F-ing Modules

Andreas Rossberg
MPI-SWS
rossberg@mpi-sws.org

Claudio V. Russo
Microsoft Research
crusso@microsoft.com

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Abstract

ML modules are a powerful language mechanism for decomposing programs into reusable components. Unfortunately, they also have a reputation for being "complex" and requiring fancy type theory that is mostly opaque to non-experts. While this reputation is certainly understandable, given the many non-standard methodologies that have been developed in the process of studying modules, we aim here to demonstrate that it is undeserved. To do so, we give a *very simple* elaboration semantics for a full-featured, higher-order ML-like module language. Our elaboration defines the meaning of module expressions by a straightforward, compositional translation into vanilla System F_ω (the higher-order polymorphic λ-calculus), under plain F_ω typing environments. We thereby show that ML modules are merely a particular mode of use of System F_ω.

Our module language supports the usual second-class modules with Standard ML-style generative functors and local module definitions. To demonstrate the versatility of our approach, we further extend the language with the ability to package modules as first-class values—a very simple extension, as it turns out. Our approach also scales to handle OCaml-style applicative functor semantics, but the details are significantly more subtle, so we leave their presentation to a future, expanded version of this paper.

Lastly, we report on our experience using the "locally nameless" approach in order to mechanize the soundness of our elaboration semantics in Coq.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, Abstract data types; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Design, Theory

Keywords Type systems, ML modules, abstract data types, existential types, System F, elaboration, first-class modules

1. Introduction

Modularity is essential to the development and maintenance of large programs. Although most modern languages support modular programming and code reuse in one form or another, the languages in the ML family employ a particularly expressive style of module system. The key features shared by all the dialects of the ML module system are their support for hierarchical namespace management (via *structures*), a fine-grained variety of interfaces (via *translucent signatures*), client-side data abstraction (via *functors*), and implementor-side data abstraction (via *sealing*).

Unfortunately, while the utility of ML modules is not in dispute, they have nonetheless acquired a reputation for being "complex". Simon Peyton Jones, in an oft-cited POPL 2003 keynote address [35], likened ML modules to a Porsche, due to their "high power, but poor power/cost ratio". (In contrast, he likened Haskell—extended with various "sexy" type system extensions—to a Ford Cortina with alloy wheels.) Although we disagree with Peyton Jones' amusing analogy, it seems, based on conversations with many others in the field, that the view that ML modules are too complex for mere mortals to understand is sadly predominant.

Why is this so? Are ML modules really more difficult to program/implement/understand than other ambitious modularity mechanisms, such as GHC's *type classes* with type equality coercions [44] or Java's *classes* with generics and wildcards [45]? We think not (although this is obviously a fundamentally subjective question). One can certainly engage in a constructive debate about whether the mechanisms that comprise the ML module system are put together in the ideal way, and in fact the first and third authors have recently done precisely that [11]. But we do not believe that the *design* of the ML module system is the primary source of the "complexity" complaint.

Rather, we believe the problem is that the literature on the *semantics* of ML-style module systems is so vast and fragmented that, to an outsider, it must surely be bewildering. Many non-standard type-theoretic [18, 16, 26, 25, 41, 9] (as well as several *ad hoc*, non-type-theoretic [30, 31, 3]) methodologies have been developed for explaining, defining, studying, and evolving the ML module systems, most with subtle semantic differences that are not spelled out clearly and are known only to experts. As a rich type theory has developed around a number of these methodologies—*e.g.,* the beautiful metatheory of singleton kinds [43]—it is perfectly understandable for someone encountering a paper on module systems for the first time to feel intimidated by the apparent depth and breadth of knowledge required to understand module typechecking, let alone module compilation.

In response to this problem, Dreyer, Crary and Harper [9] developed a unifying type theory, in which previous systems can be understood as sublanguages that selectively include different features. Although formally and conceptually elegant, their unifying account—which relies on singleton kinds, dependent types, and a subtle effect system—still gives one the impression that ML module typechecking requires sophisticated type theory.

In this paper, we take a different approach. Our modest goal is to show once and for all that, contrary to popular belief, the semantics of ML modules is immediately accessible to anyone familiar with System F_ω (the higher-order polymorphic λ-calculus).

TLDI'10, January 23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-891-9/10/01...$10.00

How do we achieve this goal? First, instead of defining the semantics of modules via a "direct" static and dynamic semantics, we employ an *elaboration* semantics in which the meaning of module expressions is defined by a simple, compositional translation into vanilla F_ω, under plain F_ω typing environments. Our approach thus synthesizes elements of the two alternative definitions of Standard ML modules given by Harper and Stone [20] and Russo [37]. Like the former, we define our semantics by elaboration; but whereas Harper and Stone elaborate ML modules into yet another module type system (a variant of Harper-Lillibridge [16]), we elaborate them into F_ω, which is a significantly simpler system. Like the latter, we classify ML modules using F_ω types; our elaboration effectively provides an *evidence translation* for a simplified variant of Russo's semantics, which lacked a dynamic semantics and type soundness proof.

The main task of the elaboration translation is to insert introduction and elimination forms for existential types and universal types in the appropriate places, as well as to infer coercions between various F_ω types. Thus, our approach substantiates the slogan that *ML modules are just a particular mode of use of System F_ω*. While other researchers have given translations from various dialects of ML modules into System F_ω before, we are (to our knowledge) the first to define the semantics of ML modules *directly* in terms of F_ω.

Second, we focus in Sections 4–5 on showing how to typecheck and implement a *representative* ML-style module language—essentially a higher-order variant of Standard ML's—and do *not* attempt to treat all possible variants of ML module semantics. In particular, our language supports only *second-class* modules (not *first-class* modules [16, 38]) and only *generative* functors (not OCaml-style *applicative* functors [25]).

Our main reason for focusing on an SML-like module system is that its semantics is *very simple*. As evidence of this simplicity, the inference rules comprising our elaboration translation are (with only one mild exception) short and sweet. For purposes of comparison, the semantics of Featherweight GJ [21] has roughly the same number of inference rules as our elaboration translation.[1] However, Featherweight GJ only defines a *toy* version of GJ, whereas our elaboration defines the semantics of a *full-featured* programmable module language, omitting no defining feature of Standard ML modules.

As an aside, we note that, for a (higher-order) SML-like module language, the generality of F_ω's higher kinds is only required when the core language supports type constructors—as is the case in ML. Viewed separately, our module elaboration does not rely on higher-kinded type abstraction. Indeed, for a simpler core language with just type (but not type constructor) definitions, all modules can be elaborated to plain System F. (By contrast, the extension to applicative functors, mentioned below, does require higher kinds.)

To demonstrate the versatility of our approach, we show in Section 6 how to extend our language (and its semantics) with the ability to package modules as first-class values. This turns out to be a very easy extension. Our approach also scales (following ideas in Russo's thesis [37]) to handle OCaml-style applicative functor semantics. However, this latter extension is significantly more involved. This makes sense since many of the subtle differences between the various accounts of ML modules in the literature [37, 25, 41, 9] revolve around the semantics of applicative functors. To avoid opening a whole can of worms, we leave the presentation of our applicative functor extension to a future, expanded version of this paper.

[1] Of course, the *complete* semantics of our language would additionally include the static and dynamic semantics of F_ω, but concerning the "effort required to grok", we think it makes more sense to compare the sizes of the *non-standard* components of the semantics.

(identifiers)	X		
(kinds)	K	$::=$	$\ldots$
(types)	T	$::=$	$\ldots \mid P$
(expressions)	E	$::=$	$\ldots \mid P$
(paths)	P	$::=$	M
(modules)	M	$::=$	$X \mid \{B\} \mid M.X \mid$ **fun** $X{:}S \Rightarrow M \mid X\,X \mid X{:>}S$
(bindings)	B	$::=$	**val** $X{=}E \mid$ **type** $X{=}T \mid$ **module** $X{=}M \mid$ **signature** $X{=}S \mid \epsilon \mid B{;}B \mid$ **include** M
(signatures)	S	$::=$	$P \mid \{D\} \mid (X{:}S) \to S \mid S$ **where type** $\overline{X}{=}T$
(declarations)	D	$::=$	**val** $X{:}T \mid$ **type** $X{=}T \mid$ **type** $X{:}K \mid$ **module** $X{:}S \mid$ **signature** $X{=}S \mid \epsilon \mid D{;}D \mid$ **include** S

Figure 1. Syntax of the module language

(types)	**let** B **in** T	$:=$	$\{B;$ **type** $X{=}T\}.X$
(expr's)	**let** B **in** E	$:=$	$\{B;$ **val** $X{=}E\}.X$
(sig's)	**let** B **in** S	$:=$	$\{B;$ **signature** $X{=}S\}.X$
(modules)	**let** B **in** M	$:=$	$\{B;$ **module** $X{=}M\}.X$
	$M_1\,M_2$	$:=$	**let module** $X_1{=}M_1;$ **module** $X_2{=}M_2$ **in** $X_1\,X_2$
	$M{:>}S$	$:=$	**let module** $X{=}M$ **in** $X{:>}S$
	$M{:}S$	$:=$	(**fun** $X{:}S \Rightarrow X)\,M$
(dec's)	**local** B **in** D	$:=$	**include** (**let** B **in** $\{D\}$)
(bindings)	**local** B **in** B'	$:=$	**include** (**let** B **in** $\{B'\}$)

Figure 2. Derived forms

Finally, as a way of corroborating the simplicity of our approach (and also as an excuse to learn Coq), we mechanized the soundness of our elaboration translation in Coq using the "locally nameless" approach of Aydemir *et al.* [1]. Towards the end of the paper (Section 7), we report on our mechanization experience, which, while ultimately successful, was not as pleasant as we had hoped.

In general, we have tried to give this paper the flavor of a brisk tutorial, assuming of the reader no prior knowledge concerning the typechecking and implementation of ML modules. However, this is *not* (intended to be) a tutorial on *programming* with ML modules, nor is it a tutorial on the design considerations that influenced the development of ML modules. For the former, there are numerous sources to choose from, such as Harper's draft book on SML [15] and Paulson's book [34]. For the latter, we refer the reader to Harper and Pierce [19], as well as the early chapters of the second and third authors' PhD theses [37, 6].

2. The Module Language

Figure 1 presents the syntax of our module language. We assume a *core* language consisting of syntax for kinds, types, and expressions, whose details do not matter for our development. The module language is very similar to that of Standard ML, except that functors are higher-order, and signature declarations may be nested inside structures. The syntax contains all the features one would expect to find: value/type/module/signature bindings/declarations; hierarchical structures with projection via the "dot notation"; structure/signature inheritance via **include**; functors and functor signatures; and sealing (aka opaque signature ascription). In some cases, the syntax restricts module expressions in certain positions (*e.g.,* the components of a functor application) to be identifiers X. This

```
signature EQ = {                         signature SET = {
    type t;                                  type set;
    val eq : t × t → bool                    type elem;
};                                           val empty : set;
signature ORD = {                            val add : elem × set → set;
    include EQ;                              val mem : elem × set → bool
    val less : t × t → bool              };
};

module Set = fun Elem : ORD ⇒ {
    type elem = Elem.t;
    type set = list elem;
    val empty = [];
    val add (x, s) = case s of
        | [] ⇒ [x]
        | y :: s' ⇒ if Elem.eq (x, y) then s
                    else if Elem.less (x, y) then x :: s
                    else y :: add (x, s');
    val mem (x, s) = case s of
        | [] ⇒ false
        | y :: s' ⇒ Elem.eq (y, x) or (Elem.less (y, x) and mem (x, s'))
} :> SET where type elem = Elem.t;

module IntSet = Set {type t = int; val eq = Int.eq; val less = Int.less}
```

Figure 3. Example: a functor for sets

is merely to make the semantics of the language that we define in Section 4 as simple as possible. More general variants of these constructs, as well as other constructs such as "local" bindings, are definable as derived forms (Figure 2). Using these derived forms, Figure 3 shows the implementation of a standard Set functor.

One point of note is the notion of *paths*. A path P is the mechanism by which types, values, and signatures may be projected out of modules. In SML and OCaml, paths are syntactically restricted module expressions, such as an identifier X followed by a series of projections. The reason for the syntactic restriction is essentially that not all projections from modules are sensible. For example, consider a module $M = (M' :> \{\textbf{type}\ \mathsf{t};\ \textbf{val}\ \mathsf{x}{:}\mathsf{t}\})$ that defines both an abstract type t and a value x of type t. Then $M.\mathsf{t}$ is not a valid path, because it denotes a type that is not in scope outside of the module. Likewise, $M.\mathsf{x}$ is not valid because it cannot be given a type that makes sense outside of the module.

Here, for simplicity, instead of restricting the *syntax* of paths P, we instead restrict their *semantics*. That is, paths are syntactically just arbitrary module expressions, but the typing rule for paths P will impose additional restrictions on P's signature.

It is worth noting that our more permissive notion of path is what allows us to define very general forms of local module bindings simply as derived syntax (Figure 2).

3. System F$_\omega$

Figure 4 gives the syntax of the variant of System F$_\omega$ that we use as the target of our elaboration translation. It includes record types (where we assume that labels are always disjoint), but is otherwise completely standard. The only point of note is that, unlike in most presentations, our typing environments Γ permit shadowing of bindings for value variables x (but not for type variables α). Allowing shadowing turns out to be convenient for our purposes. The full static semantics is given in Appendix A.1.

We assume a standard left-to-right call-by-value dynamic semantics, which is defined in Appendix A.2. Other choices of evaluation order are possible as well.

Figure 5 defines some syntactic sugar for n-ary pack's and unpack's that introduce/eliminate existential types $\exists\overline{\alpha}.\tau$ quantifying over several type variables at once. We will use n-ary forms

$$\begin{array}{llcl}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa \to \kappa \\
\text{(types)} & \tau & ::= & \alpha \mid \tau \to \tau \mid \{\overline{l{:}\tau}\} \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \\
& & & \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau \\
\text{(terms)} & e, f & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid \{\overline{l{=}e}\} \mid e.l \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \\
& & & \mathsf{pack}\ \langle\tau, e\rangle_\tau \mid \mathsf{unpack}\ \langle\alpha, x\rangle{=}e\ \mathsf{in}\ e \\
\text{(values)} & v & ::= & \lambda x{:}\tau.e \mid \{\overline{l{=}v}\} \mid \lambda\alpha{:}\kappa.e \mid \mathsf{pack}\ \langle\tau, v\rangle_\tau \\
\text{(environ's)} & \Gamma & ::= & \cdot \mid \Gamma, \alpha{:}\kappa \mid \Gamma, x{:}\tau
\end{array}$$

Figure 4. Syntax of F$_\omega$

$$\begin{array}{lcl}
\exists\epsilon.\tau & := & \tau \\
\exists\overline{\alpha}.\tau & := & \exists\alpha_1.\exists\overline{\alpha}'.\tau \\
\\
\mathsf{pack}\ \langle\epsilon, e\rangle_{\exists\epsilon.\tau_0} & := & e \\
\mathsf{pack}\ \langle\overline{\tau}, e\rangle_{\exists\overline{\alpha}.\tau_0} & := & \mathsf{pack}\ \langle\tau_1, \\
& & \quad \mathsf{pack}\ \langle\overline{\tau}', e\rangle_{\exists\overline{\alpha}'.\tau_0[\tau_1/\alpha_1]}\rangle_{\exists\overline{\alpha}.\tau_0} \\
\mathsf{unpack}\ \langle\epsilon, x{:}\tau\rangle = e_1\ \mathsf{in}\ e_2 & := & \mathsf{let}\ x{:}\tau = e_1\ \mathsf{in}\ e_2 \\
\mathsf{unpack}\ \langle\overline{\alpha}, x{:}\tau\rangle = e_1\ \mathsf{in}\ e_2 & := & \mathsf{unpack}\ \langle\alpha_1, x_1\rangle = e_1\ \mathsf{in} \\
& & \quad \mathsf{unpack}\ \langle\overline{\alpha}', x{:}\tau\rangle = x_1\ \mathsf{in}\ e_2 \\
\mathsf{let}\ \overline{x{:}\tau = e_1}\ \mathsf{in}\ e_2 & := & (\lambda\overline{x{:}\tau}.e_2)\ \overline{e_1}
\end{array}$$

$$(\text{where } \overline{\tau} = \tau_1\overline{\tau}' \text{ and } \overline{\alpha} = \alpha_1\overline{\alpha}')$$

Figure 5. Notational abbreviations for F$_\omega$

of other constructs (*e.g.,* application of a type λ), defined in all instances in the obvious way.

Lastly, to ease notation in the elaboration rules that follow, we will typically omit kind annotations on type variables in the environment and on binders. Where needed, we use the notation κ_α to refer to the kind implicitly associated with α. For brevity, we will also usually drop the type annotations from let, pack, and unpack when they are clear from context.

4. Elaboration

We will now define the semantics of the module language by *elaboration* into System F$_\omega$. That is, we will give (syntax-directed) translation rules that interpret signatures as F$_\omega$ types, and modules as F$_\omega$ terms. Our elaboration translation builds on a number of ideas for representing modules that originate in previous work (see Section 8 for a detailed discussion), but we do not assume that the reader is familiar with any of these ideas and thus explain them all from first principles.

Identifiers In order to treat identifier bindings in as simple a manner as possible, we make several assumptions. First, we assume that identifiers X of the module language can be injectively mapped to variables x of F$_\omega$. To streamline the presentation, we assume that this mapping is applied implicitly, and thus we use module-language identifiers as if they were F$_\omega$ variables.

Second, we assume that there is an injective embedding of F$_\omega$ variables into F$_\omega$ *labels*. That is, for every (free) variable x there is a unique label l_x from which x can be reconstructed. Together with the first assumption this means that, wherever we write l_X (with X being a module language identifier), we take this to mean that X has been embedded into the set of F$_\omega$ variables, which in turn has been embedded into the set of labels. Since both embeddings are injective, X uniquely determines l_X and vice versa.

Judgments The judgments comprising our elaboration semantics are listed in Figure 6. Most of these are *translation* judgments, which translate module-language entities into F$_\omega$ entities of the corresponding variety. The last two are auxiliary judgments for signature matching and subtyping, which we will explain a bit later. A number of the elaboration judgments concern *semantic signatures* Ξ or Σ. Semantic signatures are just a subclass of F$_\omega$ types

$$
\begin{array}{ll}
\Gamma \vdash K \rightsquigarrow \kappa & \text{(kind elaboration)} \\
\Gamma \vdash T : \kappa \rightsquigarrow \tau & \text{(type elaboration)} \\
\Gamma \vdash E : \tau \rightsquigarrow e & \text{(expression elaboration)} \\[1ex]
\Gamma \vdash P : \Sigma \rightsquigarrow e & \text{(path elaboration)} \\
\Gamma \vdash M : \Xi \rightsquigarrow e & \text{(module elaboration)} \\
\Gamma \vdash B : \Xi \rightsquigarrow e & \text{(binding elaboration)} \\[1ex]
\Gamma \vdash S \rightsquigarrow \Xi & \text{(signature elaboration)} \\
\Gamma \vdash D \rightsquigarrow \Xi & \text{(declaration elaboration)} \\
\Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \rightsquigarrow f & \text{(signature matching)} \\
\Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f & \text{(signature subtyping)}
\end{array}
$$

Figure 6. Elaboration judgments

$$
\begin{array}{rlll}
\text{(abstract signatures)} & \Xi & ::= & \exists\overline{\alpha}.\Sigma \\
\text{(concrete signatures)} & \Sigma & ::= & [\tau] \mid [= \tau : \kappa] \mid [= \Xi] \mid \\
 & & & \{\overline{l_X : \Sigma}\} \mid \forall\overline{\alpha}.\Sigma \to \Xi
\end{array}
$$

$$
\begin{array}{rlll}
\text{(projection)} & \Sigma.\epsilon & := & \Sigma \\
 & \{l : \Sigma, \overline{l' : \Sigma'}\}.l.\bar{l} & := & \Sigma.\bar{l}
\end{array}
$$

$$
\begin{array}{rlll}
\text{(types)} & [\tau] & := & \{\mathsf{val} : \tau\} \\
 & [= \tau : \kappa] & := & \{\mathsf{typ} : \forall\alpha : (\kappa \to \Omega).\, \alpha\,\tau \to \alpha\,\tau\} \\
 & [= \Xi] & := & \{\mathsf{sig} : \Xi \to \Xi\} \\
\text{(terms)} & [e] & := & \{\mathsf{val} = e\} \\
 & [\tau : \kappa] & := & \{\mathsf{typ} = \lambda\alpha : (\kappa \to \Omega).\, \lambda x : \alpha\,\tau.\, x\} \\
 & [\Xi] & := & \{\mathsf{sig} = \lambda x : \Xi.\, x\}
\end{array}
$$

Figure 7. Semantic signatures

that serve as the semantic interpretations of *syntactic* (*i.e.,* module-language) signatures S, as well as the classifiers of modules M. Since semantic signatures are so central to elaboration, we'll start by explaining how they work.

Semantic Signatures The syntax for semantic signatures is given in Figure 7. (And no, this is not an oxymoron, for in our setting the "semantic objects" we are using to model modules are merely pieces of F$_\omega$ syntax.)

Following Mitchell and Plotkin [32], the basic idea behind semantic signatures is to view a signature as an existential type, with the existential serving as a binder for all the abstract types declared in the signature. In particular, an *abstract* semantic signature Ξ has the form $\exists\overline{\alpha}.\Sigma$, where $\overline{\alpha}$ names all the abstract types declared in the signature, and where Σ is a *concrete* version of the signature. Σ is concrete in the sense that each (formerly) abstract type declaration is made transparently equal to the corresponding existentially-bound variable among the $\overline{\alpha}$. (We will see an example of this shortly.)

A concrete signature Σ, in turn, can be either an *atomic* signature ($[\tau]$, $[= \tau : \kappa]$, or $[= \Xi]$, each denoting a single anonymous value, type, or signature declaration, respectively), a *structure* signature (represented as a record type $\{\overline{l_X : \Sigma}\}$), or a *functor* signature (represented by the polymorphic function type $\forall\overline{\alpha}.\Sigma \to \Xi$).

The atomic signature forms are just syntactic sugar for F$_\omega$ types of a certain form. Their encodings (shown in Figure 7) refer to special labels val, typ, and sig, which we assume are disjoint from the set of labels l_X corresponding to module-language identifiers. Of particular note are the encodings for type and signature declarations, which may seem slightly odd because they both appear to declare a value of the same type as the identity function. This is merely a coding trick: type and signature declarations are only relevant at compile time, and thus the actual values that inhabit these atomic signatures are irrelevant. The important point is that (1) they are inhabited, and (2) the signatures $[= \tau : \kappa]$ and $[= \Xi]$ uniquely (up to F$_\omega$ type equivalence) determine τ and Ξ, respectively. The encoding for $[= \tau : \kappa]$ is chosen such that it supports arbitrary κ.

Signatures $\boxed{\Gamma \vdash S \rightsquigarrow \Xi}$

$$
\frac{\Gamma \vdash P : [= \Xi] \rightsquigarrow e}{\Gamma \vdash P \rightsquigarrow \Xi}
\qquad\qquad
\frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi}
$$

$$
\frac{\Gamma \vdash S_1 \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash S_2 \rightsquigarrow \Xi}{\Gamma \vdash (X{:}S_1) \to S_2 \rightsquigarrow \forall\overline{\alpha}.\,\Sigma \to \Xi}
$$

$$
\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1\alpha\overline{\alpha}_2.\Sigma \qquad \Sigma.\overline{l_X} \equiv [= \alpha : \kappa] \qquad \Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash S \;\textbf{where type}\; \overline{X}{=}T \rightsquigarrow \exists\overline{\alpha}_1\overline{\alpha}_2.\Sigma[\tau/\alpha]}
$$

Declarations $\boxed{\Gamma \vdash D \rightsquigarrow \Xi}$

$$
\frac{\Gamma \vdash T : \Omega \rightsquigarrow \tau}{\Gamma \vdash \textbf{val}\; X{:}T \rightsquigarrow \{l_X : [\tau]\}}
$$

$$
\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type}\; X{=}T \rightsquigarrow \{l_X : [= \tau : \kappa]\}}
$$

$$
\frac{\Gamma \vdash K \rightsquigarrow \kappa_\alpha}{\Gamma \vdash \textbf{type}\; X{:}K \rightsquigarrow \exists\alpha.\{l_X : [= \alpha : \kappa_\alpha]\}}
$$

$$
\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma}{\Gamma \vdash \textbf{module}\; X{:}S \rightsquigarrow \exists\overline{\alpha}.\{l_X : \Sigma\}}
$$

$$
\frac{\Gamma \vdash S \rightsquigarrow \Xi}{\Gamma \vdash \textbf{signature}\; X{=}S \rightsquigarrow \{l_X : [= \Xi]\}}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash D_1 \rightsquigarrow \exists\overline{\alpha}_1.\{\overline{l_{X_1} : \Sigma_1}\} \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash D_2 \rightsquigarrow \exists\overline{\alpha}_2.\{\overline{l_{X_2} : \Sigma_2}\} \qquad \overline{l_{X_1}} \cap \overline{l_{X_2}} = \emptyset\end{array}}{\Gamma \vdash D_1{;}D_2 \rightsquigarrow \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{l_{X_1} : \Sigma_1}, \overline{l_{X_2} : \Sigma_2}\}}
$$

$$
\frac{}{\Gamma \vdash \epsilon \rightsquigarrow \{\}}
\qquad\qquad
\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\}}{\Gamma \vdash \textbf{include}\; S \rightsquigarrow \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\}}
$$

Figure 8. Signature elaboration

Signature Elaboration The elaboration of signatures (Figure 8) is really very straightforward. The only significant difference between a syntactic module-language signature and its semantic interpretation is that, in the latter, all the abstract types declared in the signature are collected together, hoisted out, and bound existentially at the outermost level of the signature.

For example, consider the following syntactic signature:

$$
\begin{array}{c}
\{\textbf{module}\; \mathsf{A} : \{\textbf{type}\; \mathsf{t};\; \textbf{val}\; \mathsf{v} : \mathsf{t}\}; \\
\textbf{signature}\; \mathsf{S} = \{\textbf{val}\; \mathsf{f} : \mathsf{A.t} \to \mathsf{int}\}\}
\end{array}
$$

This signature declares *one* abstract type (A.t), so the semantic F$_\omega$ interpretation of the signature will bind *one* abstract type α:

$$\exists\alpha.\{\; l_\mathsf{A} : \{l_\mathsf{t} : [= \alpha : \Omega],\; l_\mathsf{v} : [\alpha]\},\; l_\mathsf{S} : [= \{l_\mathsf{f} : [\alpha \to \mathsf{int}]\}]\;\}$$

For legibility, in the sequel we'll finesse the injections (l_X) from source identifiers into labels, instead writing this signature as:

$$\exists\alpha.\{\; \mathsf{A} : \{\mathsf{t} : [= \alpha : \Omega],\; \mathsf{v} : [\alpha]\},\; \mathsf{S} : [= \{\mathsf{f} : [\alpha \to \mathsf{int}]\}]\;\}$$

The signature is modeled as a record type with two fields, A and S. The A field has two subfields—t and v—the first of which has an atomic signature denoting that t is a type component equal to α, the second of which has an atomic signature denoting that v is a value component of type α (*i.e.,* t). The S field has an atomic signature

$$\begin{array}{l}\mathsf{SET} \;\leadsto\; \exists\alpha_1\alpha_2.\{\mathsf{set} : [= \alpha_1 : \Omega],\\ \qquad\qquad\quad \mathsf{elem} : [= \alpha_2 : \Omega],\\ \qquad\qquad\quad \mathsf{empty} : [\alpha_1],\\ \qquad\qquad\quad \mathsf{add} : [\alpha_2 \times \alpha_1 \to \alpha_1],\\ \qquad\qquad\quad \mathsf{member} : [\alpha_2 \times \alpha_1 \to \mathsf{bool}]\}\end{array}$$

$$\begin{array}{l}(\mathsf{Elem} : \mathsf{ORD}) \to (\mathsf{SET}\ \textbf{where type}\ \mathsf{t} = \mathsf{Elem.t})\\ \quad \leadsto \forall\alpha.\{\mathsf{t} : [= \alpha : \Omega],\\ \qquad\qquad \mathsf{eq} : [\alpha \times \alpha \to \mathsf{bool}],\\ \qquad\qquad \mathsf{less} : [\alpha \times \alpha \to \mathsf{bool}]\}\\ \qquad\quad \to \exists\beta.\{\mathsf{set} : [= \beta : \Omega],\\ \qquad\qquad\quad \mathsf{elem} : [= \alpha : \Omega],\\ \qquad\qquad\quad \mathsf{empty} : [\beta],\\ \qquad\qquad\quad \mathsf{add} : [\alpha \times \beta \to \beta],\\ \qquad\qquad\quad \mathsf{member} : [\alpha \times \beta \to \mathsf{bool}]\}\end{array}$$

Figure 9. Example: signature elaboration

denoting that S is a signature component whose definition is the semantic signature $\{\mathsf{f} : [\alpha \to \mathsf{int}]\}$.

Note that, by hoisting the binding for the abstract type α to the outermost scope of the signature, we have made the apparent dependency between the declaration of **signature** S and the declaration of **module** A—*i.e.,* the reference in S's declaration to the type A.t—disappear! Moreover, whereas in the original syntactic signature the abstract type was referred to as t in one place and as A.t in another, in the semantic signature all references to the same abstract type component use the same name (here, α). These simplifications (1) make clear that you do not need dependent types in order to model ML signatures, and (2) allow us to avoid any "signature strengthening" (aka "selfification") machinery, of the sort one finds in all the "syntactic" type systems for modules [16, 26, 25, 41, 9].

The only semantic signature form not exhibited in the above example is the functor signature $\forall\overline{\alpha}.\Sigma \to \Xi$. The important point about this signature is that the $\overline{\alpha}$ are universally quantified, which enables them to be mentioned both in the argument signature Σ and the result signature Ξ. If functor signatures were instead represented as $\Xi \to \Xi'$, then the result signature Ξ' would not be able to depend on any abstract types declared in the argument.

An example of a functor signature can be seen in Figure 9. It gives the translations of the signature SET from the example in Figure 3, along with the translation of the signature

$$(\mathsf{Elem} : \mathsf{ORD}) \to (\mathsf{SET}\ \textbf{where type}\ \mathsf{t} = \mathsf{Elem.t})$$

which classifies the Set functor itself.

Given our informal explanation, the formal rules in Figure 8 should now be very easy to follow. A few points of note, though.

The rule for **where type** employs a convenient bit of shorthand notation defined in Figure 7, namely: $\Sigma.\overline{l_X}$ denotes the signature of the $\overline{l_X}$ component of Σ. This is used to check that the type component being refined is in fact an abstract type component (*i.e.,* equivalent to one of the $\overline{\alpha}$ bound existentially by the signature).

In the rule for sequences of declarations $D_1;D_2$, note that the side condition on the label sets $\overline{l_{X_1}}$ and $\overline{l_{X_2}}$ is in place because signatures may not declare two components with the same name. Also, note that the identifiers $\overline{X_1}$, implicitly embedded as F_ω variables, may shadow other bindings in Γ. This is one place where it is convenient to rely on shadowing being permissible in the F_ω environments.

Finally, the rule for signature paths P refers in its premise to the *path* elaboration judgment (which we will discuss later) solely in order to look up the semantic signature Ξ that P should expand to. As noted above in the discussion of atomic signatures, the actual term e inhabiting the atomic signature $[= \Xi]$ is irrelevant.

Matching $\boxed{\Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \leadsto f}$

$$\frac{\overline{\Gamma \vdash \tau : \kappa_\alpha} \qquad \Gamma \vdash \Sigma \leq \Sigma'[\overline{\tau}/\overline{\alpha}] \leadsto f}{\Gamma \vdash \Sigma \leq \exists\overline{\alpha}.\Sigma' \uparrow \overline{\tau} \leadsto f}$$

Subtyping $\boxed{\Gamma \vdash \Xi \leq \Xi' \leadsto f}$

$$\frac{\Gamma \vdash \tau \leq \tau' \leadsto f}{\Gamma \vdash [\tau] \leq [\tau'] \leadsto \lambda x{:}[\tau].[f(x.\mathsf{val})]}$$

$$\frac{\tau \equiv \tau'}{\Gamma \vdash [= \tau : \kappa] \leq [= \tau' : \kappa] \leadsto \lambda x{:}[= \tau : \kappa].x}$$

$$\frac{\Gamma \vdash \Xi \leq \Xi' \leadsto f \qquad \Gamma \vdash \Xi' \leq \Xi \leadsto f'}{\Gamma \vdash [= \Xi] \leq [= \Xi'] \leadsto \lambda x{:}[= \Xi].\, [\Xi']}$$

$$\frac{\overline{\Gamma \vdash \Sigma_1 \leq \Sigma_1' \leadsto f}}{\begin{array}{l}\Gamma \vdash \{\overline{l_1 : \Sigma_1}, \overline{l_2 : \Sigma_2}\} \leq \{\overline{l_1 : \Sigma_1'}\} \leadsto\\ \qquad \lambda x{:}\{\overline{l_1 : \Sigma_1}, \overline{l_2 : \Sigma_2}\}.\{\overline{l_1 = f\,(x.l_1)}\}\end{array}}$$

$$\frac{\Gamma, \overline{\alpha}' \vdash \Sigma' \leq \exists\overline{\alpha}.\Sigma \uparrow \overline{\tau} \leadsto f_1 \qquad \Gamma, \overline{\alpha}' \vdash \Xi[\overline{\tau}/\overline{\alpha}] \leq \Xi' \leadsto f_2}{\begin{array}{l}\Gamma \vdash (\forall\overline{\alpha}.\Sigma \to \Xi) \leq (\forall\overline{\alpha}'.\Sigma' \to \Xi') \leadsto\\ \qquad \lambda f{:}(\forall\overline{\alpha}.\Sigma \to \Xi).\, \lambda\overline{\alpha}'.\, \lambda x{:}\Sigma'.\, f_2\,(f\,\overline{\tau}\,(f_1\,x))\end{array}}$$

$$\frac{\Gamma, \overline{\alpha} \vdash \Sigma \leq \exists\overline{\alpha}'.\Sigma' \uparrow \overline{\tau} \leadsto f}{\begin{array}{l}\Gamma \vdash \exists\overline{\alpha}.\Sigma \leq \exists\overline{\alpha}'.\Sigma' \leadsto\\ \qquad \lambda x{:}(\exists\overline{\alpha}.\Sigma).\, \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = x\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\tau}, f\,y\rangle\end{array}}$$

Figure 10. Signature matching and subtyping

Signature Matching and Subtyping Signature matching (Figure 10) is a key element of the ML module system. At functor applications, we must check that the signature of the actual argument matches the formal argument signature of the functor. For sealed module expressions, we must check that the signature of the module being sealed matches the sealing signature.

What happens during signature matching is really quite simple. First of all, in all places where signature matching occurs, the *source* signature—*i.e.,* the signature of the module being matched—is expressible as a *concrete* semantic signature Σ. (To see why, skip ahead to module elaboration.) The *target* signature—*i.e.,* the signature being matched *against*—on the other hand is abstract. To match against an abstract signature $\exists\overline{\alpha}.\Sigma'$, we must solve for the $\overline{\alpha}$. That is, we must find some $\overline{\tau}$ such that the source signature matches $\Sigma'[\overline{\tau}/\overline{\alpha}]$. (Fortunately, if such a $\overline{\tau}$ exists, it is unique, and there is an easy way of finding it by inspecting Σ—the details are in Section 5.2.) Then, the problem of signature matching reduces to the question of whether Σ is a *subtype* of $\Sigma'[\overline{\tau}/\overline{\alpha}]$, which can be determined by a straightforward structural analysis of the two concrete signatures.

As a simple example, consider matching

$$\{\mathsf{A} : \{\mathsf{t} : [= \mathsf{int} : \Omega],\ \mathsf{u} : [\mathsf{int}],\ \mathsf{v} : [\mathsf{int}]\},\ \mathsf{S} : [= \{\mathsf{f} : [\mathsf{int} \to \mathsf{int}]\}]\ \}$$

against the abstract signature

$$\exists\alpha.\{\ \mathsf{A} : \{\mathsf{t} : [= \alpha : \Omega],\ \mathsf{v} : [\alpha]\},\ \mathsf{S} : [= \{\mathsf{f} : [\alpha \to \mathsf{int}]\}]\ \}$$

from our signature elaboration example (above). The $\overline{\tau}$ returned by the matching judgment would here be simply int, and the subtyping check would determine that the first signature is a width/depth subtype of the second after substituting int for α.

The signature matching judgment has the form $\Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \leadsto f$. It matches a concrete Σ against an abstract Ξ of the form $\exists\overline{\alpha}.\Sigma'$ as described above, synthesizing the solution $\overline{\tau}$ for $\overline{\alpha}$, as well as the coercion f from Σ to $\Sigma'[\overline{\tau}/\overline{\alpha}]$.

While the purpose of signature matching is to relate concrete to abstract signatures, signature *subtyping*, $\Gamma \vdash \Xi \leq \Xi' \leadsto f$, only relates signatures within the same class and synthesizes a respective coercion. Consequently, subtyping is defined by cases on Ξ and Ξ'.

For value declarations, signature subtyping appeals to the subtyping judgment for the core language, $\Gamma \vdash \tau \leq \tau' \leadsto f$. For an ML-like core language, subtyping serves to specialize a more general polymorphic type scheme to a less general one. To take a concrete example, the empty field of the Set functor in Figure 3 would, in ML, receive polymorphic scheme $\forall\beta.\mathsf{list}\,\beta$, but when the functor body is matched against the sealing signature (SET **where type** …), the type of empty would be coerced to the monomorphic type list α (where α represents Elem.t).

For type declarations, we require type equivalence, so subtyping just produces a literal identity coercion.

For signature declarations, we do not require that they are equal (as types), but merely mutual subtypes, because type equivalence would be too fine-grained. In particular, signatures that differ syntactically only in the order of their declarations will elaborate to semantic signatures that differ only in the order in which their existential type variables are bound. Such differences should be inconsequential in the source program, and thus signature equivalence has to be coarse enough to ignore such semantically irrelevant differences.

For structure signatures, we allow both width and depth subtyping. For functor signatures, $\forall\overline{\alpha}.\Sigma \to \Xi$ and $\forall\overline{\alpha}'.\Sigma' \to \Xi'$, subtyping proceeds in the usual contra/co-variant manner. After introducing $\overline{\alpha}'$, we *match* the domains contra-variantly to determine an instantiation $\overline{\tau}$ for $\overline{\alpha}$ such that $\Sigma' \leq \Sigma[\overline{\tau}/\overline{\alpha}]$; then, we (co-variantly) check that the (instantiated) co-domain $\Xi[\overline{\tau}/\overline{\alpha}]$ subtypes Ξ'. This allows for polymorphic specialization, *i.e.,* a more polymorphic functor signature may subtype a less polymorphic one.

Dually, for abstract semantic signatures $\exists\overline{\alpha}.\Sigma$ and $\exists\overline{\alpha}'.\Sigma'$, subtyping recursively reduces to eliminating $\exists\overline{\alpha}.\Sigma$, then *matching* Σ against Σ' to determine witness types $\overline{\tau}$ for $\overline{\alpha}'$; thus, a less abstract signature may subtype a more abstract one.

The coercion terms f synthesized by the subtyping rules are straightforward—given the required invariant, $\Gamma \vdash f : \Xi \to \Xi'$, they practically write themselves. The invariant also determines the elided pack annotation in the last rule.

Module Elaboration The module elaboration judgment (Figure 11), which has the form $\Gamma \vdash M : \Xi \leadsto e$, assigns module M the semantic signature Ξ and additionally translates M to an F_ω term e of type Ξ. (The invariant, $\Gamma \vdash e : \Xi$, determines elided pack annotations.) As in signature elaboration, the basic idea in module elaboration is to assign M an abstract signature $\exists\overline{\alpha}.\Sigma$ such that $\overline{\alpha}$ represent all the abstract types that M defines. The difference here is that we must also construct the term e that has this signature (*i.e.,* the *evidence*).

The easiest way to understand the evidence construction is to think of the existential type $\exists\overline{\alpha}.\Sigma$ as a *monad* that encapsulates the "effect" of defining abstract types. When we want to use a module of this signature, we must first unpack it (think: monadic bind), obtaining some fresh abstract types $\overline{\alpha}$ and a variable x of type Σ. We can then do whatever we want with x, ultimately producing another module of signature $\exists\overline{\alpha}'.\Sigma'$. Of course, Σ' may have free references to the $\overline{\alpha}$, so at the end we must repack the result with the $\overline{\alpha}$ to form a module of signature $\exists\overline{\alpha}\,\overline{\alpha}'.\Sigma'$. Thus, the abstract types $\overline{\alpha}$ defined by M propagate monadically into the set of abstract types defined by any module that uses M. While—as many researchers have pointed out—this monadic unpack/repack style of existential programming would be annoying to program manually, it is nonetheless easy for module elaboration to do automatically.

Figure 11 shows the rules for elaborating modules and bindings. The rules for projections ($M.X$), module bindings, and binding sequences ($B_1;B_2$) show the unpack/repack idiom in action. The last of these is somewhat involved, but only because ML modules allow bindings to be *shadowed*—a practical complication, incidentally, that is glossed over in most module type systems in the literature.[2]

Modules $\boxed{\Gamma \vdash M : \Xi \leadsto e}$

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X : \Sigma \leadsto X} \qquad \frac{\Gamma \vdash B : \Xi \leadsto e}{\Gamma \vdash \{B\} : \Xi \leadsto e}$$

$$\frac{\Gamma \vdash M : \exists\overline{\alpha}.\{l_X : \Sigma, \overline{l_{X'} : \Sigma'}\} \leadsto e}{\Gamma \vdash M.X : \exists\overline{\alpha}.\Sigma \leadsto \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, y.l_X\rangle}$$

$$\frac{\Gamma \vdash S \leadsto \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash M : \Xi \leadsto e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M : \forall\overline{\alpha}.\,\Sigma \to \Xi \leadsto \lambda\overline{\alpha}.\lambda X{:}\Sigma.e}$$

$$\frac{\Gamma(X_1) = \forall\overline{\alpha}.\,\Sigma' \to \Xi \qquad \Gamma(X_2) = \Sigma \qquad \Gamma \vdash \Sigma \leq \exists\overline{\alpha}.\Sigma' \uparrow \overline{\tau} \leadsto f}{\Gamma \vdash X_1\,X_2 : \Xi[\overline{\tau}/\overline{\alpha}] \leadsto X_1\,\overline{\tau}\,(f\,X_2)}$$

$$\frac{\Gamma(X) = \Sigma \qquad \Gamma \vdash S \leadsto \Xi \qquad \Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \leadsto f}{\Gamma \vdash X{:>}S : \Xi \leadsto \mathsf{pack}\ \langle\overline{\tau}, f\,X\rangle}$$

Bindings $\boxed{\Gamma \vdash B : \Xi \leadsto e}$

$$\frac{\Gamma \vdash E : \tau \leadsto e}{\Gamma \vdash \textbf{val}\ X{=}E : \{l_X : [\tau]\} \leadsto \{l_X = [e]\}}$$

$$\frac{\Gamma \vdash T : \kappa \leadsto \tau}{\Gamma \vdash \textbf{type}\ X{=}T : \{l_X : [= \tau : \kappa]\} \leadsto \{l_X = [\tau : \kappa]\}}$$

$$\frac{\Gamma \vdash M : \exists\overline{\alpha}.\Sigma \leadsto e}{\begin{array}{l}\Gamma \vdash \textbf{module}\ X{=}M : \exists\overline{\alpha}.\{l_X : \Sigma\} \\ \quad \leadsto \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \{l_X = x\}\rangle\end{array}}$$

$$\frac{\Gamma \vdash S \leadsto \Xi}{\Gamma \vdash \textbf{signature}\ X{=}S : \{l_X : [= \Xi]\} \leadsto \{l_X = [\Xi]\}}$$

$$\frac{\begin{array}{r}\Gamma \vdash B_1 : \exists\overline{\alpha}_1.\{\overline{l_{X_1} : \Sigma_1}\} \leadsto e_1 \quad \overline{l'_{X_1}} = \overline{l_{X_1}} - \overline{l_{X_2}} \\ \Gamma, \overline{\alpha}_1, \overline{X_1 : \Sigma_1} \vdash B_2 : \exists\overline{\alpha}_2.\{\overline{l_{X_2} : \Sigma_2}\} \leadsto e_2 \quad \overline{l'_{X_1}{:}\Sigma'_1} \subseteq \overline{l_{X_1}{:}\Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash B_1;B_2 : \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{l'_{X_1} : \Sigma'_1}, \overline{l_{X_2} : \Sigma_2}\} \\ \quad \leadsto \mathsf{unpack}\ \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in} \\ \qquad \mathsf{unpack}\ \langle\overline{\alpha}_2, y_2\rangle = (\mathsf{let}\ \overline{X_1 = y_1.l_{X_1}}\ \mathsf{in}\ e_2)\ \mathsf{in} \\ \qquad\quad \mathsf{pack}\ \langle\overline{\alpha}_1\overline{\alpha}_2, \{\overline{l'_{X_1} = y_1.l'_{X_1}}, \overline{l_{X_2} = y_2.l_{X_2}}\}\rangle\end{array}}$$

$$\frac{}{\Gamma \vdash \epsilon : \{\} \leadsto \{\}} \qquad \frac{\Gamma \vdash M : \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\} \leadsto e}{\Gamma \vdash \textbf{include}\ M : \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\} \leadsto e}$$

Figure 11. Module elaboration

The rule for functors is completely analogous to the one for functor signatures from Figure 8. Note that this rule and the binding sequence rule are the only two that extend the environment Γ, and that in both cases the new variable X is bound with a *concrete* signature Σ. As a result, when we look up an identifier X in the environment (as in the identifier elaboration rule), we may assume it has a concrete signature.

The rules for functor applications ($X_1\,X_2$) and sealed modules ($X :> S$) both appeal to the signature matching judgment. In the former, the $\overline{\tau}$ represent the type components of the actual functor argument corresponding to the abstract types $\overline{\alpha}$ declared in the for-

[2] A realistic implementation of modules would want to optimize the construction of structure representations and avoid the repeated record concatenation. Such an optimization is fairly easy, it essentially boils down to partially evaluating the expressions generated by our sequencing rule.

$$
\begin{array}{l}
\mathsf{Set} \;\rightsquigarrow \\
\quad \lambda\alpha.\lambda \mathit{Elem} : \{\mathsf{t} : [= \alpha : \Omega], \\
\qquad\qquad\qquad \mathsf{eq} : [\alpha \times \alpha \to \mathsf{bool}], \\
\qquad\qquad\qquad \mathsf{less} : [\alpha \times \alpha \to \mathsf{bool}]\}. \\
\qquad \mathsf{pack}\,\langle \mathsf{list}\,\alpha, \\
\qquad\qquad f\,(\mathsf{let}\ y_1 = \{\mathsf{elem} = [\alpha]\}\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ y_1' = \\
\qquad\qquad\qquad \mathsf{let}\ \mathit{elem} = y_1.\mathsf{elem}\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ y_2 = \{\mathsf{set} = [\mathsf{list}\,\alpha]\}\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ y_2' = \\
\qquad\qquad\qquad\quad \mathsf{let}\ \mathit{set} = y_2.\mathsf{set}\ \mathsf{in} \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\quad \mathsf{in}\ \{\mathsf{elem} = y_1.\mathsf{elem}, \mathsf{set} = y_1'.\mathsf{set}, \\
\qquad\qquad\qquad \mathsf{empty} = y_1'.\mathsf{empty}, \\
\qquad\qquad\qquad \mathsf{add} = y_1'.\mathsf{add}, \mathsf{mem} = y_1'.\mathsf{mem}\}) \\
\qquad\qquad \rangle_{\exists\beta.\{\mathsf{set}:[=\beta:\Omega],\ \mathsf{elem}:[=\alpha:\Omega],\ \mathsf{empty}:[\beta],\ \mathsf{add}:[\ldots],\ \mathsf{mem}:[\ldots]\}}
\end{array}
$$

$$
\begin{array}{l}
\{\mathbf{module}\ \mathsf{IS} = \mathsf{Set\ Int};\ \mathbf{val}\ \mathsf{s} = \mathsf{IS.add}\,(7,\ \mathsf{IS.empty})\} \;\rightsquigarrow \\
\quad \mathsf{unpack}\,\langle \beta, y_1\rangle = \{\mathsf{IS} = \mathit{Set}\ \mathsf{int}\ (f\ \mathit{Int})\}\ \mathsf{in} \\
\quad \mathsf{let}\ y_2 = (\mathsf{let}\ \mathit{IS} = y_1.\mathsf{IS}\ \mathsf{in}\ \{\mathsf{s} = [\mathit{IS}.\mathsf{add}\,\langle 7, \mathit{IS}.\mathsf{empty}\rangle]\})\ \mathsf{in} \\
\quad \mathsf{pack}\,\langle \beta, \{\mathsf{IS} = y_1.\mathsf{IS}, \mathsf{s} = y_2.\mathsf{s}\}\rangle_{\exists\beta.\{\mathsf{IS}:\{\ldots\},\mathsf{s}:[\beta]\}}
\end{array}
$$

Figure 12. Example: module elaboration

mal argument signature. For instance, in the functor application in Figure 3, $\overline{\tau}$ would be simply int, since that is how the argument module defines the abstract type t declared in the argument signature ORD. This information is then propagated to the result of the functor application by substituting $\overline{\tau}$ for $\overline{\alpha}$ in the result signature Ξ. The sealing rule works similarly, except that $\overline{\tau}$ is not used to eliminate a universal type, but dually, to *introduce* an *existential* type. Hence, $\overline{\tau}$ is not propagated to the signature of the sealed module, but rather hidden within the existential. This makes sense because of course the point of sealing is to hide the identity of the abstract types $\overline{\alpha}$.

As an example of the translation, Figure 12 sketches the result of elaborating the Set functor from Figure 3. It also shows the F_ω representation of a simple program involving the application of this functor. We assume that there is a suitable library module Int that matches signature ORD, and whose F_ω representation is *Int*. In order to avoid too much clutter, we do not spell out the respective coercions f occurring in both examples.

Generativity Functors in Standard ML are said to behave *generatively*, meaning that every application of a functor F will have the effect of generating fresh abstract types corresponding to whichever types are declared abstractly in F's result signature. With the existential interpretation of type abstraction that we employ here, this generativity comes for free. Applying a functor produces a module with an existential type of the form $\exists\overline{\alpha}.\Sigma$. Thus, if a functor is applied twice (say, to the same argument) and the results are bound to two different identifiers X_1 and X_2, then the binding sequence rule will ensure that two separate copies of the $\overline{\alpha}$ will be added to the environment Γ—call them $\overline{\alpha}_1$ and $\overline{\alpha}_2$—along with the bindings $X_1 : \Sigma[\overline{\alpha}_1/\overline{\alpha}]$ and $X_2 : \Sigma[\overline{\alpha}_2/\overline{\alpha}]$. In this way, the abstract type components of X_1 and X_2 will be made distinct.

Path Elaboration Figure 13 displays the last three rules of elaboration, concerning the elaboration of paths. (The elaboration rule for signature paths appeared in Figure 8.)

Paths are the means by which value, type, and signature components are projected out of modules. As explained in Section 2, in order for paths to make sense, the values, types, or signatures that they project out must be well-formed in the ambient environment. To ensure this, the path elaboration judgment, $\Gamma \vdash P : \Sigma \rightsquigarrow e$,

Paths $\boxed{\Gamma \vdash P : \Sigma \rightsquigarrow e}$

$$\frac{\Gamma \vdash P : \exists\overline{\alpha}.\Sigma \rightsquigarrow e \qquad \Sigma \equiv \Sigma' \qquad \Gamma \vdash \Sigma' : \Omega}{\Gamma \vdash P : \Sigma' \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x\rangle = e\ \mathsf{in}\ x}$$

Types $\boxed{\Gamma \vdash T : \kappa \rightsquigarrow \tau}$

$$\frac{\Gamma \vdash P : [= \tau : \kappa] \rightsquigarrow e}{\Gamma \vdash P : \kappa \rightsquigarrow \tau}$$

Expressions $\boxed{\Gamma \vdash E : \tau \rightsquigarrow e}$

$$\frac{\Gamma \vdash P : [\tau] \rightsquigarrow e}{\Gamma \vdash P : \tau \rightsquigarrow e.\mathsf{val}}$$

Figure 13. Path elaboration

uses the ordinary module elaboration judgment to synthesize P's semantic signature $\exists\overline{\alpha}.\Sigma$, and then checks that Σ does not actually depend on any of the "local" abstract types $\overline{\alpha}$ that P may have defined. The rules for type, expression, and signature paths use the path elaboration judgment to check the well-formedness of the path, and then project the component out accordingly.

For instance, consider the example from Section 2 of an ill-formed path. Let M be the module expression

$$\{\mathbf{type}\ \mathsf{t} = \mathsf{int};\ \mathbf{val}\ \mathsf{v} = 3\} :> \{\mathbf{type}\ \mathsf{t};\ \mathbf{val}\ \mathsf{v} : \mathsf{t}\}$$

The semantic signature that module elaboration assigns to M is:

$$\exists\alpha.\{\mathsf{t} : [= \alpha : \Omega], \mathsf{v} : [\alpha]\}$$

Thus, if we try to project either t or v from M directly, the resulting type or expression would not be well-formed, since both $[= \alpha : \Omega]$ and $[\alpha]$ refer to the local abstract type α. If, on the other hand, we were to first bind M to an identifier X, and then subsequently project out X.t or X.v, the paths *would* be well-formed. The reason is that the binding sequence rule would extend the ambient environment with a fresh α, as well as $X : \{\mathsf{t} : [= \alpha : \Omega], \mathsf{v} : [\alpha]\}$. Under such an extended environment, X.t would simply elaborate to α, and X.v would elaborate to X.v.val of type α, both of which are well-formed since α is now bound in the environment. In general, since identifiers have concrete signatures, any well-formed module of the form $X.\overline{l_Y}$ will also be a well-formed path.

If one views existential types as a monad, then the path elaboration rule may seem superficially odd because it allows one to "escape" the monad by going from $\exists\overline{\alpha}.\Sigma$ to Σ. However, the point is that one can only do this if the "effects" encapsulated by the monad—*i.e.,* the abstract types $\overline{\alpha}$ defined by the path—are strictly local. This is similar conceptually to the hiding of "benign" effects by Haskell's runST mechanism [23].

5. Metatheoretic Properties

Don't believe a type system until you have shown it unsound [10], or better yet, proven it sound. We will do so in this section, with a proof that we mechanized in the Coq proof assistant [4]. We also prove that our elaboration is decidable, which is an important property if the semantics is to be useful. Finally, we give some additional properties of signature matching.

5.1 Soundness

Proving soundness of a language specified by an elaboration semantics consists of two steps:

1. Proving that elaboration only produces well-typed terms of the target language.
2. Showing that the type system of the target language is sound.

Fortunately, in our case, since the target language is the very well-studied System F_ω, we can simply borrow the second part from the literature. It remains to be shown that the elaboration rules produce well-formed F_ω programs. Of course, since our development is parametric in the concrete choice of a core language, the result only holds relative to suitable assumptions about the soundness of the elaboration rules for the core language.

Theorem 1 (Soundness of Elaboration)
Provided $\Gamma \vdash \Box$ we have:

1. If $\Gamma \vdash T : \kappa \rightsquigarrow \tau$, then $\Gamma \vdash \tau : \kappa$.
2. If $\Gamma \vdash E : \tau \rightsquigarrow e$, then $\Gamma \vdash e : \tau$.
3. If $\Gamma \vdash \tau \leq \tau' \rightsquigarrow f$ and $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$, then $\Gamma \vdash f : \tau \to \tau'$.
4. If $\Gamma \vdash S \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
5. If $\Gamma \vdash D \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
6. If $\Gamma \vdash M : \Xi \rightsquigarrow e$, then $\Gamma \vdash e : \Xi$.
7. If $\Gamma \vdash B : \Xi \rightsquigarrow e$, then $\Gamma \vdash e : \Xi$.
8. If $\Gamma \vdash P : \Sigma \rightsquigarrow e$, then $\Gamma \vdash e : \Sigma$.
9. If $\Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f$ and $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$, then $\Gamma \vdash f : \Xi \to \Xi'$.
10. If $\Gamma \vdash \Sigma \leq \exists\overline{\alpha}.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f$ and $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Sigma' : \Omega$, then $\overline{\Gamma \vdash \tau : \kappa_\alpha}$ and $\Gamma \vdash f : \Sigma \to \Sigma'[\overline{\tau}/\overline{\alpha}]$.

Proof (sketch): The proof is by relatively straightforward simultaneous induction on derivations. The arguments for properties 1-3 clearly depend on the core language. We have performed the entire proof in Coq (Section 7), and transliterate only two interesting cases here:

Case $X_1\ X_2$ By induction we know that (1) $\overline{\Gamma \vdash \tau : \kappa_\alpha}$ and (2) $\Gamma \vdash f : \Sigma' \to \Sigma[\overline{\tau}/\overline{\alpha}]$. From (1) we can derive that $\Gamma \vdash X_1\ \overline{\tau} : (\Sigma \to \Xi)[\overline{\tau}/\overline{\alpha}]$. From (2) it follows that $\Gamma \vdash f\ X_2 : \Sigma[\overline{\tau}/\overline{\alpha}]$. Thus, we can conclude $\Gamma \vdash X_1\ \overline{\tau}\ (f\ X_2) : \Xi[\overline{\tau}/\overline{\alpha}]$ by the typing rule for application.

Case $B_1;B_2$ By induction on the first premise we know that (1) $\Gamma \vdash e_1 : \exists\overline{\alpha}_1.\{\overline{l_{X_1} : \Sigma_1}\}$. Let $\Gamma_1 = \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1}$. By validity and inversion, from (1) we derive $\overline{\Gamma, \overline{\alpha}_1 \vdash \Sigma_1 : \Omega}$, so $\Gamma_1 \vdash \Box$. By induction on the second premise, (2) $\Gamma_1 \vdash e_2 : \exists\overline{\alpha}_1.\{\overline{l_{X_2} : \Sigma_2}\}$. It is easy to show $\overline{\Gamma, \overline{\alpha}_1, y_1{:}\{\overline{l_{X_1} : \Sigma_1}\} \vdash y_1.l_{X_1} : \Sigma_1}$. By convention, y_1 and y_2 are fresh, so $\Gamma, \overline{\alpha}_1, y_1{:}\{\overline{l_{X_1} : \Sigma_1}\}, \overline{\alpha}_2, y_2{:}\{\overline{l_{X_2} : \Sigma_2}\} \vdash \{\overline{l'_{X_1} = y_1.l'_{X_1}}, \overline{l_{X_2} = y_2.l_{X_2}}\} : \{\overline{l'_{X_1} : \Sigma'_1}, \overline{l_{X_2} : \Sigma_2}\}$ as well. From (1) and weakening (2), the overall goal follows by inner induction on the lengths of $\overline{\alpha}_1$, $\overline{\alpha}_2$, and $\overline{l_{X_1}}$, and expanding the n-ary versions of pack, unpack and let. ■

5.2 Decidability

All our elaboration rules are syntax-directed, so they can be interpreted directly as an algorithm. Provided core elaboration is terminating, this algorithm clearly terminates as well.

There is one niggle, though: the signature matching rule requires a non-deterministic guess of suitable instantiating types $\overline{\tau}$. To prove elaboration decidable, we must provide a sound and complete algorithm for finding these types. It's not obvious that such an algorithm should exist at all. For example, consider the following matching problem (cf. [9]):

$$\forall\alpha.[= \alpha : \kappa] \to [= \tau_1 : \kappa'] \quad \leq_\beta \quad [= \beta : \kappa] \to [= \tau_2 : \kappa']$$

The matching rule must find an instantiation type $\tau : \kappa$ for β such that the left signature is a subtype of $[= \tau : \kappa] \to [= \tau_2[\tau/\beta] : \kappa']$, which in turn will only hold if $\tau_1[\tau/\alpha] \equiv \tau_2[\tau/\beta]$. Since κ may be a higher kind, this amounts to a higher-order unification problem, which is undecidable in general [14].

Fortunately, under minimal assumptions about the initial environment, we can show that such problematic cases never arise during elaboration. More precisely, we can show that, whenever we invoke $\Sigma \leq \exists\overline{\alpha}.\Sigma'$, the target signature Σ' has the property that each abstract type variable $\alpha \in \overline{\alpha}$ actually occurs *explicitly* in the form of an embedded type field $[= \alpha : \kappa_\alpha]$. We say that α is *rooted* in Σ' in this case. An abstract signature in which *all* quantified variables are rooted is called *explicit*.

Figure 14 gives a judgmental definition of these properties. However, this is not all. Subtyping is contra-variant for functors, so we also need to ensure that, whenever we invoke subtyping to determine whether $\Sigma \leq \Sigma'$ and Σ is a *functor* signature, its argument signature is explicit as well. The figure hence defines the second notion of a *valid* signature that captures this property and extends it to environments. Ultimately, we require all signatures and environments used in elaboration to be valid.

We can show that validity and explicitness of signatures are established and maintained by our elaboration:

Lemma 2 (Signature Validity)

1. If Ξ explicit, then Ξ valid.
2. If Ξ explicit (valid), then $\Xi[\overline{\tau}/\overline{\alpha}]$ explicit (valid).
3. If Γ valid and $\Gamma \vdash S \rightsquigarrow \Xi$ or $\Gamma \vdash D \rightsquigarrow \Xi$, then Ξ explicit.
4. If Γ valid and $\Gamma \vdash M : \Xi \rightsquigarrow e$ or $\Gamma \vdash B : \Xi \rightsquigarrow e$, then Ξ valid.
5. If Γ valid and $\Gamma \vdash P : \Sigma \rightsquigarrow e$, then Σ valid.

If the $\exists\overline{\alpha}.\Sigma'$ in the matching rule is explicit, then the instantiation of each α can be found by a simple pre-pass on Σ and Σ', thanks to the following observation: if the subsequent subtyping check is ever going to succeed, then Σ must feature an atomic type signature $[= \tau : \kappa_\alpha]$ at the same location where α is rooted in Σ'. Moreover, α must be instantiated with a type equivalent to τ.

Consequently, the lookup function defined in Figure 15 implements a suitable algorithm, through a straightforward parallel traversal of the two signatures. One slight twist is that an abstract type variable actually may have multiple roots in a signature. For example, the external signature {**type** t; **type** u = t} elaborates to $\exists\alpha.\{t : [= \alpha : \Omega], u : [= \alpha : \Omega]\}$. Intuitively, it does not matter which one we pick, so the algorithm simply chooses the "first" one. To this end, we impose some fixed but arbitrary total ordering on labels (solely for the purpose of the lookup algorithm) and choose the first root that we find in an ordered, depth-first traversal of the signature. Note that we never need to lookup inside a functor signature (this would change were one to add applicative functors).

Our definition of lookup is a suitably sound and complete algorithm for finding instantiations:

Theorem 3 (Soundness of Type Lookup)
Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Sigma' : \Omega$. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') = \overline{\tau}$, then $\overline{\Gamma \vdash \tau : \kappa_\alpha}$.

Theorem 4 (Completeness of Type Lookup)
Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma \vdash \exists\overline{\alpha}.\Sigma' : \Omega$, with Σ valid and $\exists\overline{\alpha}.\Sigma'$ explicit. If there exists $\overline{\tau}$ such that $\Gamma \vdash \Sigma \leq \Sigma'[\overline{\tau}/\overline{\alpha}]$, then $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') = \overline{\tau}'$ with $\overline{\tau \equiv \tau'}$.

Proof (sketch): By induction on the structure of Σ'. Since $\exists\overline{\alpha}.\Sigma'$ is explicit, and thus $\overline{\alpha}$ rooted in Σ', and since also $\Sigma \leq \Sigma[\overline{\tau}/\overline{\alpha}]$, it is clear from the definitions that $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') = \overline{\tau}'$ for some $\overline{\tau}'$. However, the choice may be different from $\overline{\tau}$. Assume α is a variable for which the choices differ, i.e., $\tau \neq \tau'$. Then the derivation $\Gamma \vdash \Sigma \leq \Sigma'[\overline{\tau}/\overline{\alpha}]$ will necessarily contain a subderivation $\Gamma \vdash [= \tau' : \kappa] \leq [= \tau : \kappa]$ for the location used in the lookup. By inversion, $\tau' \equiv \tau$. ■

Corollary 5 (Decidability of Matching)
Assume that $\Gamma \vdash \tau \leq \tau' \rightsquigarrow f$ is decidable. If Σ valid and Ξ explicit, then $\Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \rightsquigarrow f$ is decidable.

Rootedness $\qquad \dfrac{\alpha \equiv \tau}{\alpha \text{ rooted in } [= \tau : \kappa]} \qquad \dfrac{\alpha \text{ rooted in } \Sigma}{\alpha \text{ rooted in } \{l : \Sigma, \overline{l' : \Sigma'}\}} \qquad \boxed{\alpha \textbf{ rooted in } \Sigma}$

Explict Signatures $\qquad \boxed{\Xi \textbf{ explicit}}$

$$\frac{}{[\tau] \text{ explicit}} \qquad \frac{}{[= \tau : \kappa] \text{ explicit}} \qquad \frac{\Xi \text{ explicit}}{[= \Xi] \text{ explicit}} \qquad \frac{\overline{\Sigma \text{ explicit}}}{\{\overline{l : \Sigma}\} \text{ explicit}} \qquad \frac{\exists\overline{\alpha}.\Sigma \text{ explicit} \quad \Xi \text{ explicit}}{\forall\overline{\alpha}.\Sigma \to \Xi \text{ explicit}} \qquad \frac{\overline{\alpha \text{ rooted in } \Sigma} \quad \Sigma \text{ explicit}}{\exists\overline{\alpha}.\Sigma \text{ explicit}}$$

Valid Signatures and Environments $\qquad \boxed{\Xi \textbf{ valid}}\ \boxed{\Gamma \textbf{ valid}}$

$$\frac{}{[\tau] \text{ valid}} \qquad \frac{}{[= \tau : \kappa] \text{ valid}} \qquad \frac{\Xi \text{ explicit}}{[= \Xi] \text{ valid}} \qquad \frac{\overline{\Sigma \text{ valid}}}{\{\overline{l : \Sigma}\} \text{ valid}} \qquad \frac{\exists\overline{\alpha}.\Sigma \text{ explicit} \quad \Xi \text{ valid}}{\forall\overline{\alpha}.\Sigma \to \Xi \text{ valid}} \qquad \frac{\Sigma \text{ valid}}{\exists\overline{\alpha}.\Sigma \text{ valid}} \qquad \frac{\forall(X{:}\Sigma) \in \Gamma,\ \Sigma \text{ valid}}{\Gamma \text{ valid}}$$

Figure 14. Signature explicitness and validity

$$\begin{array}{lcll}
\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') & = & \overline{\tau} & \text{if } \text{lookup}_{\alpha}(\Sigma, \Sigma') = \tau \text{ for each } \alpha, \tau \in \overline{\alpha}, \overline{\tau} \\
\text{lookup}_{\alpha}([= \tau : \kappa], [= \tau' : \kappa]) & = & \tau & \text{if } \tau' \equiv \alpha \\
\text{lookup}_{\alpha}(\{\overline{l_1 : \Sigma_1}\}, \{\overline{l_2 : \Sigma_2}\}) & = & \tau & \text{if } \text{lookup}_{\alpha}(\{\overline{l_1 : \Sigma_1}\}.l, \{\overline{l_2 : \Sigma_2}\}.l) = \tau \text{ (for the smallest possible } l\text{)}
\end{array}$$

Figure 15. Algorithmic type lookup

Corollary 6 (Decidability of Elaboration)
Under valid Γ, provided we can (simultaneously) show that core elaboration is decidable, then all judgments of module elaboration are decidable too.

5.3 Declarative Properties of Signature Matching

Finally, we want to show that signature matching has the declarative properties that you would expect from a subtype relation, namely it is a preorder. This is not actually relevant for soundness or decidability, but provides a sanity check that the language we are defining actually makes sense. It is also relevant to our translation of modules as first-class values in the next section.

One complication in stating the following properties is that subtyping is defined in terms of the core language subtyping judgment $\Gamma \vdash \tau \leq \tau' \rightsquigarrow e$. Most of the properties only hold if we assume that the respective property has already been shown for that judgment. To avoid clumsy repetition, we leave this assumption implicit in the theorem statements (similarly in Theorem 11 in Section 6).

First, we need a technical lemma stating that subtyping is stable under substitution:

Lemma 7 (Subtyping under Substitution)
Let $\overline{\Gamma \vdash \tau : \kappa_\alpha}$. If $\Gamma, \overline{\alpha} \vdash \Xi \leq \Xi'$, then $\Gamma \vdash \Xi[\overline{\tau}/\overline{\alpha}] \leq \Xi'[\overline{\tau}/\overline{\alpha}]$. Moreover, the derivations have the same size, up to core language judgments.

Now for the actual theorems:

Theorem 8 (Reflexivity of Subtyping)
If $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$ and $\Xi \equiv \Xi'$, then $\Gamma \vdash \Xi \leq \Xi'$.

Theorem 9 (Transitivity of Subtyping)
If $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$ and $\Gamma \vdash \Xi'' : \Omega$ and $\Gamma \vdash \Xi \leq \Xi'$ and $\Gamma \vdash \Xi' \leq \Xi''$, then $\Gamma \vdash \Xi \leq \Xi''$.

6. Modules as First-Class Values

ML modules exhibit a strict stratification between module and core language, turning modules into second-class entities. Consequently, the kinds of computations that are possible on the module level are quite restricted. Extending modules computation to make modules first-class leads to undecidable typechecking [28]. However, it is straightforward to allow modules to be used *as first-class core values* after explicit injection into a core type of *packaged* modules [38]. In fact, in our setting, the extension is almost trivial.

Syntax Figure 16 summarizes the syntax added to the external language. We add *package types* of the form **pack** S to the core language. These are inhabited by packaged modules of signature S. Correspondingly, there is a core language expression form **pack** $M{:}S$ that produces values of this type. To unpack such a module, the inverse form **unpack** $E{:}S$ is introduced as an additional *module* expression. It expects E to be a package of type **pack** S and extracts the constituent module of signature S. (This is more liberal than the closed-scope open *expression* of [38].)

Why all the signature annotations? To avoid running into the same problems as caused by first-class modules, we do not assume any form of subtyping on package types (even if the core language had subtyping). That is, package types are only compatible if they consist of equivalent signatures. The type annotation for **pack** ensures that packaged modules still have principal types under these circumstances, so that core type checking is not compromised. For **unpack**, the annotation determines the type of E — which is necessary if we want to support ML-style type *inference* in the core language (but could be omitted otherwise).

Elaboration Figure 17 gives the corresponding elaboration rules. Let us ignore the use of *signature normalization* $\text{norm}(\Xi)$ in these rules for a minute and think of it as the identity function (which, morally, it is). Then a module M and its packaged version have essentially the same F_ω representation as a value of existential type. Consequently, elaboration becomes almost trivial. A package type simply elaborates to the very existential type that represents the constituent signature. Packing has to check that the module's signature actually matches the annotation and coerce it accordingly. Unpacking is a real no-op: there is no subtyping on package types, so the type of E has to coincide *exactly* with the annotated signature. No coercion is necessary. Proving soundness of these rules is straightforward given Lemma 10 below.

Signature Normalization So what is the business with normalization? Unfortunately, typing of packaged modules would be overly restrictive if we just used signature representations immediately to represent the corresponding package type. Consider the following example:

```
signature A = {type t; type u}
signature B = {type u; type t}
val f = fun p : (pack A) ⇒ . . .
val g = fun p : (pack B) ⇒ f p
```

(types)	T	$::=$	$\ldots \mid \textsf{\textbf{pack}}\ S$
(expressions)	E	$::=$	$\ldots \mid \textsf{\textbf{pack}}\ M{:}S$
(modules)	M	$::=$	$\ldots \mid \textsf{\textbf{unpack}}\ E{:}S$

Figure 16. Extension with modules as first-class values

Types $\boxed{\Gamma \vdash T : \kappa \rightsquigarrow \tau}$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi}{\Gamma \vdash \textsf{\textbf{pack}}\ S : \Omega \rightsquigarrow \mathrm{norm}(\Xi)}$$

Expressions $\boxed{\Gamma \vdash E : \tau \rightsquigarrow e}$

$$\frac{\Gamma \vdash M : \Xi' \rightsquigarrow e \qquad \Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash \Xi' \leq \mathrm{norm}(\Xi) \rightsquigarrow f}{\Gamma \vdash \textsf{\textbf{pack}}\ M{:}S : \mathrm{norm}(\Xi) \rightsquigarrow f\ e}$$

Modules $\boxed{\Gamma \vdash M : \Xi \rightsquigarrow e}$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash E : \mathrm{norm}(\Xi) \rightsquigarrow e}{\Gamma \vdash \textsf{\textbf{unpack}}\ E{:}S : \mathrm{norm}(\Xi) \rightsquigarrow e}$$

Figure 17. Elaboration of modules as first-class values

Intuitively, the signatures A and B are equivalent, and in fact, their semantic representations are in mutual subtyping relation. But these representations will not actually be equivalent System F$_\omega$ types—A elaborates to $\exists\alpha_1\alpha_2.\{\mathsf{t} : [= \alpha_1 : \Omega], \mathsf{u} : [= \alpha_2 : \Omega]\}$ and B to $\exists\alpha_2\alpha_1.\{\mathsf{t} : [= \alpha_1 : \Omega], \mathsf{u} : [= \alpha_2 : \Omega]\}$ according to our rules (cf. Figure 8). In the module language this is no problem: whenever we have to check a signature against another we are using coercive matching, which is oblivious to the internal ordering of quantifiers. But in the core language no signature matching is performed; package types really have to be equivalent F$_\omega$ types in order to be compatible. In that case, the order matters. So the definition of g above would not type check.

To compensate, our elaboration must ensure that two package types pack S_1 and pack S_2 translate to equivalent F$_\omega$ types whenever S_1 and S_2 are mutual subtypes. Toward this end, we employ the normalization function defined in Figure 18. All this function does is put the quantifiers of a semantic signature into a canonical order. For example, for a signature $\exists\overline{\alpha}.\Sigma$, normalization will sort the variables $\overline{\alpha}$ according to their (first) appearance as a root in a left-to-right depth-first traversal of Σ. As in Section 5.2, we assume a total ordering on the set of labels to make this well-defined. Note that we only need to normalize the representations of signatures appearing as annotations, so normalization is defined only for explicit signatures (Section 5.2), where every variable is rooted.

In the base case of atomic value signatures $[\tau]$, we assume that a similar normalization function $\mathrm{norm}_{\mathrm{core}}(\tau)$ exists for normalizing core-level types according to core-level subtyping $\Gamma \vdash \tau \leq \tau'$. (For instance, for ML this core type normalization would canonicalize the order of quantified type variables in polymorphic types.)

It is not difficult to show the following properties:

Lemma 10 (Signature Normalization)

1. If Ξ explicit, then $\mathrm{norm}(\Xi)$ explicit.
2. If $\Gamma \vdash \Xi : \Omega$, then $\Gamma \vdash \mathrm{norm}(\Xi) : \Omega$.
3. If Ξ explicit, then $\Gamma \vdash \Xi \leq \mathrm{norm}(\Xi)$ and $\Gamma \vdash \mathrm{norm}(\Xi) \leq \Xi$.

The main result then is a form of anti-symmetry for subtyping:

Theorem 11 (Antisymmetry of Subtyping up to Normalization)

Let $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$, and Ξ, Ξ' explicit. If $\Gamma \vdash \Xi \leq \Xi'$ and $\Gamma \vdash \Xi' \leq \Xi$, then $\mathrm{norm}(\Xi) \equiv \mathrm{norm}(\Xi')$.

By normalizing semantic signatures in all places where they are used as package types, we hence establish the desired property

$$\begin{array}{lcl}
\mathrm{norm}([\tau]) & = & [\mathrm{norm}_{\mathrm{core}}(\tau)] \\
\mathrm{norm}([= \tau : \kappa]) & = & [= \tau : \kappa] \\
\mathrm{norm}([= \Xi]) & = & [= \mathrm{norm}(\Xi)] \\
\mathrm{norm}(\{\overline{l : \Sigma}\}) & = & \{\overline{l : \mathrm{norm}(\Sigma)}\} \\
\mathrm{norm}(\forall\overline{\alpha}.\Sigma \to \Xi) & = & \forall\overline{\alpha}'.\,\mathrm{norm}(\Sigma) \to \mathrm{norm}(\Xi) \\
 & & \text{where } \overline{\alpha}' = \mathrm{dfv}(\overline{\alpha}, \mathrm{norm}(\Sigma)) \\
\mathrm{norm}(\exists\overline{\alpha}.\Sigma) & = & \exists\overline{\alpha}'.\,\mathrm{norm}(\Sigma) \\
 & & \text{where } \overline{\alpha}' = \mathrm{dfv}(\overline{\alpha}, \mathrm{norm}(\Sigma)) \\
\mathrm{dfv}(\overline{\alpha}, [\tau]) & = & \epsilon \\
\mathrm{dfv}(\overline{\alpha}, [= \tau : \kappa]) & = & \alpha \ \text{ if } \tau \equiv \alpha \text{ for some } \alpha \in \overline{\alpha} \\
\mathrm{dfv}(\overline{\alpha}, [= \tau : \kappa]) & = & \epsilon \ \text{ otherwise} \\
\mathrm{dfv}(\overline{\alpha}, [= \Xi]) & = & \epsilon \\
\mathrm{dfv}(\overline{\alpha}, \{\}) & = & \epsilon \\
\mathrm{dfv}(\overline{\alpha}, \{l_1 : \Sigma_1, \overline{l_2 : \Sigma_2}\}) & = & \overline{\alpha}_1, \mathrm{dfv}(\overline{\alpha} - \overline{\alpha}_1, \{\overline{l_2 : \Sigma_2}\}) \\
 & & \text{where } \overline{\alpha}_1 = \mathrm{dfv}(\overline{\alpha}, \Sigma_1), l_1\overline{l}_2 \text{ sorted} \\
\mathrm{dfv}(\overline{\alpha}, \forall\overline{\alpha}'.\Sigma \to \Xi) & = & \epsilon
\end{array}$$

Figure 18. Signature normalization

that type equivalence coincides with signature equivalence. By applying the coercion f in the rule for **pack**, we also ensure that the representation of the module itself is normalized accordingly.

7. Mechanization in Coq

Although our elaboration semantics is small, it is still large and informal enough to contain errors, so we embarked on mechanizing it in Coq [4] using the locally nameless approach (LN) of Aydemir *et al.* [1]. (There is no reason we could not have used other proof assistants such as Twelf or Isabelle; we were just interested in learning Coq and testing the effectiveness of the locally nameless approach.) We have mechanized the elaboration semantics of Section 4 and Section 6 (but omitting normalization) and proved the soundness result of Theorem 1. This effort required roughly 13,000 lines of Coq code. As inexpert users of Coq, we made little use of automation, so probably the proofs could easily be shortened.

As with any mechanization, there are some minor differences compared with the informal system. Our mechanized F$_\omega$ is simpler than the one we use here in that it supports just binary products, not records. Instead, we encode ordered records as derived forms using pairs, with derived typing rules, and target those during elaboration. Ordered records are easier to mechanize, yet adequate for elaboration.

The F$_\omega$ mechanization does not allow rebindings of term variables in the context as our informal presentation does. Indeed, using the LN approach, subderivations arising from binding constructs have to hold for all locally fresh names. In the mechanization, we had to abandon the use of the injection from source identifiers to F$_\omega$ variables, and instead use a translation environment that twins source identifiers (which may be shadowed) with locally fresh F$_\omega$ variables (which may not). In this way, source identifiers are used to determine record labels, while their twinned variables are used to translate free occurrences of identifiers. Lee *et al.* [24] use a similar trick in their Twelf mechanization of Standard ML.

Our use of a non-injective record encoding means that different semantic signatures may be encoded by the same type. To avoid ambiguity, the mechanization therefore introduces a special syntactic class of semantic signatures (corresponding to the grammar in Figure 7), and separately defines the interpretation of semantic signatures as System F$_\omega$ types by an inductive definition (again much like the syntactic sugar definitions in Figure 7). Consequently, the mechanized soundness theorems state that if $C \vdash M : \Xi \rightsquigarrow e$, then $C^\circ \vdash e : \Xi^\circ$, where $\_^\circ$ denotes the interpretation of elaboration environments and semantic signatures into plain F$_\omega$ contexts and types. In retrospect, it would perhaps have been simpler to just beef up our target language with primitive records (as we have done on paper here). In any case, this issue is orthogonal to the rest of the mechanization effort.

Our experience of applying the LN approach as advertised was more painful than we had anticipated. Compared to the sample LN developments, ours was different in making use of various forms of derived n-ary (as well as basic unary binders) and in dealing with a larger number of syntactic categories. Out of a total of around 550 lemmas, approximately 400 were tedious "infrastructure" lemmas; only the remainder had direct relevance to the metatheory of F_ω or elaboration. The number of required infrastructure lemmas appears to be quadratic in the number of variable classes (type and value variables for us), the number of "substitution" operations needed per class (we got away with only using LN's `subst` and `open`, and avoiding `close`) and the arity (unary and n-ary) of binding constructs. So we cannot, hand-on-heart, recommend the vanilla LN style for anything but small, kernel language developments. Recent proposals to streamline the LN approach [2] may help.

8. Related Work and Discussion

The literature on ML module semantics is voluminous and varied. We will therefore focus on the most closely related work.

Existential Types for ADTs Mitchell and Plotkin [32] were the first to connect the informal notion of "abstract type" to the existential types of System F. In F, values of existential type are first-class, in the sense that the construction of an ADT may depend on *run-time* information. We exploit this observation in our support for modules as first-class values (Section 6), which are simply existential packages.

Dependent Type Systems for Modules In a very influential position paper, MacQueen [29] criticized existential types as a basis for modular programming, arguing that the closed-scope elimination construct for existentials (`unpack`) is too weak and awkward to be usable in practice. MacQueen instead promoted the use of dependent function types and "strong sums" (*i.e.,* dependently-typed record/tuple types) as a basis for modular programming. Since then, there has been a long line of work on understanding and evolving the ML module system in terms of increasingly more refined dependent type theories [17, 18, 16, 26, 25, 41, 9].

On the design side, the work on dependent type systems led to significant improvements in the expressiveness of ML modules, most notably the idea of *translucency*—*i.e.,* the ability to include both abstract and transparent type declarations in signatures—which was independently proposed by Harper and Lillibridge [16] and Leroy [26]. On the semantics side, however, the use of dependent type formalisms unleashed quite a can of worms. Several ideas and issues pop up again and again in the literature, and for the most part the "F-ing modules" approach either renders these issues moot or offers straightforward ways of handling them.

One recurrent notion is *phase separation*, which is essentially the observation that the "dependent" types in these module systems are not *really* dependent. The signature of a module may depend on the *type* components of another module, but not on its *value* components. Thus, as Harper, Mitchell, and Moggi [18] showed (for an early ML-style module system without translucency or sealing), one can "phase-split" a (higher-order) module into an F_ω type (representing its type components) and an F_ω expression (representing its value components). Our approach of interpreting ML modules into F_ω is of course completely compatible with the idea of phase separation, since we don't pretend our type system is dependent in the first place.

Another recurrent notion is *projectibility*—that is, from which module expressions can one project out the type and value components? As Dreyer, Crary, and Harper [9] observed, the differences between several different dialects of the ML module system can be characterized by how they define projectibility. Most dependent module type systems define projectibility by only allowing projections from modules from a certain restricted *syntactic* class of *paths*. We also employ paths, but define them *semantically* to be any module expressions whose signatures do not mention any "local" (*i.e.,* existentially-quantified) abstract types. We consider this criterion to be simpler to understand and less *ad hoc*.

A common stumbling block in dependent module type systems is the so-called *avoidance problem*. Originally observed in the setting of (a bounded existential extension of) System $F_\leq$ by Ghelli and Pierce [13], the avoidance problem is roughly that a module might not have a *principal* signature (*i.e.,* minimal in the subtyping hierarchy) that "avoids" (*i.e.,* does not depend on) some local abstract type. As principal signatures are important for practical typechecking, dependent module type systems typically either lack complete typechecking algorithms (*e.g.,* [28, 27]) or else require (at least in some cases) extra signature annotations when leaving the scope of an abstract type (*e.g.,* [41, 9]). In contrast, under our approach the avoidance problem does not arise at all: the semantic signature $\exists\overline{\alpha}.\Sigma$ of a module M keeps track of *all* the abstract types $\overline{\alpha}$ defined by M, even those which have "gone out of scope" in the sense that they are not "rooted" anywhere in Σ (to use the terminology of Section 5). Thus, the only point at which we need to "avoid" anything is when we typecheck a *path* and need to make sure that its signature does not depend on any local abstract types. Of course, at that point the avoidance check is not a "problem" but rather the crucial defining element of well-formedness for paths.

Elaboration Semantics for Modules Our avoidance of the avoidance problem is due primarily to our use of an elaboration semantics, which gives us the flexibility to classify a module using a semantic signature Ξ that is not the translation of any syntactic signature S. Harper and Stone [20] exploit elaboration in a similar fashion and to similar ends. One downside of this approach, some would argue [41], is that one loses "fully syntactic" signatures, but it is not clear that in practice this is such a big deal.

Perhaps a more serious concern is: how does the elaboration semantics we have given here correspond to existing specifications of ML modules, such as the Definition of SML or Harper-Stone? In what sense are we formalizing the semantics of "ML modules"? The short answer is that it is very difficult to prove a precise correspondence between different accounts of the ML module system. In the few cases where such proofs have been attempted, the formalizations in question were either not representative of the full ML module system (*e.g.,* [26]) or were lacking some key component, such as a dynamic semantics (*e.g.,* [37]). Moreover, one of the main advantages of our approach (we believe) is that it is simpler than previous approaches. We are not so interested in "correctness", *i.e.,* whether our semantics precisely matches that of Standard ML, the archaeological artifact; rather, we wish to suggest a way forward in the understanding and evolution of ML-style module systems. That said, we believe (based on experience) that our semantics for modules is essentially a conservative extension of SML's, capturing the generative fragment of Moscow ML [39].

Higher-Order Modules and Applicative Functors The main way in which we diverge from SML is that we support higher-order modules. Our semantics for higher-order modules is similar to that of Leroy [26] and Harper-Lillibridge [16]. As in those systems, all functors in our language behave generatively, thus causing the signatures of some higher-order functors to be more abstract than is arguably desirable. MacQueen and Tofte [30] proposed a more flexible semantics for functors, but it relies conceptually on the idea of re-elaborating a functor's body at each application. Leroy [25], Shao [41], and others have proposed *applicative* functors as a more type-theoretic way of supporting "fully transparent" higher-order modules. As Dreyer *et al.* [9] point out, however, applicative functors are not a replacement for generative functors, both varieties being useful in different circumstances. We will show how to support applicative functors, F-ing style, in a future, expanded version of this paper.

Interpreting ML Modules into F_ω We are certainly not the first to explain ML modules by translation into F_ω. Harper, Mitchell, and Moggi [18] give a "phase-splitting" translation of an early ML module calculus into F_ω. Shao [41] gives a multi-stage translation of his module calculus into F_ω. Shan [40] presents a type-directed translation of the Dreyer-Crary-Harper calculus [9] into F_ω.

The difference between these previous translations and ours is that the previous ones all start from a pre-existing dependently-typed module language and show how to compile it down to F_ω. We instead use the type structure of F_ω in order to give a static semantics for ML modules directly. Thus, we feel our approach is simpler and more accessible to someone who already understands F_ω and does not want to learn a new dependent type system just in order to understand the semantics of ML modules.

As explained in the introduction, our approach can be viewed as giving an evidence translation, and thus a soundness proof, for (a variant of) the static semantics of SML modules given in Russo's thesis [37, 36]. Russo started with the Definition of Standard ML [31], and observed that its *ad hoc* "semantic object" language could be understood quite clearly in terms of universal and existential types. A key observation, also made by Elsman [12], was that the state of generated type variables, threaded monadically as it was through the static semantics of SML, could be presented more declaratively as the systematic introduction and elimination of existential types. Given the non-dependent, F_ω-like structure of the semantic objects, it was also relatively straightforward to extend them to higher-order and first-class modules [37, 38].

Our approach also scales to handle more ambitious module-language extensions, at least if one is willing to beef up the target language somewhat. Inspired by Russo's work, Dreyer proposed an extension of F_ω called RTG [7], which he and coauthors later used as the target of an elaboration semantics for recursive modules [5], mixin modules [11], and modules in the presence of type inference [8]. These elaboration semantics are similar to ours in that they use the type structure of the (beefed-up) F_ω language in order to directly encode semantic signatures for ML-style modules. However, our semantics is significantly simpler, since we are only trying to formalize an SML-like module system and we are only using vanilla F_ω as the target language.

Mechanization of Module Semantics Lee *et al.* [24] mechanized the metatheory of full Standard ML, based on a variant of Harper-Stone elaboration given by Dreyer in his thesis [6]. It is difficult to compare the mechanizations, since theirs uses Twelf. However, it is worth noting that a significant piece of their mechanization is devoted to proving metatheoretic properties of their target language, which employs singleton kinds [43]. In contrast, since our internal language is so simple and well-studied, we largely took it for granted (though we have proved the F_ω properties that we use).

Direct Modular Programming in F_ω Lastly, several authors have advocated doing modular programming directly in a rich F_ω-like core language like Haskell's [22, 42, 40], using universal types for client-side data abstraction and existential types for implementor-side data abstraction. Several other authors [29, 19] have argued why this approach is not practical. The common theme of the arguments is that F_ω is too low-level a language to program modules in directly, and that ML modules provide a much higher-level idiom for modular programming. More recently, Montagu and Rémy [33] have proposed directly programming in a variant of Dreyer's RTG [7] (see above), because RTG addresses to some extent the limitations of closed-scope existential elimination. However, RTG is still quite low-level compared to ML modules.

In some sense, the point of this paper is to observe that the high-level elegance of ML modules and the simplicity of F_ω typing are not mutually exclusive. One can understand ML modules precisely as a stylized idiom—a design pattern, if you will—for constructing F_ω programs. The key benefit of programming this idiom using the ML module system, instead of directly in F_ω, is that elaboration offers a significant degree of automation (*e.g.,* by inferring signature coercions and implicitly unpacking/repacking existentials), which in practice is extremely useful.

9. Conclusion

Our contribution is a dead simple, type-theoretic semantics for a representative ML module system. The language defined here is essentially a generalization of Standard ML modules with higher-order functors and first-class packages. We have shown not only how to typecheck this language, but also how to compile it, by translation into a vanilla, off-the-shelf target language F_ω. Essentially, the translation does little more than inserting introduction and elimination forms for existential and universal quantifiers in the appropriate places. The semantics is so elementary, it could be mechanized by novice users of Coq using textbook meta-theory. In an expanded version of this paper, we will report on how to extend our elaboration semantics, *without changing the target language of F_ω*, in order to account for OCaml-style applicative functors.

References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08*.

[2] B. Aydemir, S. Weirich, and S. Zdancewic. Abstracting syntax. Technical Report MS-CIS-09-06, U. Penn, 2009.

[3] S. K. Biswas. Higher-order functors with transparent signatures. In *POPL '95*.

[4] Coq Development Team. *The Coq proof assistant reference manual, v. 8.1.* INRIA, 2007. `http://coq.inria.fr/`.

[5] D. Dreyer. A type system for recursive modules. In *ICFP '07*.

[6] D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, CMU, 2005.

[7] D. Dreyer. Recursive type generativity. *JFP*, 17(4&5):433–471, 2007.

[8] D. Dreyer and M. Blume. Principal type schemes for modular programs. In *ESOP '07*.

[9] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL '03*.

[10] D. Dreyer, K. Crary, and R. Harper. Moscow ML's higher-order modules are unsound, 17 September 2002. (Types Forum).

[11] D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ICFP '08*.

[12] M. Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation.* PhD thesis, U. of Copenhagen, 1999.

[13] G. Ghelli and B. Pierce. Bounded existentials and minimal typing. *TCS*, 193(1-2):75–96, 1998.

[14] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[15] R. Harper. Programming in Standard ML. Working draft available at: `http://www.cs.cmu.edu/~rwh/smlbook/`.

[16] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.

[17] R. Harper and J. C. Mitchell. On the type structure of Standard ML. In *TOPLAS*, volume 15(2), pages 211–252, 1993.

[18] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *POPL '90*.

[19] R. Harper and B. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8. MIT Press, 2005.

[20] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[21] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3), 2001.

[22] M. P. Jones. Using parameterized signatures to express modular structure. In *POPL '96*.

[23] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[24] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07*.

[25] X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*.

[26] X. Leroy. A syntactic theory of type generativity and sharing. *JFP*, 6(5):1–32, September 1996.

[27] X. Leroy. A modular module system. *JFP*, 10(3):269–303, 2000.

[28] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, CMU, 1997.

[29] D. MacQueen. Using dependent types to express modular structure. In *POPL '86*.

[30] D. MacQueen and M. Tofte. A semantics for higher-order functors. In *ESOP '94*.

[31] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[32] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3):470–502, July 1988.

[33] B. Montagu and D. Rémy. Modeling abstract types in modules with open existential types. In *POPL '09*.

[34] L. C. Paulson. *ML for the Working Programmer, 2nd Edition*. Cambridge University Press, 1996.

[35] S. Peyton Jones. Wearing the hair shirt: a retrospective on Haskell. Invited talk, *POPL '03*.

[36] C. V. Russo. Non-dependent types for Standard ML modules. In *PPDP '99*.

[37] C. V. Russo. *Types For Modules*. PhD thesis, LFCS, University of Edinburgh, 1998.

[38] C. V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, November 2000.

[39] C. V. Russo. Types for Modules. *ENTCS*, 60, 2003. Chapter 10.

[40] C. Shan. Higher-order modules in System F_ω and Haskell, 2004. `http://www.cs.rutgers.edu/~ccshan/xlate/xlate.pdf`.

[41] Z. Shao. Transparent modules with fully syntactic signatures. In *ICFP '99*.

[42] M. Shields and S. Peyton Jones. First-class modules for Haskell. In *FOOL 9*, 2002.

[43] C. A. Stone and R. Harper. Extensional equivalence and singleton types. *TOCL*, 7(4):676–722, 2006.

[44] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07*.

[45] M. Torgersen, E. Ernst, and C. P. Hanser. Wild FJ. In *FOOL 12*, 2005.

Proof Script

The interested reader can find our Coq [4] proof script at:

`http://www.mpi-sws.org/~rossberg/f-ing/`

A. Semantics of System $\mathbf{F}_\omega$

A.1 Static Semantics

Environments $\boxed{\Gamma \vdash \square}$

$$\frac{}{\cdot \vdash \square} \qquad \frac{\Gamma \vdash \square \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma, \alpha{:}\kappa \vdash \square} \qquad \frac{\Gamma \vdash \tau : \Omega}{\Gamma, x{:}\tau \vdash \square}$$

Types $\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad \frac{\overline{\Gamma \vdash \tau : \Omega}}{\Gamma \vdash \{\overline{l{:}\tau}\} : \Omega}$$

$$\frac{\Gamma \vdash \square}{\Gamma \vdash \alpha : \Gamma(\alpha)} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau : \Omega} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau : \Omega}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda \alpha{:}\kappa.\tau : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

Terms $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \square}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\overline{\Gamma \vdash e : \tau}}{\Gamma \vdash \{\overline{l{=}e}\} : \{\overline{l{:}\tau}\}} \qquad \frac{\Gamma \vdash e : \{l{:}\tau, \overline{l'{:}\tau'}\}}{\Gamma \vdash e.l : \tau}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\tau} \qquad \frac{\Gamma \vdash e : \forall \alpha{:}\kappa.\tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\ \tau : \tau'[\tau/\alpha]}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \tau'[\tau/\alpha] \qquad \Gamma \vdash \exists \alpha{:}\kappa.\tau' : \Omega}{\Gamma \vdash \mathsf{pack}\ \langle \tau, e \rangle_{\exists \alpha:\kappa.\tau'} : \exists \alpha{:}\kappa.\tau'}$$

$$\frac{\Gamma \vdash e_1 : \exists \alpha{:}\kappa.\tau' \qquad \Gamma, \alpha{:}\kappa, x{:}\tau' \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathsf{unpack}\ \langle \alpha, x \rangle {=} e_1\ \mathsf{in}\ e_2 : \tau}$$

Type Equivalence $\boxed{\tau \equiv \tau'}$

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau' \equiv \tau}{\tau \equiv \tau'} \qquad \frac{\tau \equiv \tau' \qquad \tau' \equiv \tau''}{\tau \equiv \tau''}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\overline{\tau \equiv \tau'}}{\{\overline{l{:}\tau}\} \equiv \{\overline{l{:}\tau'}\}}$$

$$\frac{\tau \equiv \tau'}{\forall \alpha{:}\kappa.\tau \equiv \forall \alpha{:}\kappa.\tau'} \qquad \frac{\tau \equiv \tau'}{\exists \alpha{:}\kappa.\tau \equiv \exists \alpha{:}\kappa.\tau'}$$

$$\frac{\tau \equiv \tau'}{\lambda \alpha{:}\kappa.\tau \equiv \lambda \alpha{:}\kappa.\tau'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}$$

$$\frac{}{(\lambda \alpha{:}\kappa.\tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]} \qquad \frac{\alpha \notin \mathrm{fv}(\tau)}{(\lambda \alpha{:}\kappa.\tau\ \alpha) \equiv \tau}$$

A.2 Dynamic Semantics

Reduction $\boxed{e \hookrightarrow e'}$

$$\begin{array}{rcl}
(\lambda x{:}\tau.e)\ v & \hookrightarrow & e[v/x] \\
\{\overline{l_1{=}v_1}, l{=}v, \overline{l_2{=}v_2}\}.l & \hookrightarrow & v \\
(\lambda \alpha{:}\kappa.e)\ \tau & \hookrightarrow & e[\tau/\alpha] \\
\mathsf{unpack}\ \langle \alpha, x \rangle = \mathsf{pack}\ \langle \tau, v \rangle_{\tau'}\ \mathsf{in}\ e & \hookrightarrow & e[\tau/\alpha][v/x]
\end{array}$$

$$\frac{e \hookrightarrow e'}{C[e] \hookrightarrow C[e']}$$

where:

$$\begin{array}{rcl}
C & ::= & [] \mid C\ e \mid v\ C \mid \{\overline{l_1{=}v}, l{=}C, \overline{l_2{=}e}\} \mid C.l \mid \\
 & & C\ \tau \mid \mathsf{pack}\ \langle \tau, C \rangle_\tau \mid \mathsf{unpack}\ \langle \alpha, x \rangle {=} C\ \mathsf{in}\ e
\end{array}$$