subseteq \Sigma_4$ and $\Sigma_4 \setminus \Sigma_2 \subseteq \mathrm{dom}(m_2)$ with $\mathrm{dom}(m_2)$ fresh for Σ_2. Therefore, given that neither $\ell \in \mathrm{dom}(m_1)$ nor $\ell \in \mathrm{dom}(m_2)$, and $\Sigma_3[\ell \mapsto x] \sim \Sigma_4[\ell \mapsto x]$, it is then immediate from definition 16 that $\Sigma_3 \sim \Sigma_4$. Now, since $\sigma_1 \hookrightarrow \sigma_2$ is a renaming, so is $m_1 \hookrightarrow m_2$. So, by the inductive hypothesis, $\Delta_1 = \Delta_2 = \Delta$ with $\tau_1 \models_{\rho_3} \Delta$, $\tau_2 \models_{\rho_4} \Delta$ and $\Delta \vdash (\rho_3, d_1) \sim (\rho_4, d_2)$, where $(\!|m_1|\!)_{\theta_1} = (\tau_1, d_1)$ and $(\!|m_2|\!)_{\theta_2} = (\tau_2, d_2)$, with

$\Sigma_3 = (\rightarrowtail_3, \mathbb{E}_3, \rho_3)$ and $\Sigma_3 = (\rightarrowtail_4, \mathbb{E}_4, \rho_4)$. Thus, according to definition 27, we straightforwardly obtain

$$(\Sigma_3[\ell \mapsto x], \Gamma_1[x \mapsto \Delta], \theta_1[x \mapsto (d_1, \tau_1)]) \sim (\Sigma_4[\ell \mapsto x], \Gamma_2[x \mapsto \Delta], \theta_2[x \mapsto (d_2, \tau_2)])$$

since $(\Sigma_3[\ell \mapsto x], \Gamma_1[x \mapsto \Delta]) \sim (\Sigma_4[\ell \mapsto x], \Gamma_2[x \mapsto \Delta])$ is a precondition to the definedness of $\llbracket P_1 \rrbracket_{\Sigma_3[\ell\mapsto x];\Gamma_1[x\mapsto\Delta]}$ and $\llbracket P_2 \rrbracket_{\Sigma_4[\ell\mapsto x];\Gamma_2[x\mapsto\Delta]}$. Finally, this allows us to obtain from the inductive hypothesis that $(\!| P_1 |\!)_{\theta_1[x\mapsto(d_1,\tau_1)]} = (\!| P_2 |\!)_{\theta_2[x\mapsto(d_2,\tau_2)]}$, whence the result follows from the definition of the denotational semantics.

Value specifications. So $\sigma_1 \equiv \texttt{val}\ v_\ell\ :\ \_\ ;;\ S_1$ and $\sigma_2 \equiv \texttt{val}\ v'_\ell\ :\ \_\ ;;\ S_2$, with $\llbracket S_1 \rrbracket_{\Sigma'_1;\Gamma'_1} = \Sigma''_1$ and $\llbracket S_2 \rrbracket_{\Sigma'_2;\Gamma'_2} = \Sigma''_2$ where $\Sigma'_1 = \Sigma_1[\ell \mapsto v]$ and $\Sigma'_2 = \Sigma_2[\ell \mapsto v']$ with $\Gamma'_1 = \Gamma_1[v \mapsto \ell]$ and $\Gamma'_2 = \Gamma_2[v' \mapsto \ell]$. Then, let $D_1 = \mathcal{D}_{\Sigma'_1;\Gamma'_1}(S_1)$ and $D_2 = \mathcal{D}_{\Sigma'_2;\Gamma'_2}(S_2)$. So, $\Sigma' = \Sigma''_1[\{\ell\} \otimes D_1]$ and $\Sigma'' = \Sigma''_2[\{\ell\} \otimes D_2]$. Since Σ_1 and Σ_2 are proper, we have by lemma 11 that Σ'_1 and Σ'_2 are also proper. Moreover, since $\Sigma_1 \sim \Sigma_2$, we have by lemma 12(1) that $\Sigma'_1 \sim \Sigma'_2$. Similarly, since $\Gamma_1 \sim \Gamma_2$, we have by lemma 12(2) that $\Gamma'_1 \sim \Gamma'_2$. Now, take $\theta'_1 = \theta_1[v \mapsto \mathbf{wrong}]$ and $\theta'_2 = \theta_2[v' \mapsto \mathbf{wrong}]$. Thus, by lemma 19, we have $(\Sigma'_1, \Gamma'_1, \theta'_1) \sim (\Sigma'_2, \Gamma'_2, \theta'_2)$. Since $\ell \in \mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2)$, it follows from definition 15 and the fact that Γ_1 and Γ_2 are well-behaved w.r.t. Σ_1 and Σ_2, respectively, that $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma_1)$ and $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma_2)$. Thus $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma'_1)$ and $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma'_2)$, also. Notice too that $\ell \notin \mathrm{dom}(S_1) = \mathrm{dom}(S_2)$. Therefore, by lemma 7, $\ell \notin D_1$ and $\ell \notin D_2$. Furthermore, since ℓ does not appear in Σ_1 or Σ_2 (by definition 15), it follows from lemma 8 that there is no ℓ' such that (ℓ, ℓ') is contained in the extension kernels of Σ''_1 or Σ''_2. Thus by lemma 13 we have $\Sigma''_1 \sim \Sigma''_2$. So, by the inductive hypothesis, we obtain $D_1 = D_2 = D$ with $(\!| S_1 |\!)_{\theta'_1} \models_{\rho'_1} D$ and $(\!| S_2 |\!)_{\theta'_2} \models_{\rho'_2} D$, where ρ'_1 and ρ'_2 are the reification functions of Σ''_1 and Σ''_2, respectively. Then $\mathcal{D}_{\Sigma_1;\Gamma_1}(\sigma_1) = \{\ell\} \oplus_{\Sigma'} D$ and $\mathcal{D}_{\Sigma_2;\Gamma_2}(\sigma_2) = \{\ell\} \oplus_{\Sigma''} D$. We also have, by lemma 9, that $\Sigma'_1 \subseteq \Sigma''_1$ and $\Sigma'_2 \subseteq \Sigma''_2$. Let $\Sigma''_1 = (\rightarrowtail', \mathbb{E}', \rho')$ and $\Sigma''_2 = (\rightarrowtail'', \mathbb{E}'', \rho'')$. Then

- $\Sigma' = (\rightarrowtail', \mathbb{E}' \cup (\{\ell\} \otimes_{\rho'} D), \rho')$; and
- $\Sigma'' = (\rightarrowtail'', \mathbb{E}'' \cup (\{\ell\} \otimes_{\rho''} D), \rho'')$.

We now need to show the following.

1. $\mathcal{D}_{\Sigma_1,\Gamma_1}(\sigma_1) = \mathcal{D}_{\Sigma_2,\Gamma_2}(\sigma_2)$, i.e. $\{\ell\} \oplus_{\rho'} D = \{\ell\} \oplus_{\rho''} D$. By definition 10, it suffices to prove $\exists \ell' \in D.\, \rho'(\ell) = \rho'(\ell')$ if and only if $\exists \ell' \in D.\, \rho''(\ell) = \rho''(\ell')$. We show the 'only if' direction; the other is symmetric. Assume $\ell' \in D$ with $\rho'(\ell) = \rho'(\ell')$. Then by definition 11 $(\ell, \ell') \in \{\ell\} \otimes_{\rho'} D$. Therefore, since $\Sigma' \sim \Sigma''$, we have by definition 16 that also $(\ell, \ell') \in \mathbb{E}'' \cup (\{\ell\} \otimes_{\rho''} D)$. The result is then obtained immediately from definition 19 since, by lemma 6, Σ'' is proper and so $\rho''(\ell) = \rho''(\ell')$.
2. $(\!| \sigma_1 |\!)_{\theta_1} \models_{\rho'} D'$ and $(\!| \sigma_2 |\!)_{\theta_2} \models_{\rho''} D'$, for $D' = \{\ell\} \oplus_{\rho'} D = \{\ell\} \oplus_{\rho''} D$. We show that $(\!| \sigma_1 |\!)_{\theta_1} \models_{\rho'} D'$; showing the other is similar. We distinguish two cases.
   - If there exists some $\ell' \in D$ such that $\rho'(\ell') = v$ then $D' = \{\ell\} \oplus_{\rho'} D = D$ and, by clause 2(i) of definition 25, $(\!| \sigma_1 |\!)_{\theta_1} = \{v\} + (\!| S_1 |\!)_{\theta'_1} = (\!| S_1 |\!)_{\theta'_1}$ since $v \in \mathcal{V}((\!| S_1 |\!)_{\theta'_1})$. Therefore the result follows, by lemma 17, from the fact that $(\!| S_1 |\!)_{\theta'_1} \models_{\rho'_1} D$ and $\rho'_1 \subseteq \rho'$, the latter entailed by $\Sigma'_1 \subseteq \Sigma''_1$.
   - Otherwise, then $\ell \in D' = \{\ell\} \cup D$ and $v \in \mathcal{V}((\!| \sigma_1 |\!)_{\theta_1}) = \{v\} \cup (\!| S_1 |\!)_{\theta'_1}$. Since $\Sigma'_1 \subseteq \Sigma''_1$, and thus $\rho'_1 \subseteq \rho'$, we have by lemma 17 that $(\!| S_1 |\!)_{\theta'_1} \models_{\rho'} D$. Notice that we also thus have $\rho'(\ell) = v$ since $\rho'_1(\ell) = v$. The result then follows straightforwardly by definition 25.

Value definitions. This is similar to the case for value specifications above. Here we have $\sigma_1 \equiv \texttt{let}\ v_\ell\ =\ e_1\ ;;\ s_1$ and $\sigma_1 \equiv \texttt{let}\ v'_\ell\ =\ e_2\ ;;\ s_2$ with $\llbracket e_1 \rrbracket_{\Sigma_1;\Gamma_1} = \Sigma_3$, $\llbracket e_2 \rrbracket_{\Sigma_2;\Gamma_2} = \Sigma_4$, $\llbracket s_1 \rrbracket_{\Sigma'_3;\Gamma'_1} = \Sigma''_3$, and $\llbracket s_2 \rrbracket_{\Sigma'_4;\Gamma'_2} = \Sigma''_4$ where $\Sigma'_3 = \Sigma_3[\ell \mapsto v]$, $\Gamma'_1 = \Gamma_1[v \mapsto \ell]$, $\Sigma'_4 = \Sigma_4[\ell \mapsto v']$, and $\Gamma'_2 = \Gamma_2[v' \mapsto \ell]$, Moreover, let $D'_1 = \mathcal{D}_{\Sigma'_3;\Gamma'_1}(s_1)$ and $D'_2 = \mathcal{D}_{\Sigma'_4;\Gamma'_2}(s_2)$. So, $\Sigma' = \Sigma''_3[\{\ell\} \otimes D'_1]$ and $\Sigma'' = \Sigma''_4[\{\ell\} \otimes D'_2]$. Since $\ell \in \mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2)$, it follows from definition 15 and the fact that Γ_1 and Γ_2 are well-behaved w.r.t. Σ_1 and Σ_2, respectively, that $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma_1)$ and $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma_2)$. Thus $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma'_1)$ and $\ell \notin \mathrm{ran}_{\mathcal{D}}(\Gamma'_2)$, also. Notice too that $\ell \notin \mathrm{dom}(s_1) = \mathrm{dom}(s_2)$. Therefore, by lemma 7, $\ell \notin D_1$ and $\ell \notin D_2$. Furthermore, since ℓ does not appear in Σ_1 or Σ_2 (by definition 15), nor in $\mathrm{dom}(e_1) = \mathrm{dom}(e_2)$, it follows from lemma 9(2) that ℓ does not appear in Σ_3 or Σ_4. Then, by lemma 8, we have that there is no ℓ' such that (ℓ, ℓ') is contained in the extension kernels of Σ''_3 or Σ''_4. Thus by lemma 13 we have $\Sigma''_3 \sim \Sigma''_4$. Now, by lemma 9 we have that $\Sigma'_3 \subseteq \Sigma''_3$ and $\Sigma'_4 \subseteq \Sigma''_4$ (properly), as well as $\Sigma''_3 \setminus \Sigma'_3 \subseteq L$ and $\Sigma''_4 \setminus \Sigma'_4 \subseteq L$ with L fresh for both Σ'_3 and Σ'_4, where $L = \mathrm{dom}(s_1) = \mathrm{dom}(s_2)$. So, by lemma 10, it follows that $\Sigma'_3 \sim \Sigma'_4$ and therefore, by lemma 12(1), that $\Sigma_3 \sim \Sigma_4$. Thus, by the inductive hypothesis, we have $(\!| e_1 |\!)_{\theta_1} = (\!| e_2 |\!)_{\theta_2} = d$. Furthermore, by lemma 6, both Σ_3 and Σ_4 are proper. Thus, by lemma 11 it follows that Σ'_3 and Σ'_4 are proper too. We also have, by lemma 12(2) that $\Gamma'_1 \sim \Gamma'_2$. Now, take $\theta'_1 = \theta_1[v \mapsto d]$ and $\theta'_2 = \theta_2[v' \mapsto d]$. It then follows from lemma 19 that $(\Sigma'_3, \Gamma'_1, \theta'_1) \sim (\Sigma'_4, \Gamma'_2, \theta'_2)$. Thus, another application of the inductive hypothesis derives that $D'_1 = D'_2 = D'$ with $\tau'_1 \models_{\rho'} D'$, $\tau'_2 \models_{\rho''} D'$, and $D' \vdash (\rho', d'_1) \sim (\rho'', d'_2)$, for $(d'_1, \tau'_1) = (\!| s_1 |\!)_{\theta'_1}$ and $(d'_2, \tau'_2) = (\!| s_2 |\!)_{\theta'_2}$, where $\Sigma''_3 = (\rightarrowtail', \mathbb{E}', \rho')$, and $\Sigma''_4 = (\rightarrowtail'', \mathbb{E}'', \rho'')$. We must now show three things.

(i) $\mathcal{D}_{\Sigma_1;\Gamma_1}(\sigma_1) = \{\ell\} \otimes_{\rho'} D' = \{\ell\} \otimes_{\rho''} D' = \mathcal{D}_{\Sigma_2;\Gamma_2}(\sigma_2)$.

(ii) $\{v\} + \tau'_1 \models_{\rho'} \{\ell\} \otimes_{\rho'} D'$ and $\{v'\} + \tau'_2 \models_{\rho''} \{\ell\} \otimes_{\rho''} D'$.

(iii) $D \vdash (\rho', d_1) \sim (\rho'', d_2)$, where $\mathcal{D}_{\Sigma_1;\Gamma_1}(\sigma_1) = \mathcal{D}_{\Sigma_2;\Gamma_2}(\sigma_2) = D$ with $d_1 = \{(v, d)\} + d'_1$ and $d_2 = \{(v', d)\} + d'_2$.

The first two properties hold by the same reasoning as shown in the case for value descriptions above. To show that the last property holds, we consider two cases.

($D = D'$): So there is $\ell' \in D'$ such that $\rho'(\ell') = v$ and $\ell'' \in D'$ such that $\rho''(\ell'') = v'$. Thus, since $D' \vdash (\rho', d'_1) \sim (\rho'', d'_2)$ we have by definition 26 that $v \in \mathrm{dom}(d'_1)$

and $v' \in \mathrm{dom}(d_2')$. Therefore, $\{(v, d)\} + d_1' = d_1'$ and $\{(v', d)\} + d_2' = d_2'$, whence the result follows directly.

($\ell \notin D'$, $D = D' \cup \{\ell\}$): Since $D' \vdash (\rho', d_1') \sim (\rho'', d_2')$ it follows from definition 26 that $v \notin \mathrm{dom}(d_1')$ and $v' \notin \mathrm{dom}(d_2')$. Thus, we have that $\mathrm{dom}(d_1) = \mathrm{dom}(d_1') \cup \{v\}$ and $\mathrm{dom}(d_2) = \mathrm{dom}(d_2') \cup \{v'\}$. Moreover, $d_1(v) = d = d_2(v')$. From these properties, we can derive the result by definition 26. □

Proposition 6 (Adequacy). *$(\!|P|\!) = (\!|P'|\!)$ if $P \hookrightarrow P'$ is valid.*

Proof. By straightforward instantiation of theorem 20 with $\sigma \equiv P$ and $\sigma' \equiv P'$, interpreted with respect to $\Sigma_1 = \Sigma_2 = \Sigma_\perp$, $\Gamma_1 = \Gamma_2 = \Gamma_\perp$, and $\theta_1 = \theta_2 = \theta_\perp$, for which it is straightforward to show that $(\Sigma_\perp, \Gamma_\perp, \theta_\perp) \sim (\Sigma_\perp, \Gamma_\perp, \theta_\perp)$. □