JFP **24** (5): 529–607, 2015. © Cambridge University Press 2014
doi:10.1017/S0956796814000264 First published online 19 November 2014

F-ing modules

ANDREAS ROSSBERG
Google, Dienerstr. 12, 80331 Munich, Germany
(*e-mail:* rossberg@mpi-sws.org)

CLAUDIO RUSSO
Microsoft Research 21 Station Road, Cambridge CB1 2FB, United Kingdom
(*e-mail:* crusso@microsoft.com)

DEREK DREYER
Max Planck Institute for Software Systems (MPI-SWS), 66123 Saarbrücken, Germany
(*e-mail:* 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 present a novel formalization of ML modules, which defines their semantics directly by a compositional "elaboration" translation into plain System F_ω (the higher-order polymorphic λ-calculus). To demonstrate the scalability of our "F-ing" semantics, we use it to define a representative, higher-order ML-style module language, encompassing all the major features of existing ML module dialects (except for recursive modules). We thereby show that ML modules are merely a particular mode of use of System F_ω.

To streamline the exposition, we present the semantics of our module language in stages. We begin by defining a subset of the language supporting a Standard ML-like language with *second-class* modules and *generative* functors. We then extend this sublanguage with the ability to package modules as *first-class* values (a very simple extension, as it turns out) and OCaml-style *applicative* functors (somewhat harder). Unlike previous work combining both generative and applicative functors, we do not require two distinct forms of functor or signature sealing. Instead, whether a functor is applicative or not depends only on the computational purity of its body. In fact, we argue that applicative/generative is rather incidental terminology for *pure* versus *impure* functors. This approach results in a semantics that we feel is simpler and more natural than previous accounts, and moreover prohibits breaches of abstraction safety that were possible under them.

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*), implementor-side data abstraction (via *sealing*), and a flexible form of signature matching (via *structural subtyping*).

Unfortunately, while the utility of ML modules is not in dispute, they have nonetheless acquired a reputation for being "complex". Simon Peyton Jones (2003), in an oft-cited POPL keynote address, 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, or understand than other ambitious modularity mechanisms, such as GHC's *type classes* with type equality coercions (Sulzmann *et al.*, 2007) or Java's *classes* with generics and wildcards (Torgersen *et al.*, 2005)? 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 (Rossberg & Dreyer, 2013). 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 (Harper *et al.*, 1990; Harper & Lillibridge, 1994; Leroy, 1994; Leroy, 1995; Russo, 1998; Shao, 1999; Dreyer *et al.*, 2003), as well as several *ad hoc*, non-type-theoretic (MacQueen & Tofte, 1994; Biswas, 1995; Milner *et al.*, 1997) 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 meta-theory of singleton kinds (Stone & Harper, 2006) — 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 *et al.* (2003) developed a unifying type theory, in which previous systems could be understood as sublanguages that selectively include different combinations of 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 article, we take a different approach. Our goal is to show once and for all that, contrary to popular belief (even among experts in the field!), the semantics of ML modules is immediately accessible to anyone familiar with System F_ω, the higher-order polymorphic λ-calculus. How do we achieve this goal?

First, instead of defining the semantics of modules — as most prior work has done — via a bespoke module type system (Dreyer *et al.*, 2003) or a non-type-theoretic formalization (Milner *et al.*, 1997), we employ an *elaboration* semantics, in which the meaning of module expressions is defined by a compositional, syntax-directed translation into plain System F_ω. Through this elaboration, we show that

ML modules are merely a particular mode of use of System F_ω.

A *structure* is just a record of existential type $\exists\overline{\alpha}.\{\overline{l : \tau}\}$, where the type variables $\overline{\alpha}$ represent the abstract types defined in the structure. A *functor* is just a function of polymorphic type $\forall\overline{\alpha}.\tau \to \tau'$, parameterized over the abstract types $\overline{\alpha}$ in its module argument. No dependent types of any form are required. However, as is often the case for common programming idioms, it is extremely helpful to have built-in language support for inference and automation where possible. In our "F-ing" elaboration semantics, this amounts to inserting the right introduction and elimination forms for universal and existential types in the right places, *e.g.,* using "signature matching" to infer the appropriate type arguments when a functor is applied or when a structure is sealed with a signature.

Our approach thus synthesizes elements of two alternative definitions of Standard ML modules given by Harper & Stone (2000) and Russo (1998). Like Harper & Stone (2000), we define our semantics by elaboration; but whereas Harper & Stone elaborated ML modules into yet another (dependently-typed) module type system — a variant of Harper & Lillibridge (1994) — we elaborate them into F_ω, which is a significantly simpler system. Like Russo (1998), we classify ML modules — and interpret ML signatures — directly using the types of System F_ω; but whereas Russo only presented a static semantics, our elaboration effectively provides an *evidence translation* for a simplified and streamlined variant of his definition, thus equipping it with a dynamic semantics and type soundness proof.

Second, we demonstrate the broad applicability of our F-ing elaboration semantics by using it to define a richly-featured — and, we argue, representative — ML-style module language. By "representative", we mean that the language we define encompasses all the major features of existing ML module dialects except for recursive modules.[1] While other researchers have given translations from dialects of ML modules into versions of System F_ω before (Shan, 2004; Shao, 1999), we are, to our knowledge, the first to *define* the semantics of a *full-fledged* ML-style module language *directly* in terms of System F_ω. By "directly", we mean that there is no other high-level static semantics involved — F_ω types are enough to classify modules and understand their semantics.

In contrast, most previous work on modules has focused on bespoke module calculi that are (a) defined independently of F_ω and (b) somewhat idealized, relying on a separate non-trivial stage of pre-elaboration to handle certain features, and often glossing over essential aspects of a real module language, such as shadowing

[1] A proper handling of type abstraction in the presence of recursive modules seems to require both a more sophisticated underlying type theory (Dreyer, 2007a), as well as a more radical departure from the linking mechanisms of the ML module system (Rossberg & Dreyer, 2013).

between declarations, local or shadowed types (and the so-called *avoidance problem* that they induce), or composition constructs like **open**, **include** and **where**/**with**, all of which add — in some cases quite substantial — complexity.

To ease the presentation, we present the semantics of our module language in stages. In the first part of the article (Sections 2–5), we show how to typecheck and implement a subset of our language that roughly corresponds to the Standard ML module language extended with higher-order functors. This subset supports only *second-class* modules, not *first-class* modules (Harper & Lillibridge, 1994; Russo, 2000), and only SML-style *generative* functors, not OCaml-style *applicative* functors (Leroy, 1995). We start with this SML-style language because its F-ing semantics is relatively simple and direct.

In the second part of the article (Sections 6–9), we extend the language of the first part with both *modules-as-first-class-values* (Section 6, *easy*) and *applicative functors* (Sections 7–9, *harder*). For the extension to applicative functors, we have taken the opportunity to address some overly complex and/or semantically problematic aspects of previous approaches. In particular, unlike earlier unifying accounts of ML modules (Romanenko *et al.*, 2000; Dreyer *et al.*, 2003; Russo, 2003), we do not require two distinct forms of functor declaration (or two different forms of module sealing). Instead, our type system deems a functor to be applicative iff the body of the functor is *computationally pure*, and generative otherwise. We believe this is about as simple a characterization of the applicative/generative distinction as one could hope for.

That said, the semantics we give for applicative functors is definitely not as simple as the elaboration semantics for generative functors given in the first part of the article. We believe the relative complexity of our applicative functor semantics is not a weakness of our approach, but rather a reflection of the inescapable fact that the applicative semantics for functors is inherently subtler (and harder to get right!) than the generative semantics. We substantiate this claim by showing that *no previous account* of applicative functors has properly guaranteed *abstraction safety* — *i.e.,* the ability to locally establish representation invariants for abstract types.[2] To avoid this problem, we revive the long-lost notion of *structure sharing* from Standard ML'90 (Milner *et al.*, 1990), in the form of more fine-grained *value sharing*. Although previous work on module type systems has disparaged this form of sharing as type-theoretically questionable, we observe that it is in fact *necessary* in order to ensure abstraction safety in the presence of applicative functors. Furthermore, it is easy to account for in a type-theoretic manner using "phantom types" as "stamps".

[2] As further evidence of the relative complexity of applicative functors, we note that the F-ing semantics for applicative functors fundamentally *requires* F_ω's higher kinds, while the generative functor semantics presented in the first part of the article does not. Higher kinds are of course needed if the underlying core language (on top of which the module system is built) supports type constructors — as is the case in ML. However, setting the core language aside, the elaboration semantics we give in the first part of the article does not itself rely on higher-kinded type abstraction, and indeed, for a simpler core language with just type (but not type constructor) definitions, that language can be elaborated to plain System F. By contrast, the applicative functor extension presented in the second part of the article relies on higher kinds in an essential way.

In general, we have tried to give this article 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 (Harper, 2012) and Paulson's book (1996). For the latter, we refer the reader to Harper & Pierce (2005), as well as the early chapters of the second and third authors' PhD theses (Russo, 1998; Dreyer, 2005).

The F-ing approach has of course not fallen from the sky. It naturally builds on many ideas from previous work. As mentioned above, the central insight of viewing the seemingly dependent type system of ML modules through the lens of System F types is due to Russo (1998; 1999), and many of the ideas for translating module terms are already present in prior work by Harper *et al.* (1990), Harper & Stone (2000), and Dreyer (2007b). Our technical development of applicative functors is directly influenced by the work of Biswas (1995), Russo (1998; 2003), and Shan (2004), and more indirectly by Shao (1999) and Dreyer *et al.* (2003). But instead of frontloading this article with a survey of the literature, we will point to the origins of some key ideas as we come to them. A more comprehensive discussion can be found in Section 11.

To summarize our contributions, we present the first formalization of ML modules that

1. explains the static and dynamic semantics of a *full-fledged* module system, *directly* in terms of System F_ω terms, types and environments, requiring only *plain* F_ω to do so, and
2. characterizes applicativity/generativity of functors as a matter of *purity*, and supports applicative functors in a way that is *abstraction-safe*, by relying crucially on a novel account of *value sharing*.

For those familiar with an earlier version of this article that was published in the TLDI workshop (Rossberg *et al.*, 2010), we note that the major difference in the present version is contribution #2, that is, the novel account of applicative functors in Sections 7–9 (the workshop version only treated generative functors). We now also offer expanded discussions of first-class modules (Section 6), our Coq mechanization (Section 10), and related work (Section 11), as well as more details of the meta-theory (Section 5).

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. Binding constructs for types and values are provided as part

(identifiers)	X		
(kinds)	K	::=	$\ldots$
(types)	T	::=	$\ldots \mid P$
(expressions)	E	::=	$\ldots \mid P$
(paths)	P	::=	M

(modules)	M	::=	X	(signatures)	S	::=	P
		|	$\{B\} \mid M.X$			|	$\{D\}$
		|	**fun** $X{:}S \Rightarrow M \mid X\ X$			|	$(X{:}S) \to S$
		|	$X{:>}S$			|	S **where type** $\overline{X}{=}T$
(bindings)	B	::=	**val** $X{=}E$	(declarations)	D	::=	**val** $X{:}T$
		|	**type** $X{=}T$			|	**type** $X{=}T \mid$ **type** $X{:}K$
		|	**module** $X{=}M$			|	**module** $X{:}S$
		|	**signature** $X{=}S$			|	**signature** $X{=}S$
		|	**include** M			|	**include** S
		|	ϵ			|	ϵ
		|	$B;B$			|	$D;D$

Fig. 1. Syntax of the module language.

of the module language. For simplicity, we assume that all language entities share a single identifier syntax.[3]

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: bindings and declarations of values, types, modules, and signatures (where, as in SML, we implicitly allow omitting the separating ";" between the bindings/declarations in a sequence); hierarchical structures with projection via the dot notation; structure/signature inheritance with **include**; functors and functor signatures; and sealing (a.k.a. opaque signature ascription). In the grammar for the "**where type**" construct we abuse the notation $\overline{X}$ to denote an identifier followed by a (possibly empty) sequence of projections, *e.g.,* X or $X.Y.Z$.

In some cases, the syntax restricts module expressions in certain positions (*e.g.,* the components of a functor application) to be identifiers X. This is merely to make the semantics of the language that we define in Section 4 as simple as possible.

Fully general variants of these constructs are definable as straightforward derived forms, as shown in Figure 2. The same figure also defines other constructs that are available in various dialects of ML modules, such as "let"-expressions on all syntactic levels (including types and signatures), "local" bindings/declarations,[4] and parameterized signatures.[5] Using some of these derived forms, Figure 3 shows the implementation of a standard Set functor.

[3] For an ML-like core language, this is meant to include type variables ′a, and we do not impose any restrictions on where type variables from the context can appear in type and signature expressions.

[4] The module-level **include** M is spelled **open** M in Standard ML. OCaml's version of **open** M can be expressed as **local include** M **in** $\ldots$ in our system.

[5] Parameterized signatures may be less familiar to many readers, given that only a few ML dialects support them. A signature declared via **signature** A (X : B) = $\ldots$ takes a module parameter, and is instantiated with an application A M in a signature expression. Such a parameterized signature definition simply desugars to a functor definition wherein the result contains a single (ordinary) signature component under the fixed (but otherwise arbitrary) name S.

(types)	**let** B **in** T	:=	$\{B;$ **type** $X{=}T\}.X$
(expressions)	**let** B **in** E	:=	$\{B;$ **val** $X{=}E\}.X$
(signatures)	**let** B **in** S	:=	$\{B;$ **signature** $X{=}S\}.X$
	$P\ M$	:=	$(P\ M).\mathsf{S}$
(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** $X{:}S \Rightarrow X)\ M$
(declarations)	**local** B **in** D	:=	**include** (**let** B **in** $\{D\}$)
	signature $X(X'{:}S'){=}S$	:=	**module** $X : (X'{:}S') \to \{$**signature** $\mathsf{S}{=}S\}$
(bindings)	**local** B **in** B'	:=	**include** (**let** B **in** $\{B'\}$)
	signature $X(X'{:}S'){=}S$	:=	**module** $X =$ **fun** $X'{:}S' \Rightarrow \{$**signature** $\mathsf{S}{=}S\}$

Fig. 2. Derived forms.

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 :> {**type** t; **val** v:t}) that defines both an abstract type t and a value v of type t. Then (M :> {**type** t; **val** v:t}).t is not a valid path, because it denotes a fresh abstract type that is not well defined outside of the module. Put another way, projecting t does not make sense because the sealing in the definition of the module should prevent one from tying the identity of its t component back to the module expression itself. Likewise, (M :> {**type** t; **val** v:t}).v is not valid because it cannot be given a type that makes sense outside of the module. (We will explain the issue with paths in more detail in Section 4.)

Here, instead of restricting the *syntax* of paths P, we instead restrict their *semantics*. That is, paths are syntactically just arbitrary module expressions, but with a separate typing rule. This rule will impose additional restrictions on P's signature, to make sure that no locally defined abstract types escape their scope.

In a similar manner, our module-level projection construct $M.X$ is also more permissive than in actual SML, in that M is allowed to be an arbitrary module expression. It is worth noting that this, together with our more permissive notion of path, allows us to define very general forms of local module bindings simply as derived syntax (Figure 2).

3 System F_ω

Our goal in this article is to define the semantics of the module language by translation into System F_ω. To differentiate *external* (module) and *internal* (F_ω) language, we use lowercase letters to range over phrases of the latter. Figure 4 gives the syntax of the variant of System F_ω that we use as the target of our elaboration. It includes record types (where we assume that labels are always disjoint), but is otherwise completely standard.

```
signature EQ =
{
  type t
  val eq : t × t → bool
}

signature ORD =
{
  include EQ
  val less : t × t → bool
}

signature SET =
{
  type set
  type elem
  val empty : set
  val add : elem × set → set
  val mem : elem × set → 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}
```

Fig. 3. Example: a functor for sets.

We note in passing that we are using the usual impredicative definition of F_ω in this article. Up to the introduction of first-class modules in Section 6 we could actually restrict ourselves to a predicative variant. Likewise, as mentioned earlier, up to the introduction of applicative functors in Section 7, the elaboration does not actually require higher kinds (unless used by the core language); second-order System F would suffice. But for simplicity, we have chosen to use just one version of the target language throughout the article.

In the grammar, and elsewhere, we liberally use the meta-notation $\overline{A}$ to stand for zero or more iterations of an object or formula A. (We will also sometimes abuse the notation $\overline{A}$ to actually denote the unordered set $\{\overline{A}\}$.)

We write $\mathrm{fv}(\tau)$ for the free variables of τ.

Semantics. The full static semantics is given in Figure 5. Type equivalence is defined as $\beta\eta$-equivalence. The only other point of note is that, unlike in most presentations,

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

Fig. 4. Syntax of F_ω.

our typing environments Γ permit shadowing of bindings for value variables x (but not for type variables α). Thus, we take the notation $\Gamma(x)$ to denote the rightmost binding of x in Γ. Allowing shadowing turns out to be convenient for our purposes (see Section 4).

We assume a standard left-to-right call-by-value dynamic semantics, which is defined in Figure 6. However, other choices of evaluation order are possible as well, and would not affect our development.

Properties. The calculus as defined here enjoys the standard soundness properties:

Theorem 3.1 (*Preservation*)
If $\cdot \vdash e : \tau$ and $e \hookrightarrow e'$, then $\cdot \vdash e' : \tau$.

Theorem 3.2 (*Progress*)
If $\cdot \vdash e : \tau$ and $e \neq v$ for any v, then $e \hookrightarrow e'$ for some e'.

The proofs are entirely standard, and thus omitted.

The calculus also has the usual technical properties, the most relevant for our purposes being the following:

Lemma 3.3 (*Validity*)

1. If $\Gamma \vdash \tau : \Omega$, then $\Gamma \vdash \Box$.
2. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \Omega$.

Lemma 3.4 (*Weakening*)
Let $\Gamma' \supseteq \Gamma$ with $\Gamma' \vdash \Box$.

1. If $\Gamma \vdash \tau : \kappa$, then $\Gamma' \vdash \tau : \kappa$.
2. If $\Gamma \vdash e : \tau$, then $\Gamma' \vdash e : \tau$.

Lemma 3.5 (*Strengthening*)
Let $\Gamma' \subseteq \Gamma$ with $\Gamma' \vdash \Box$ and $D = \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma')$.

1. If $\Gamma \vdash \tau : \kappa$ and $\mathrm{fv}(\tau) \cap D = \emptyset$, then $\Gamma' \vdash \tau : \kappa$.
2. If $\Gamma \vdash e : \tau$ and $\mathrm{fv}(e) \cap D = \emptyset$, then $\Gamma' \vdash e : \tau$.

Theorem 3.6 (*Uniqueness of types and kinds*)
Assume $\Gamma \vdash \Box$.

1. If $\Gamma \vdash \tau : \kappa_1$ and $\Gamma \vdash \tau : \kappa_2$, then $\kappa_1 = \kappa_2$.
2. If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 \equiv \tau_2$.

Finally, all judgments of the F_ω type system are decidable:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. F_ω typing.

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

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

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

Fig. 6. F_ω reduction.

Theorem 3.7 (*Decidability*)

1. It is decidable whether $\Gamma \vdash \Box$.
2. It is decidable whether $\Gamma \vdash \tau : \kappa$.
3. It is decidable whether $\Gamma \vdash e : \tau$.
4. If $\Gamma \vdash \tau_1 : \kappa$ and $\Gamma \vdash \tau_2 : \kappa$, it is decidable whether $\tau_1 \equiv \tau_2$.

Note that $\tau_1 \equiv \tau_2$ is defined over raw (*i.e.,* not necessarily well-kinded) types; in particular, even if τ_1 and τ_2 are well-kinded, their equivalence may be established by transitively connecting them through some intermediate types that are ill-kinded. However, as long as τ_1 and τ_2 are well-kinded, and they have *the same kind*, one can test for their equality by $\beta\eta$-reducing them to normal forms (a process which must terminate due to strong normalization of $\beta\eta$-reduction) and then comparing the normal forms for α-equivalence. The proof that this algorithm is complete requires only a straightforward extension of the corresponding proof for the simply-typed λ-calculus (Geuvers, 1992), of which F_ω's type language is but a minor generalization.

From here on, we will usually silently assume all these standard properties as given and omit any explicit reference to the above lemmas and theorems.

Parallel substitution. We will also make use of parallel type substitutions on F_ω types and terms. We write them as $[\overline{\tau}/\overline{\alpha}]$ and implicitly assume that $\overline{\tau}$ and $\overline{\alpha}$ are vectors with the same arity. Furthermore, the following definitions and lemmas will come in handy in dealing with parallel type substitutions in proofs.

Definition 3.8 (*Typing of type substitutions*)
We write $\Gamma' \vdash [\overline{\tau}/\overline{\alpha}] : \Gamma$ if and only if

1. $\Gamma' \vdash \Box$,
2. $\overline{\alpha} \subseteq \mathrm{dom}(\Gamma)$,
3. for all $\alpha \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash \alpha[\overline{\tau}/\overline{\alpha}] : \Gamma(\alpha)$,
4. for all $x \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash x : \Gamma(x)[\overline{\tau}/\overline{\alpha}]$.

Lemma 3.9 (*Type substitution*)
Let $\Gamma' \vdash [\overline{\tau}/\overline{\alpha}] : \Gamma$. Then:

1. If $\Gamma \vdash \tau' : \kappa$, then $\Gamma' \vdash \tau'[\overline{\tau}/\overline{\alpha}] : \kappa$.
2. If $\Gamma \vdash e : \tau'$, then $\Gamma' \vdash e[\overline{\tau}/\overline{\alpha}] : \tau'[\overline{\tau}/\overline{\alpha}]$.

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

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

Fig. 7. Notational abbreviations for F_ω.

Abbreviations. Figure 7 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 of other constructs (*e.g.*, application of a type λ), defined in all instances in the obvious way.

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_ω. That is, we will give (syntax-directed) translation rules that interpret signatures as F_ω types, and modules as F_ω terms. Our elaboration translation builds on a number of ideas for representing modules that originate in previous work (see Section 11 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 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_ω. To streamline the presentation, we assume that this mapping is applied implicitly, and thus we use module-language identifiers as if they were F_ω variables.

Second, we assume that there is an injective embedding of F_ω variables into F_ω *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_ω 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.

For simplicity, we assume here that all entities of the language share a single identifier namespace. Obviously, this could be refined by using different injection functions for the different namespaces, with disjoint images.

Finally, we deal with shadowing of module-language identifiers simply via shadowing in the F_ω environment (see Section 3). Consequently, we need not make any

(kind elaboration)	$\Gamma \vdash K \rightsquigarrow \kappa$	
(type elaboration)	$\Gamma \vdash T : \kappa \rightsquigarrow \tau$	such that $\Gamma \vdash \tau : \kappa$
(expression elaboration)	$\Gamma \vdash E : \tau \rightsquigarrow e$	such that $\Gamma \vdash e : \tau$
(path elaboration)	$\Gamma \vdash P : \Sigma \rightsquigarrow e$	such that $\Gamma \vdash e : \Sigma$
(module elaboration)	$\Gamma \vdash M : \Xi \rightsquigarrow e$	such that $\Gamma \vdash e : \Xi$
(binding elaboration)	$\Gamma \vdash B : \Xi \rightsquigarrow e$	such that $\Gamma \vdash e : \Xi$
(signature elaboration)	$\Gamma \vdash S \rightsquigarrow \Xi$	such that $\Gamma \vdash \Xi : \Omega$
(declaration elaboration)	$\Gamma \vdash D \rightsquigarrow \Xi$	such that $\Gamma \vdash \Xi : \Omega$
(signature subtyping)	$\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f$	such that $\Gamma \vdash f : \Xi \to \Xi'$
(signature matching)	$\Gamma \vdash \Sigma \leqslant \Xi' \uparrow \overline{\tau} \rightsquigarrow f$	such that $\Gamma \vdash f : \Sigma \to \Sigma'[\overline{\tau}/\overline{\alpha}]$ (where $\Xi' = \exists\overline{\alpha}.\Sigma'$)

Fig. 8. Elaboration judgments.

specific provision for variable shadowing in our rules. Only when identifiers are turned into *labels* (*e.g.,* as structure fields) do we need to explicitly avoid duplicates.

Judgments. The judgments comprising our elaboration semantics are listed in Figure 8. Most of these are *translation* judgments, one for each syntactic class of the module language, which translate module-language entities into F_ω entities of the corresponding variety. (Strictly speaking, we ambiguously overload the same notation for module and path judgments, since P syntactically expands to M. But it will always be clear from context which judgment is referenced.) The last two are auxiliary judgments for signature subtyping and matching, which we will explain a bit later.

For each judgment, the figure also shows the corresponding elaboration invariant. We will prove that these invariants hold (and that the translation thereby is sound) later, in Section 5.1. To prove them, we assume that elaboration starts out with a well-formed context Γ. In fact, elaboration will maintain much stronger invariants for Γ, which are important in the proof of decidability of typechecking, but we leave discussion of the details until later (see the "Module elaboration" section below, as well as Section 5.2).

In places where we do not care about evidence terms, we will often write judgments without the "$\rightsquigarrow e$" or "$\rightsquigarrow f$" part. In addition, we use $\Gamma \vdash \Xi \leqslant\!\geqslant \Xi'$ as a shorthand for mutual subtyping $\Gamma \vdash \Xi \leqslant \Xi' \ \wedge \ \Gamma \vdash \Xi' \leqslant \Xi$.

A number of the elaboration judgments concern *semantic signatures* Σ or Ξ. Semantic signatures are just a subclass of F_ω types 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 will start by explaining how they work.

Semantic signatures. The syntax of semantic signatures is given in Figure 9. (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_ω syntax.)

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

$$\begin{array}{lll} \text{(meta-projection)} \quad \Sigma.\epsilon & := & \Sigma \\ \{l : \Sigma, \overline{l' : \Sigma'}\}.l.\overline{l} & := & \Sigma.\overline{l} \end{array}$$

Fig. 9. Semantic signatures.

Following Mitchell & Plotkin (1988), 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 below.) The splitting of an abstract signature $\exists\overline{\alpha}.\Sigma$ into these two components — the abstract types $\overline{\alpha}$ and the concrete signature Σ — plays a key role in the elaboration of module binding (as we explain in the "Module elaboration" section below).

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 $\forall\overline{\alpha}.\Sigma \to \Xi$).

Instead of adding atomic signatures as primitive constructs to the type system of the internal language (like in previous work, *e.g.,* Dreyer *et al.* (2003)), we simply encode them as syntactic sugar for F_ω types of a certain form. Their encodings are shown in Figure 10, along with corresponding term forms (which we will use in the translation of modules), and associated typing rules that are admissible in System F_ω. The encodings 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 $[= \Xi]$ are injective, *i.e.,* uniquely (up to F_ω type equivalence) determine τ and Ξ, respectively. The encoding for $[= \tau : \kappa]$ is chosen such that it supports arbitrary κ. Beyond these properties the "implementation details" of the encodings are immaterial to the rest of our development, and the reader should simply view them as abstractions.

In the remainder of this article, we will assume implicitly that all semantic types and signatures are reduced to $\beta\eta$-normal form. Likewise, we assume that all uses of substitution are followed by an implicit normalization step. This is convenient as a way of determinizing elaboration, as well as ensuring that types produced by

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

Types

$$\frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash [\tau] : \Omega} \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash [= \tau : \kappa] : \Omega} \qquad \frac{\Gamma \vdash \Xi : \Omega}{\Gamma \vdash [= \Xi] : \Omega} \qquad \boxed{\Gamma \vdash \tau : \kappa}$$

Terms

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] : [\tau]} \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash [\tau : \kappa] : [= \tau : \kappa]} \qquad \frac{\Gamma \vdash \Xi : \Omega}{\Gamma \vdash [\Xi] : [= \Xi]} \qquad \boxed{\Gamma \vdash e : \tau}$$

Type equivalence

$$\frac{\tau \equiv \tau'}{[= \tau : \kappa] \equiv [= \tau' : \kappa]} \qquad \frac{\Xi \equiv \Xi'}{[= \Xi] \equiv [= \Xi']} \qquad \boxed{\tau \equiv \tau'}$$

Fig. 10. F_ω encodings of atomic signatures and admissible typing rules.

elaboration mention the minimal set of free type variables relevant to their identity (cf. "path elaboration" below).

Signature elaboration. The elaboration of signatures (Figure 11) is not difficult. 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 (notably, in rule D-MOD), and bound existentially at the outermost level of the signature.

For example, consider the following syntactic signature:

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

This signature declares *one* abstract type (A.t), so the semantic F_ω 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 denoting that S is a signature component whose definition is the semantic signature $\{\mathsf{f} : [\alpha \to \mathsf{int}]\}$.

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

$$\frac{\Gamma \vdash P : [= \Xi] \rightsquigarrow e}{\Gamma \vdash P \rightsquigarrow \Xi}\ \text{S-PATH}$$

$$\frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi}\ \text{S-STRUCT}$$

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

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

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

$$\frac{\Gamma \vdash T : \Omega \rightsquigarrow \tau}{\Gamma \vdash \textbf{val}\ X{:}T \rightsquigarrow \{l_X : [\tau]\}}\ \text{D-VAL}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type}\ X{=}T \rightsquigarrow \{l_X : [= \tau : \kappa]\}}\ \text{D-TYP-EQ} \qquad \frac{\Gamma \vdash K \rightsquigarrow \kappa_\alpha}{\Gamma \vdash \textbf{type}\ X{:}K \rightsquigarrow \exists \alpha.\{l_X : [= \alpha : \kappa_\alpha]\}}\ \text{D-TYP}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma}{\Gamma \vdash \textbf{module}\ X{:}S \rightsquigarrow \exists \overline{\alpha}.\{l_X : \Sigma\}}\ \text{D-MOD}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi}{\Gamma \vdash \textbf{signature}\ X{=}S \rightsquigarrow \{l_X : [= \Xi]\}}\ \text{D-SIG-EQ}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\{\overline{l_X : \Sigma}\}}{\Gamma \vdash \textbf{include}\ S \rightsquigarrow \exists \overline{\alpha}.\{\overline{l_X : \Sigma}\}}\ \text{D-INCL}$$

$$\frac{}{\Gamma \vdash \epsilon \rightsquigarrow \{\}}\ \text{D-EMT} \qquad \frac{\Gamma \vdash D_1 \rightsquigarrow \exists \overline{\alpha}_1.\{\overline{l_{X_1} : \Sigma_1}\} \qquad \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash D_2 \rightsquigarrow \exists \overline{\alpha}_2.\{\overline{l_{X_2} : \Sigma_2}\} \qquad \overline{l_{X_1}} \cap \overline{l_{X_2}} = \emptyset}{\Gamma \vdash D_1;D_2 \rightsquigarrow \exists \overline{\alpha}_1 \overline{\alpha}_2.\{\overline{l_{X_1} : \Sigma_1}, \overline{l_{X_2} : \Sigma_2}\}}\ \text{D-SEQ}$$

Fig. 11. Signature elaboration.

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

$$
\begin{aligned}
&\mathsf{SET} \rightsquigarrow \exists \alpha_1 \alpha_2.\{\mathsf{set} : [= \alpha_1 : \Omega],\\
&\qquad\qquad \mathsf{elem} : [= \alpha_2 : \Omega],\\
&\qquad\qquad \mathsf{empty} : [\alpha_1],\\
&\qquad\qquad \mathsf{add} : [\alpha_2 \times \alpha_1 \to \alpha_1],\\
&\qquad\qquad \mathsf{mem} : [\alpha_2 \times \alpha_1 \to \mathsf{bool}]\}
\end{aligned}
$$

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

Fig. 12. Example: signature elaboration.

the "syntactic" type systems for modules (Harper & Lillibridge, 1994; Leroy, 1994; Leroy, 1995; Shao, 1999; Dreyer *et al.*, 2003).

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 in both 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 12. It gives the translation of the signature SET from the example in Figure 3, along with the translation of the signature

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

which classifies the Set functor itself.

Given our informal explanation, the formal rules in Figure 11 should now be very easy to follow. A few points of note, though.

The rule S-WHERE-TYP for **where type** employs a convenient bit of shorthand notation defined in Figure 9, 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 D-SEQ, for sequences of declarations $D_1;D_2$, the side condition that the label sets $\overline{l_{X_1}}$ and $\overline{l_{X_2}}$ are disjoint is imposed 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 S-PATH for signature paths P refers in its premise to the *path* elaboration judgment (which we will discuss later, see Figure 17) 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 $\boxed{\Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f}$

$$\frac{\overline{\Gamma \vdash \tau : \kappa_\alpha} \qquad \Gamma \vdash \Sigma \leqslant \Sigma'[\overline{\tau}/\overline{\alpha}] \rightsquigarrow f}{\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f} \text{ U-MATCH}$$

Subtyping $\boxed{\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f}$

$$\frac{\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f}{\Gamma \vdash [\tau] \leqslant [\tau'] \rightsquigarrow \lambda x{:}[\tau].[f\,(x.\mathsf{val})]} \text{ U-VAL}$$

$$\frac{\tau = \tau'}{\Gamma \vdash [= \tau : \kappa] \leqslant [= \tau' : \kappa] \rightsquigarrow \lambda x{:}[= \tau : \kappa].x} \text{ U-TYP}$$

$$\frac{\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f \qquad \Gamma \vdash \Xi' \leqslant \Xi \rightsquigarrow f'}{\Gamma \vdash [= \Xi] \leqslant [= \Xi'] \rightsquigarrow \lambda x{:}[= \Xi].\,[\Xi']} \text{ U-SIG}$$

$$\frac{\overline{\Gamma \vdash \Sigma_1 \leqslant \Sigma_1' \rightsquigarrow f}}{\Gamma \vdash \{\overline{l_1 : \Sigma_1}, \overline{l_2 : \Sigma_2}\} \leqslant \{\overline{l_1 : \Sigma_1'}\} \rightsquigarrow \lambda x{:}\{\overline{l_1 : \Sigma_1}, \overline{l_2 : \Sigma_2}\}.\{\overline{l_1 = f\,(x.l_1)}\}} \text{ U-STRUCT}$$

$$\frac{\Gamma, \overline{\alpha}' \vdash \Sigma' \leqslant \exists \overline{\alpha}.\Sigma \uparrow \overline{\tau} \rightsquigarrow f_1 \qquad \Gamma, \overline{\alpha}' \vdash \Xi[\overline{\tau}/\overline{\alpha}] \leqslant \Xi' \rightsquigarrow f_2}{\Gamma \vdash (\forall \overline{\alpha}.\Sigma \to \Xi) \leqslant (\forall \overline{\alpha}'.\Sigma' \to \Xi') \rightsquigarrow \lambda f{:}(\forall \overline{\alpha}.\Sigma \to \Xi).\,\lambda \overline{\alpha}'.\,\lambda x{:}\Sigma'.\,f_2\,(f\,\overline{\tau}\,(f_1\,x))} \text{ U-FUNCT}$$

$$\frac{\Gamma, \overline{\alpha} \vdash \Sigma \leqslant \exists \overline{\alpha}'.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash \exists \overline{\alpha}.\Sigma \leqslant \exists \overline{\alpha}'.\Sigma' \rightsquigarrow \lambda x{:}(\exists \overline{\alpha}.\Sigma).\ \mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle} \text{ U-ABS}$$

Fig. 13. Signature matching and subtyping.

Signature matching and subtyping. Signature matching (Figure 13) is a key element of the ML module system. For sealed module expressions, we must check that the signature of the module being sealed matches the sealing signature. At functor applications, we must check that the signature of the actual argument matches the formal argument signature of the functor.

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 structural (width and depth) subtype of the second after substituting int for α.

The signature matching judgment has the form $\Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f$. It matches a concrete Σ against an abstract Ξ of the form $\exists\overline{\alpha}.\Sigma'$ as described above, non-deterministically synthesizing the solution $\overline{\tau}$ for $\overline{\alpha}$, as well as the coercion f from Σ to $\Sigma'[\overline{\tau}/\overline{\alpha}]$ (rule U-MATCH).

While the purpose of signature matching is to relate concrete to abstract signatures, signature *subtyping*, $\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f$, 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 (rule U-VAL), signature subtyping appeals to an assumed subtyping judgment for the core language, $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f$. For a core language with no subtyping the premise would degenerate to "$\tau = \tau'$". 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 (rule U-TYP), we require type equivalence, so subtyping just produces an identity coercion.

For signature declarations (rule U-SIG), 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 indeed, the order of quantifiers does not matter anywhere in our rules, because they are only used for matching, and pushed around *en bloc* in all other places. (Ordering of quantifiers will, however, matter for modules as first-class values — see the discussion of signature normalization in Section 6.)

For structure signatures, we allow both width and depth subtyping (rule U-STRUCT). For functor signatures, $\forall\overline{\alpha}.\Sigma \to \Xi$ and $\forall\overline{\alpha}'.\Sigma' \to \Xi'$, subtyping proceeds in the usual contra- and co-variant manner (rule U-FUNCT): after introducing $\overline{\alpha}'$, we *match* the domains contravariantly to determine an instantiation $\overline{\tau}$ for $\overline{\alpha}$ such that $\Sigma' \leqslant \Sigma[\overline{\tau}/\overline{\alpha}]$; then, we (covariantly) 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 (rule U-ABS).

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. This invariant also determines the elided type annotation on the pack expression in the U-ABS rule.

We assume $\beta\eta$-equivalence for System F_ω types, which is important to make certain examples work as expected. Consider the following two signatures[6]:

$$\textbf{signature}\ \mathsf{A} = \{\textbf{type}\ \mathsf{t} : \star \to \star;\ \textbf{type}\ \mathsf{u} = \textbf{fun}\ \mathsf{a} \Rightarrow \mathsf{t}\ \mathsf{a}\}$$
$$\textbf{signature}\ \mathsf{B} = \{\textbf{type}\ \mathsf{u} : \star \to \star;\ \textbf{type}\ \mathsf{t} = \textbf{fun}\ \mathsf{a} \Rightarrow \mathsf{u}\ \mathsf{a}\}$$

Semantically, they are represented as:

$$\begin{aligned} A &= \exists\beta_1 : \Omega \to \Omega.\{\mathsf{t} : [= \beta_1 : \Omega \to \Omega], \mathsf{u} : [= \lambda\alpha.\beta_1\,\alpha : \Omega \to \Omega]\} \\ B &= \exists\beta_2 : \Omega \to \Omega.\{\mathsf{u} : [= \beta_2 : \Omega \to \Omega], \mathsf{t} : [= \lambda\alpha.\beta_2\,\alpha : \Omega \to \Omega]\} \end{aligned}$$

Intuitively, $A \leqslant B$ is expected to hold (and vice versa). According to rules U-ABS and U-MATCH, this boils down to finding a type $\tau : \Omega \to \Omega$ such that

$$\begin{aligned} &\{\mathsf{t} : [= \beta_1 : \Omega \to \Omega], \mathsf{u} : [= \lambda\alpha.\beta_1\,\alpha : \Omega \to \Omega]\} \\ &\quad\leqslant \{\mathsf{u} : [= \tau : \Omega \to \Omega], \mathsf{t} : [= \lambda\alpha.\tau\,\alpha : \Omega \to \Omega]\} \end{aligned}$$

By rule U-TYP, the following equivalences need to hold for a suitable choice of τ:

$$\begin{aligned} \beta_1 &= \lambda\alpha.\tau\,\alpha \quad \text{(via t)} \\ \lambda\alpha.\beta_1\,\alpha &= \tau \quad \text{(via u)} \end{aligned}$$

Substituting the solution for τ, given by the second equation, into the first reveals that the following will have to hold:

$$\beta_1 = \lambda\alpha.(\lambda\alpha.\beta_1\,\alpha)\,\alpha$$

Clearly, this is only the case under a combination of both β- and η-equivalence.

Module elaboration. The module elaboration judgment (Figure 14), which has the form $\Gamma \vdash M : \Xi \rightsquigarrow e$, 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*.

One 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 *abstract* (think: monadic) signature, we must first unpack it (think: the bind operation for the monad), obtaining some fresh

[6] In this and later examples, we use the syntax **fun** X ⇒ T to denote a type function in our imaginary Core language.

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

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X : \Sigma \rightsquigarrow X}\ \text{M-VAR}$$

$$\frac{\Gamma \vdash B : \Xi \rightsquigarrow e}{\Gamma \vdash \{B\} : \Xi \rightsquigarrow e}\ \text{M-STRUCT}$$

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

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash M : \Xi \rightsquigarrow e}{\Gamma \vdash \mathbf{fun}\ X{:}S \Rightarrow M : \forall \overline{\alpha}.\, \Sigma \to \Xi \rightsquigarrow \lambda \overline{\alpha}.\lambda X{:}\Sigma.e}\ \text{M-FUNCT}$$

$$\frac{\Gamma(X_1) = \forall \overline{\alpha}.\, \Sigma' \to \Xi \qquad \Gamma(X_2) = \Sigma \qquad \Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X_1\, X_2 : \Xi[\overline{\tau}/\overline{\alpha}] \rightsquigarrow X_1\, \overline{\tau}\, (f\, X_2)}\ \text{M-APP}$$

$$\frac{\Gamma(X) = \Sigma \qquad \Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X{:>}S : \Xi \rightsquigarrow \mathsf{pack}\ \langle \overline{\tau}, f\, X\rangle}\ \text{M-SEAL}$$

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

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e}{\Gamma \vdash \mathbf{val}\ X{=}E : \{l_X : [\tau]\} \rightsquigarrow \{l_X = [e]\}}\ \text{B-VAL}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \mathbf{type}\ X{=}T : \{l_X : [= \tau : \kappa]\} \rightsquigarrow \{l_X = [\tau : \kappa]\}}\ \text{B-TYP}$$

$$\frac{\Gamma \vdash M : \exists \overline{\alpha}.\Sigma \rightsquigarrow e \qquad \Sigma \text{ not atomic}}{\Gamma \vdash \mathbf{module}\ X{=}M : \exists \overline{\alpha}.\{l_X : \Sigma\} \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\alpha}, \{l_X = x\}\rangle}\ \text{B-MOD}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi}{\Gamma \vdash \mathbf{signature}\ X{=}S : \{l_X : [= \Xi]\} \rightsquigarrow \{l_X = [\Xi]\}}\ \text{B-SIG}$$

$$\frac{\Gamma \vdash M : \exists \overline{\alpha}.\{\overline{l_X : \Sigma}\} \rightsquigarrow e}{\Gamma \vdash \mathbf{include}\ M : \exists \overline{\alpha}.\{\overline{l_X : \Sigma}\} \rightsquigarrow e}\ \text{B-INCL}$$

$$\frac{}{\Gamma \vdash \epsilon : \{\} \rightsquigarrow \{\}}\ \text{B-EMT}$$

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

Fig. 14. Module elaboration.

abstract types $\overline{\alpha}$ and a variable x of *concrete* (think: pure) signature Σ. We can then do whatever we want with x, ultimately producing another module of (monadic) 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. As many researchers have pointed out (MacQueen, 1986; Cardelli & Leroy, 1990), this monadic unpack/repack style of existential programming would be annoying to program manually. Fortunately, it is easy for module elaboration to perform it automatically.

Figure 14 shows the rules for elaborating modules and bindings. The rules for projections (M-DOT), module bindings (B-MOD), and binding sequences (B-SEQ) 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 (with the exception of Harper & Stone (2000), who account for full Standard ML).[7] It is here primarily that we rely on the fact that the F_ω version from Section 3 allows shadowing in Γ, in order to avoid having to map external identifiers to fresh internal variables. (In fact, we have already relied on this for rule S-FUNCT, and do so again for rule M-FUNCT.)

The rule M-FUNCT for functors is completely analogous to rule S-FUNCT for functor signatures (cf. Figure 11). Note that this rule and the sequence rule B-SEQ 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 (rule M-VAR), we may assume it has a concrete signature. This is a key invariant of elaboration.

The rules for functor applications (M-APP) and sealed modules (M-SEAL) 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 formal 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 $\overline{\alpha}$.

Note that both M-APP and M-SEAL are made simpler by our language's restriction of functor applications and sealing to module *identifiers* ($X_1\,X_2$ and $X :> S$), which enables us to exploit the elaboration invariant that those identifiers (the X's) already have concrete signatures and need not be unpacked. As the **let**-binding encodings

[7] Of course, 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.

```
Set ⇝
 λα.λElem : {t : [= α : Ω],
             eq : [α × α → bool],
             less : [α × α → bool]}.
      pack ⟨list α,
            f (let y1 = {elem = [α : Ω]} in
               let y2 =
                  let elem = y1.elem in
                  let y21 = {set = [list α : Ω]} in
                  let y22 =
                     let set = y21.set in
                     ...
               in {elem = y1.elem,
                   set = y2.set,
                   empty = y2.empty,
                   add = y2.add,
                   mem = y2.mem})
           ⟩∃β.{set:[=β:Ω], elem:[=α:Ω], empty:[β], add:[α×β→β], mem:[α×β→bool]}

{module IS = Set Int; val s = IS.add (7, IS.empty)} ⇝
    unpack ⟨β, y1⟩ = {IS = Set int (f′ Int)} in
    let y2 = (let IS = y1.IS in {s = [IS.add ⟨7, IS.empty⟩]}) in
    pack ⟨β, {IS = y1.IS, s = y2.s}⟩∃β.{IS:{...},s:[β]}
```

Fig. 15. Example: module elaboration.

of the more general forms $M_1\,M_2$ and $M{:>}S$ in Figure 2 suggest, elaboration of those forms just involves monadically unpacking the M's to X's first before applying M-APP or M-SEAL, and then repacking afterward.

As an example of the module elaboration translation, Figure 15 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, whose t component is transparently equal to *int*, and whose F_ω representation is *Int*. In order to avoid too much clutter, we do not spell out the respective coercions f and f' occurring in both examples.

To make the essence of the translation a bit more apparent, Figure 16 shows simplified versions of the same translations with all intermediate redexes (in particular, intermediate structures) removed, via straightforward $\beta\eta$-transformations of let-bindings and records. In particular, once we eliminate the administrative overhead of rule B-SEQ, a structure simply becomes a sequence of let-bindings for the declarations in its body, feeding into a record that collects all bound variables as fields.

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

$$
\begin{array}{l}
\mathsf{Set} \rightsquigarrow \\
\quad \lambda\alpha.\lambda \mathit{Elem} : \{\mathsf{t} : [= \alpha : \Omega], \\
\qquad\qquad \mathsf{eq} : [\alpha \times \alpha \to \mathsf{bool}], \\
\qquad\qquad \mathsf{less} : [\alpha \times \alpha \to \mathsf{bool}]\}. \\
\qquad \mathsf{pack}\ \langle \mathsf{list}\,\alpha, \\
\qquad\qquad f\,(\mathsf{let}\ \mathit{elem} = [\alpha : \Omega]\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ \mathit{set} = [\mathsf{list}\,\alpha : \Omega]\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ \mathit{empty} = [\mathsf{nil}]\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ \mathit{add} = [\ldots \mathit{Elem}.\mathsf{eq} \ldots \mathit{Elem}.\mathsf{less} \ldots]\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ \mathit{mem} = [\ldots \mathit{Elem}.\mathsf{eq} \ldots \mathit{Elem}.\mathsf{less} \ldots]\ \mathsf{in} \\
\qquad\qquad\quad \{\mathsf{elem} = \mathit{elem},\ \mathsf{set} = \mathit{set},\ \mathsf{empty} = \mathit{empty},\ \mathsf{add} = \mathit{add},\ \mathsf{mem} = \mathit{mem}\}) \\
\qquad\qquad \rangle_{\exists\beta.\{\mathsf{set}:[=\beta:\Omega],\ \mathsf{elem}:[=\alpha:\Omega],\ \mathsf{empty}:[\beta],\ \mathsf{add}:[\alpha\times\beta\to\beta],\ \mathsf{mem}:[\alpha\times\beta\to\mathsf{bool}]\}}
\end{array}
$$

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

Fig. 16. Example: module elaboration, simplified.

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.

In Section 7, we will explore an alternative semantics, where functors can be *applicative*, *i.e.,* applying such a functor twice (to the same argument) will only produce one copy of the abstract types it defines.

Path elaboration. Figure 17 displays the last three rules of elaboration, concerning the elaboration of paths. (The elaboration rule for signature paths appeared in Figure 11.)

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 Γ. In other words, paths P need to elaborate to a *concrete* signature Σ, because (unlike for module constructs) existential quantifiers cannot be "extruded" further in the contexts where paths occur. To ensure this, the path elaboration judgment, $\Gamma \vdash P : \Sigma \rightsquigarrow e$, uses the *ordinary* module elaboration judgment, $\Gamma \vdash M : \Xi \rightsquigarrow e$, in its premise (with $M = P$) to synthesize P's semantic signature $\exists\overline{\alpha}.\Sigma$, which still allows "local" abstract types $\overline{\alpha}$ to occur. It then checks that Σ does not actually depend on any of these $\overline{\alpha}$ that P may have defined (note that we assume all types normalized, so any spurious dependencies are implicitly eliminated). The rules for

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

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

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

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

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

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

Fig. 17. Path elaboration.

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

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

The semantic signature that module elaboration assigns to M is:

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

Thus, if we were to 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 that is not going to be bound in the environment Γ. 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, as we have suggested, 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" (or "encapsulated") effects by Haskell's runST mechanism (Launchbury & Peyton Jones, 1995).

5 Meta-theoretic properties

Having defined the semantics of ML modules by elaboration into System F_ω, it is time to prove it (a) sound, and (b) decidable.

Some theorems about the module language depend on the assumption that respective properties can be proved for core language elaboration (*i.e.,* the first three judgments listed in Figure 8). However, because both language layers are mutually recursive through the syntax of paths (and after Section 6, also through modules as first-class values), these proofs are typically not independent — they need to be performed by simultaneous induction on the derivations for both language layers. We hence state all properties that we assume about the core language as part of the respective theorems below. The theorems then hold provided that the inductive argument can also be shown for all additional cases not specified by our grammar for types T and expressions E.

5.1 Soundness

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

1. Showing 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 thus remains to be shown that the elaboration rules produce well-formed F_ω expressions. 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.

Formally, we state the following theorem, which collects the elaboration invariants already stated in Figure 8:

Theorem 5.1 (*Soundness of elaboration*)
Provided $\Gamma \vdash \square$ 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 $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f$ and $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$, then $\Gamma \vdash f : \tau \to \tau'$.
4. If $\Gamma \vdash S/D \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
5. If $\Gamma \vdash P/M/B : \Xi \rightsquigarrow e$, then $\Gamma \vdash e : \Xi$.
6. If $\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f$ and $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$, then $\Gamma \vdash f : \Xi \to \Xi'$.
7. If $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f$ 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
The proof is by relatively straightforward simultaneous induction on derivations. The arguments for properties 1–3 clearly depend on the core language, and we assume that it can be proved for all additional cases not specified in our grammar. We have performed the entire proof in Coq (Section 10), and transliterate only two representative cases here:

- Case M-APP: 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-SEQ: By induction on the first premise we know (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 $\Gamma_1 \vdash \square$. 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 $\Gamma, \overline{\alpha}_1, y_1 : \{\overline{l_{X_1} : \Sigma_1}\} \vdash y_1.l_{X_1} : \Sigma_1$. By convention, y_1 and y_2 are fresh, and so it follows that $\Gamma, \overline{\alpha}_1, y_1 : \{\overline{l_{X_1} : \Sigma_1}\}, \overline{\alpha}_2, y_2 : \{\overline{l_{X_2} : \Sigma_2}\} \vdash \{\overline{l'_{X_1} = y_1.l'_{X_1}}, \overline{l_{X_2} = y_2.l_{X_2}}\} : \{\overline{l'_{X_1} : \Sigma'_1}, \overline{l_{X_2} : \Sigma_2}\}$ from the typing rules. 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. $\square$

If the reader finds the proof cases shown here to be boring and straightforward, that is because they are! The remaining cases are even more boring. In other words, there is nothing tricky going on in our elaboration — which substantiates our claim that it is simple.

5.2 Decidability

All our elaboration rules are syntax-directed, and they can be interpreted directly as a deterministic 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 (Dreyer *et al.*, 2003):

$$\forall\alpha.[= \alpha : \kappa] \to [= \tau_1 : \kappa'] \quad \leqslant \quad \exists\beta.([= \beta : \kappa] \to [= \tau_2 : \kappa'])$$

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 $\tau_1[\tau/\alpha] = \tau_2[\tau/\beta]$. Since κ may be a higher kind, this amounts to a higher-order unification problem, which is undecidable in general (Goldfarb, 1981).

Validity. 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 $\Sigma \leqslant \exists\overline{\alpha}.\Sigma'$, the target signature Σ' has the property that each abstract type variable $\alpha \in \overline{\alpha}$ actually occurs *explicitly* in Σ' 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*. Intuitively, the reason we can expect the target signature $\exists\overline{\alpha}.\Sigma'$ to be explicit is that (1) the only signatures we ever match *against* during elaboration are themselves the result of elaborating some *ML signature* S, and (2) all of such a signature's abstract types $\overline{\alpha}$ must originate in some opaque type specification appearing in S.

$$\begin{array}{lcl}
\overline{\alpha} \text{ rooted in } \Sigma & :\Leftrightarrow & \overline{\alpha \text{ rooted in } \Sigma} \\
\alpha \text{ rooted in } [= \tau : \kappa] \text{ (at } \epsilon) & :\Leftrightarrow & \alpha = \tau \\
\alpha \text{ rooted in } \{\overline{l : \Sigma}\} \text{ (at } l.\overline{l'}) & :\Leftrightarrow & \alpha \text{ rooted in } \{\overline{l : \Sigma}\}.l \text{ (at } \overline{l'})
\end{array}$$

$$\begin{array}{rcl|rcl}
[\tau] \text{ explicit} & & \text{(always)} & [\tau] \text{ valid} & & \text{(always)} \\
[= \tau : \kappa] \text{ explicit} & & \text{(always)} & [= \tau : \kappa] \text{ valid} & & \text{(always)} \\
[= \Xi] \text{ explicit} & :\Leftrightarrow & \Xi \text{ explicit} & [= \Xi] \text{ valid} & :\Leftrightarrow & \Xi \text{ explicit} \\
\{\overline{l : \Sigma}\} \text{ explicit} & :\Leftrightarrow & \overline{\Sigma \text{ explicit}} & \{\overline{l : \Sigma}\} \text{ valid} & :\Leftrightarrow & \overline{\Sigma \text{ valid}} \\
\forall \overline{\alpha}.\Sigma \to \Xi \text{ explicit} & :\Leftrightarrow & \exists \overline{\alpha}.\Sigma \text{ explicit} \wedge \Xi \text{ explicit} & \forall \overline{\alpha}.\Sigma \to \Xi \text{ valid} & :\Leftrightarrow & \exists \overline{\alpha}.\Sigma \text{ explicit} \wedge \Xi \text{ valid} \\
\exists \overline{\alpha}.\Sigma \text{ explicit} & :\Leftrightarrow & \overline{\alpha} \text{ rooted in } \Sigma \wedge \Sigma \text{ explicit} & \exists \overline{\alpha}.\Sigma \text{ valid} & :\Leftrightarrow & \Sigma \text{ valid} \\
\Gamma \vdash \Xi : \Omega \text{ explicit} & :\Leftrightarrow & \Gamma \vdash \Xi : \Omega \ \wedge \ \Xi \text{ explicit} & \Gamma \vdash \Xi : \Omega \text{ valid} & :\Leftrightarrow & \Gamma \vdash \Xi : \Omega \ \wedge \ \Xi \text{ valid} \\
 & & & \Gamma \text{ valid} & :\Leftrightarrow & \forall (X : \Sigma) \in \Gamma,\ \Sigma \text{ valid}
\end{array}$$

Fig. 18. Signature explicitness and validity.

Figure 18 gives an inductive definition of these properties. (We typically drop the explicit path description "(at $\overline{l}$)" from the rootedness judgment — the only place where we actually need it will be the definition of signature normalization in Section 6.)

However, this is not all. While it is necessary (in general) that a signature Ξ is explicit to decide matching $\Sigma \leqslant \Xi$, it is not sufficient. Subtyping is contra-variant in functor arguments, so we also need to ensure that, whenever we invoke subtyping to determine whether $\Sigma \leqslant \Sigma'$ and Σ is a *functor* signature, its argument signature is explicit as well. Unfortunately, we cannot require all of Σ to be explicit, because not all module expressions (as opposed to signature expressions) yield explicit signatures. For example,

$$\begin{array}{l}
\textbf{let module } \mathsf{A} = \ \{\textbf{type } \mathsf{t} = \mathsf{int};\ \textbf{val } \mathsf{v} = 5;\ \textbf{val } \mathsf{f}\ \mathsf{x} = \mathsf{x}\} \\
\qquad\qquad\quad :> \{\textbf{type } \mathsf{t};\ \textbf{val } \mathsf{v} : \mathsf{t};\ \textbf{val } \mathsf{f} : \mathsf{t} \to \mathsf{int}\} \\
\textbf{in } \{\textbf{val } \mathsf{f} = \mathsf{A.f};\ \textbf{val } \mathsf{v} = \mathsf{A.v}\}
\end{array}$$

defines a module with the non-explicit signature $\exists\alpha.\{\mathsf{f} : [\alpha \to \mathsf{int}], \mathsf{v} : [\alpha]\}$.

Figure 18 hence defines the second notion of a *valid* signature that captures the relevant property — that is, a signature is valid if all contained functor arguments are explicit (but other constituent signatures need not be). Intuitively, it is expected that modules have valid signatures, because the language requires explicit signature annotations on all functor arguments. The notion of validity is extended to environments, and we require all signatures and environments used in elaboration to be valid.[8] Note that validity of environments only cares about variables bound to concrete signatures Σ because of the elaboration invariant (discussed in Section 4, "Module elaboration") that all modules of signature $\exists\overline{\alpha}.\Sigma$ are unpacked into $\overline{\alpha}$ and $X : \Sigma$ before being added to the context.

[8] The notions of *explicit* and *valid* signatures are also called *analysis* and *synthesis* signatures in the literature (Dreyer *et al.*, 2003; Rossberg & Dreyer, 2013); Russo (1998) used the terms *solvable* and *ground*.

$$\begin{array}{ll}
\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau} & \text{if } \overline{\mathrm{lookup}_{\alpha}(\Sigma, \Sigma') \uparrow \tau} \\
\mathrm{lookup}_{\alpha}([= \tau : \kappa], [= \tau' : \kappa]) \uparrow \tau & \text{if } \tau' = \alpha \\
\mathrm{lookup}_{\alpha}(\{\overline{l : \Sigma}\}, \{\overline{l' : \Sigma'}\}) \uparrow \tau & \text{if } \exists l \in \overline{l} \cap \overline{l'}.\ \mathrm{lookup}_{\alpha}(\{\overline{l : \Sigma}\}.l, \{\overline{l' : \Sigma'}\}.l) \uparrow \tau
\end{array}$$

Fig. 19. Algorithmic type lookup.

With a little auxiliary lemma, we can show that our elaboration establishes and maintains explicit signatures for signature expressions, and valid signatures for module expressions:

Lemma 5.2 (*Simple properties of validity*)

1. If Ξ explicit, then Ξ valid.
2. If Ξ explicit/valid, then $\Xi[\overline{\tau}/\overline{\alpha}]$ explicit/valid.

Lemma 5.3 (*Signature validity*)
Assume Γ valid.

1. If $\Gamma \vdash S/D \rightsquigarrow \Xi$, then Ξ explicit.
2. If $\Gamma \vdash P/M/B : \Xi \rightsquigarrow e$, then Ξ valid.

Type lookup. If the $\exists\overline{\alpha}.\Sigma'$ in the matching rule U-MATCH 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 definition of lookup in Figure 19 implements a suitable algorithm for finding the types $\overline{\tau}$ in rule U-MATCH, through a straightforward parallel traversal of the two signatures involved. There is a twist, though: an abstract type variable may actually have multiple roots in a signature. For example, the external signature $\{\textbf{type}\ \mathsf{t};\ \textbf{type}\ \mathsf{u} = \mathsf{t}\}$ elaborates to $\exists\alpha.\{t : [= \alpha : \Omega], u : [= \alpha : \Omega]\}$. The lookup algorithm, as given in the figure, is non-deterministic in that it can pick any suitable root — specifically, the choice of l in the last clause is not necessarily unique. This formulation simplifies the proof of completeness below. Intuitively, it does not matter which one we pick, they all have to be equivalent. The soundness theorem proves that, but first we need a little technical lemma:

Lemma 5.4 (*Simple properties of type lookup*)

1. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$, then $\mathrm{fv}(\overline{\tau}) \subseteq \mathrm{fv}(\Sigma)$.
2. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$ and $\overline{\alpha} \cap \overline{\alpha}' = \emptyset$, then $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma'[\overline{\tau}'/\overline{\alpha}']) \uparrow \overline{\tau}$ (and both derivations have the same size).
3. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$ and $\Gamma \vdash \Sigma : \Omega$, then $\overline{\Gamma \vdash \tau : \kappa}$.

Theorem 5.5 (*Soundness of type lookup*)

1. Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \alpha \vdash \Sigma' : \Omega$. If $\mathrm{lookup}_{\alpha}(\Sigma, \Sigma') \uparrow \tau_1$, then $\Gamma \vdash \tau_1 : \kappa_\alpha$. Furthermore, if $\Gamma \vdash \Sigma \leqslant \Sigma'[\tau_2/\alpha]$ for $\Gamma \vdash \tau_2 : \kappa_\alpha$, then $\tau_1 = \tau_2$.
2. Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Sigma' : \Omega$. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_1$, then $\overline{\Gamma \vdash \tau_1 : \kappa_\alpha}$. Furthermore, if $\Gamma \vdash \Sigma \leqslant \exists\overline{\alpha}.\Sigma' \uparrow \overline{\tau}_2$, then $\overline{\tau}_1 = \overline{\tau}_2$.

Proof
Part 1 is by easy induction on the size of the derivation of the lookup. Part 2 follows by induction on the length of $\overline{\alpha}$. When $\overline{\alpha}$ is empty, then there is nothing to show. Otherwise, $\overline{\alpha} = \alpha, \overline{\alpha}'$ and $\overline{\tau}_1 = \tau_1, \overline{\tau}'_1$, such that $\text{lookup}_\alpha(\Sigma, \Sigma') \uparrow \tau_1$ and $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}'_1$. Let $\Gamma' = \Gamma, \overline{\alpha}'$. With weakening, respectively reordering, $\Gamma' \vdash \Sigma : \Omega$ and $\Gamma', \alpha \vdash \Sigma' : \Omega$. By part 1, we then know $\Gamma' \vdash \tau_1 : \kappa_\alpha$. Lemma 5.4 implies $\text{fv}(\tau_1) \subseteq \text{fv}(\Sigma)$, and because Σ is well-formed under Γ, it follows that $\text{fv}(\tau_1) \subseteq \text{dom}(\Gamma)$, so that we can strengthen to $\Gamma \vdash \tau_1 : \kappa_\alpha$. Substitution yields $\Gamma' \vdash \Sigma'[\tau_1/\alpha] : \Omega$, and from Lemma 5.4 we get $\text{lookup}_{\overline{\alpha}'}(\Sigma, \Sigma'[\tau_1/\alpha]) \uparrow \overline{\tau}'_1$, such that we can apply the induction hypothesis to conclude $\overline{\Gamma \vdash \tau'_1 : \kappa_{\alpha'}}$.

Furthermore, in order to prove the type equivalence, we first invert U-MATCH to reveal $\Gamma \vdash \Sigma \leqslant \Sigma'[\overline{\tau}_2/\overline{\alpha}]$ and $\overline{\Gamma \vdash \tau_2 : \kappa_\alpha}$. Consequently, $\overline{\tau}_2 = \tau_2, \overline{\tau}'_2$ and $\text{fv}(\overline{\tau}_2) \subseteq \text{dom}(\Gamma)$, *i.e.*, $\overline{\alpha} \cap \text{fv}(\overline{\tau}_2) = \emptyset$ by the usual conventions. The latter implies $\Sigma'[\overline{\tau}_2/\overline{\alpha}] = \Sigma'[\tau_2/\alpha][\overline{\tau}'_2/\overline{\alpha}'] = \Sigma'[\overline{\tau}'_2/\overline{\alpha}'][\tau_2/\alpha]$. Similar to before, Lemma 5.4 gets us $\text{lookup}_\alpha(\Sigma, \Sigma'[\overline{\tau}'_2/\overline{\alpha}']) \uparrow \tau_1$, and substitution $\Gamma, \alpha \vdash \Sigma'[\overline{\tau}'_2/\overline{\alpha}'] : \Omega$. By part 1, $\tau_1 = \tau_2$ then. To invoke the induction hypothesis for concluding $\overline{\tau}'_1 = \overline{\tau}'_2$ as well, we first note that by substitution, $\Gamma' \vdash \Sigma'[\tau_2/\alpha] : \Omega$, and second, by Lemma 5.4 again, $\text{lookup}_{\overline{\alpha}'}(\Sigma, \Sigma'[\tau_2/\alpha]) \uparrow \overline{\tau}'_1$. Third, since $\Sigma'[\overline{\tau}_2/\overline{\alpha}] = \Sigma'[\overline{\tau}'_2/\overline{\alpha}'][\tau_2/\alpha]$, we can construct a derivation for $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}'.\Sigma'[\tau_2/\alpha] \uparrow \overline{\tau}'_2$ with rule U-MATCH. □

According to soundness, if there is any type at all that makes a match succeed, then lookup can only deliver a well-formed, equivalent type. Despite being non-deterministic, the result of lookup hence is unique:

Corollary 5.6 (*Uniqueness of type lookup*)
Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma \vdash \exists \overline{\alpha}.\Sigma' : \Omega$ and $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau}$. If $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_1$ and $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_2$, then $\overline{\tau}_1 = \overline{\tau}_2$.

Because of this result, we can implement lookup as a deterministic algorithm by simply choosing the "first" root we encounter for each type variable, in any signature traversal order of our liking.

For explicit signatures, our definition of type lookup is also a complete algorithm for finding instantiations in the matching judgment:

Theorem 5.7 (*Completeness of type lookup*)
Assume $\exists \overline{\alpha}.\Sigma'$ explicit.

1. If $\Gamma \vdash \Sigma \leqslant \Sigma'[\overline{\tau}/\overline{\alpha}]$ and $\alpha \in \overline{\alpha}$, then $\text{lookup}_\alpha(\Sigma, \Sigma') \uparrow \alpha[\overline{\tau}/\overline{\alpha}]$.
2. If $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau}$, then $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$.

Proof
Explicitness of $\exists \overline{\alpha}.\Sigma'$ implies $\overline{\alpha}$ rooted in Σ', which in turn implies $\overline{\alpha \text{ rooted in } \Sigma'}$. Part 1 is then proved by simple induction on the derivation of α rooted in Σ'. Part 2 follows as a straightforward corollary. □

Note that this proof relies on the ability of the lookup algorithm to non-deterministically pick the root at the same path that was used in the respective derivation of α rooted in Σ'. Combined with uniqueness we know that any other path — and thus a deterministic choice — would work as well. Which gives us:

Corollary 5.8 (*Decidability of matching*)
Assume that Γ is valid and well-formed, and $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f$ is decidable for types well-formed under Γ. If Σ valid and Ξ explicit, and both are well-formed under Γ, then $\Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f$ is decidable (and does not actually require checking well-formedness of types).

This result follows directly, because subtyping and matching is defined by induction on the structure of the semantic signatures, and this structure remains fixed under type substitution, as performed in rules U-MATCH and U-FUNCT. (We do not need to check the well-formedness of $\overline{\tau}$ in U-MATCH because via Lemma 5.4, it is a consequence of looking up the types in the well-formed signature Σ.)

From there, decidability of elaboration follows because, up to matching, elaboration is syntax-directed:

Corollary 5.9 (*Decidability of elaboration*)
Under valid and well-formed Γ, provided we can (simultaneously) show that core elaboration is decidable, all judgments of module elaboration are decidable as well.

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 that it is a preorder. These properties are not actually relevant for soundness or decidability of the basic language, but they provide a sanity check that the language we are defining actually makes sense. They are also relevant to our translation of modules as first-class values (Section 6), and for the meta-theory of applicative functors (Section 9).

One complication in stating the following properties is that subtyping is defined in terms of the core language subtyping judgment $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow e$. Most of the properties only hold if we assume that the analogous property can be shown for that judgment. To avoid clumsy repetition, we leave this assumption implicit in the theorem statements.

First, we need a couple of technical lemmas stating that subtyping is stable under weakening and substitution:

Lemma 5.10 (*Subtyping under weakening*)
Let $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash \square$.

1. If $\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f$, then $\Gamma' \vdash \Xi \leqslant \Xi' \rightsquigarrow f$.
2. If $\Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f$, then $\Gamma' \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f$.

(Moreover, the derivations have the same size, up to core language judgments.)

Lemma 5.11 (*Subtyping under substitution*)
Let $\overline{\Gamma \vdash \tau : \kappa_\alpha}$.

1. If $\Gamma, \overline{\alpha} \vdash \Xi \leqslant \Xi' \rightsquigarrow f$, then $\Gamma \vdash \Xi[\overline{\tau}/\overline{\alpha}] \leqslant \Xi'[\overline{\tau}/\overline{\alpha}] \rightsquigarrow f[\overline{\tau}/\overline{\alpha}]$.
2. If $\Gamma, \overline{\alpha} \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau}' \rightsquigarrow f$, then $\Gamma \vdash \Sigma[\overline{\tau}/\overline{\alpha}] \leqslant \Xi[\overline{\tau}/\overline{\alpha}] \uparrow \overline{\tau}'[\overline{\tau}/\overline{\alpha}] \rightsquigarrow f[\overline{\tau}/\overline{\alpha}]$.

(Moreover, the derivations have the same size, up to core language judgments.)

Now for the actual theorems:

Theorem 5.12 (*Reflexivity of subtyping and matching*)

1. If $\Gamma \vdash \Xi : \Omega$, then $\Gamma \vdash \Xi \leqslant \Xi \rightsquigarrow f$.
2. If $\Gamma, \overline{\alpha} \vdash \Sigma : \Omega$, then $\Gamma, \overline{\alpha} \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma \uparrow \overline{\alpha} \rightsquigarrow f$.

Proof
By simultaneous induction on the structure of Ξ and Σ, respectively. □

Theorem 5.13 (*Transitivity of subtyping and matching*)

1. If $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$ and $\Gamma \vdash \Xi'' : \Omega$ and $\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f'$ and $\Gamma \vdash \Xi' \leqslant \Xi'' \rightsquigarrow f''$, then $\Gamma \vdash \Xi \leqslant \Xi'' \rightsquigarrow f$.
2. If $\Gamma \vdash \Sigma : \Omega$ and $\Gamma \vdash \exists \overline{\alpha}'.\Sigma' : \Omega$ and $\Gamma \vdash \exists \overline{\alpha}''.\Sigma'' : \Omega$, and $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}'.\Sigma' \uparrow \overline{\tau}' \rightsquigarrow f'$ and $\Gamma, \overline{\alpha}' \vdash \Sigma' \leqslant \exists \overline{\alpha}''.\Sigma'' \uparrow \overline{\tau}'' \rightsquigarrow f''$, then $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}''.\Sigma'' \uparrow \overline{\tau} \rightsquigarrow f$.

Proof
Since matching is syntax-directed, the proofs are a relatively straightforward simultaneous induction on the cumulative size of the subtyping/matching derivations (up to core language rules). In part (2), we need to apply the above substitution lemma. □

A further property one might expect from a subtyping relation is anti-symmetry, *i.e.,* if $\Xi \leqslant \Xi'$ and $\Xi' \leqslant \Xi$ (which we will abbreviate as $\Xi \leqslant\!\geqslant \Xi'$), then $\Xi = \Xi'$. This does not hold directly in our system, because the ordering of quantified variables might differ. We defer discussion of anti-symmetry to the next section, where we will prove it in a slight variation.

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 the module system to make modules first-class leads to undecidable typechecking (Lillibridge, 1997). 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 (Russo, 2000). In fact, in our setting, the extension is almost trivial.

Syntax. Figure 20 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 Russo (2000).)

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

$$\begin{array}{lllll} \text{(types)} & T & ::= & \ldots \mid \textbf{pack}\ S \\ \text{(expressions)} & E & ::= & \ldots \mid \textbf{pack}\ M{:}S \\ \text{(modules)} & M & ::= & \ldots \mid \textbf{unpack}\ E{:}S \end{array}$$

Fig. 20. Extension with modules as first-class values.

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

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

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

$$\frac{\Gamma \vdash M : \Xi' \rightsquigarrow e \qquad \Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash \Xi' \leqslant \mathrm{norm}(\Xi) \rightsquigarrow f}{\Gamma \vdash \textbf{pack}\ M{:}S : \mathrm{norm}(\Xi) \rightsquigarrow f\ e}\ \textsc{E-pack}$$

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

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

Fig. 21. Elaboration of modules as first-class values.

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 typechecking 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 21 gives the corresponding elaboration rules. Let us ignore the use of *signature normalization* $\mathrm{norm}(\Xi)$ 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 term 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.

Signature normalization. So what is the business with normalization? Unfortunately, were we to just use an unadulterated signature to directly represent its corresponding package type, the typing of packaged modules would become overly restrictive. 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
```

Intuitively, the signatures A and B are equivalent, and in fact, their semantic representations are mutual subtypes. But these representations will not actually be equivalent System F_ω types — A elaborates to $\exists\alpha_1\alpha_2.\{t : [= \alpha_1 : \Omega], u : [= \alpha_2 : \Omega]\}$ and B to $\exists\alpha_2\alpha_1.\{t : [= \alpha_1 : \Omega], u : [= \alpha_2 : \Omega]\}$ according to our rules (cf. Figure 11). 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_ω types in order to be compatible. In that case, the order matters. So the definition of g above would not typecheck.

To compensate, our elaboration must ensure that two package types pack S_1 and pack S_2 translate to equivalent F_ω types whenever S_1 and S_2 are mutual subtypes. Toward this end, we employ the normalization function defined in Figure 22. 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. In order to make this well defined, we impose a fixed but arbitrary total ordering on the set of labels l, which we extend to a lexicographical order on lists $\bar{l}$ of labels. Further, we assume a meta-function $\text{sort}_\leqslant$ which sorts its argument vector according to the given (total) order $\leqslant$. We instantiate it with an ordering $\alpha_1 \leqslant_\Sigma \alpha_2$ on type variables (also defined in Figure 22) according to their "first" occurrence as a root in Σ — expressed by reference to the "(at $\bar{l}$)" part of the rootedness judgment.

Note that normalization is defined only for explicit signatures (Section 5.2), where every variable is rooted. However, that is fine because we only need to normalize the representations of signatures appearing as annotations on pack or unpack. In the base case of atomic value signatures $[\tau]$, we assume that a similar normalization function $\text{norm}_{\text{core}}(\tau)$ exists for normalizing core-level types according to core-level subtyping $\Gamma \vdash \tau \leqslant \tau'$. (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 6.1 (*Signature normalization*)
Assume $\text{fv}(\text{norm}_{\text{core}}(\tau)) = \text{fv}(\tau)$ and $\text{norm}_{\text{core}}(\tau'[\overline{\tau}/\overline{\alpha}]) = \text{norm}_{\text{core}}(\tau')[\overline{\tau}/\overline{\alpha}]$. Then:

1. $\text{fv}(\text{norm}(\Xi)) = \text{fv}(\Xi)$.
2. $\text{norm}(\Xi[\overline{\tau}/\overline{\alpha}]) = \text{norm}(\Xi)[\overline{\tau}/\overline{\alpha}]$.
3. If Ξ explicit, then $\text{norm}(\Xi)$ explicit.
4. If $\Gamma \vdash \Xi : \Omega$, then $\Gamma \vdash \text{norm}(\Xi) : \Omega$.
5. If Ξ explicit, then $\Gamma \vdash \Xi \leqslant\geqslant \text{norm}(\Xi)$.

The main result regarding normalization, then, is a form of anti-symmetry for subtyping. But first, a technical lemma that we need for the proof. It effectively says that two abstract signatures mutually matching each other quantify, up to reordering and renaming, the same abstract type variables.

$$
\begin{array}{lcll}
\mathrm{norm}([\tau]) & = & [\mathrm{norm}_{\mathrm{core}}(\tau)] & \\
\mathrm{norm}([= \tau : \kappa]) & = & [= \tau : \kappa] & \\
\mathrm{norm}([= \Xi]) & = & [= \mathrm{norm}(\Xi)] & \\
\mathrm{norm}(\{\overline{l : \Sigma}\}) & = & \{\overline{l : \mathrm{norm}(\Sigma)}\} & \\
\mathrm{norm}(\forall \overline{\alpha}.\Sigma \to \Xi) & = & \forall \overline{\alpha}'.\, \mathrm{norm}(\Sigma) \to \mathrm{norm}(\Xi) & \text{where } \overline{\alpha}' = \mathrm{sort}_{\leqslant_{\mathrm{norm}(\Sigma)}}(\overline{\alpha}) \\
\mathrm{norm}(\exists \overline{\alpha}.\Sigma) & = & \exists \overline{\alpha}'.\, \mathrm{norm}(\Sigma) & \text{where } \overline{\alpha}' = \mathrm{sort}_{\leqslant_{\mathrm{norm}(\Sigma)}}(\overline{\alpha}) \\
\alpha_1 \leqslant_\Sigma \alpha_2 & \Leftrightarrow & \min\{\bar{l} \mid \alpha_1 \text{ rooted in } \Sigma \text{ (at } \bar{l})\} \leqslant \min\{\bar{l} \mid \alpha_2 \text{ rooted in } \Sigma \text{ (at } \bar{l})\} &
\end{array}
$$

Fig. 22. Signature normalization.

Lemma 6.2 (*Mutual matching*)
Suppose $\overline{\alpha}$ rooted in Σ and $\overline{\alpha}'$ rooted in Σ'. Moreover, $\overline{\alpha} \cap \mathrm{fv}(\overline{\tau}) = \overline{\alpha}' \cap \mathrm{fv}(\overline{\tau}') = \emptyset$. If $\Gamma, \overline{\alpha} \vdash \Sigma \leqslant \Sigma'[\overline{\tau}'/\overline{\alpha}']$ and inversely, $\Gamma, \overline{\alpha}' \vdash \Sigma' \leqslant \Sigma[\overline{\tau}/\overline{\alpha}]$, then $[\overline{\tau}/\overline{\alpha}] = [\overline{\tau}'/\overline{\alpha}']^{-1}$, *i.e.,* $|\overline{\alpha}| = |\overline{\alpha}'|$, and there is a reordering $\overline{\alpha}''$ of $\overline{\alpha}'$, and a corresponding reordering $\overline{\tau}''$ of $\overline{\tau}'$, such that $\overline{\tau} = \overline{\alpha}''$ and $\overline{\tau}'' = \overline{\alpha}$.

Proof
For every $\alpha' \in \overline{\alpha}'$, we can show by induction on its rootedness derivation that there are atomic type signatures with $\Gamma, \overline{\alpha} \vdash [= \tau_0 : \kappa] \leqslant [= \alpha'[\overline{\tau}'/\overline{\alpha}'] : \kappa]$, and conversely, $\Gamma, \overline{\alpha}' \vdash [= \alpha' : \kappa] \leqslant [= \tau_0[\overline{\tau}/\overline{\alpha}] : \kappa]$. By inverting those subtypings, $\tau_0 = \alpha'[\overline{\tau}'/\overline{\alpha}']$, and at the same time $\alpha' = \tau_0[\overline{\tau}/\overline{\alpha}]$. That is, $\alpha' = \alpha'[\overline{\tau}'/\overline{\alpha}'][\overline{\tau}/\overline{\alpha}]$. Since $\alpha' \in \overline{\alpha}'$, there is a corresponding $\tau' \in \overline{\tau}'$, such that $\alpha' = \tau'[\overline{\tau}/\overline{\alpha}]$. Because $\tau' \neq \alpha'$ according to the assumptions about $\mathrm{fv}(\overline{\tau}')$, there has to be an $\alpha \in \overline{\alpha}$, such that $\tau' = \alpha$ and $\alpha[\overline{\tau}/\overline{\alpha}] = \alpha'$. We can prove the same for every other $\alpha' \in \overline{\alpha}'$. Consequently, because all $\overline{\alpha}'$ are distinct, all $\overline{\tau}'$ have to be distinct, too, and thus $|\overline{\alpha}| \geqslant |\overline{\alpha}'|$. By symmetry, *i.e.,* exchanging roles and repeating the argument, we obtain that both substitutions have the same cardinality and are mutual inverses. □

Theorem 6.3 (*Anti-symmetry of subtyping up to normalization*)
Let $\Gamma \vdash \Xi : \Omega$ explicit and $\Gamma \vdash \Xi' : \Omega$ explicit. Furthermore, assume that if $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$ and $\Gamma \vdash \tau \leqslant\!\geqslant \tau'$, then $\mathrm{norm}_{\mathrm{core}}(\tau) = \mathrm{norm}_{\mathrm{core}}(\tau')$. Then, if both $\Gamma \vdash \Xi \leqslant \Xi'$ and $\Gamma \vdash \Xi' \leqslant \Xi$, it holds that $\mathrm{norm}(\Xi) = \mathrm{norm}(\Xi')$.

Proof
By induction on the (size of the) derivations. In the cases of rules U-ABS and U-FUNCT, invert the matching premise and apply the previous lemma to reveal that the quantified variables are equivalent up to reordering (and α-renaming). Hence, we can assume (after α-renaming) that both inner signatures are well-formed under the same extension of Γ, and apply the induction hypothesis to know that their normalization are equal. Since sorting of the variables is independent of the original quantifier order as well, it also produces the same result for both sides. □

By normalizing semantic signatures in all places where they are used as package types, we hence establish the desired property that the intuitive notion of signature equivalence coincides with type equivalence. By applying the coercion f in the rule for **pack**, we also ensure that the representation of the module itself is normalized accordingly.

Soundness. The package semantics is so simple that soundness is an entirely straightforward property.

Theorem 6.4 (*Soundness of elaboration with packages*)
Theorem 5.1 still holds with the additional rules from Figure 21.

Proof
By simultaneous induction on derivations. The existing cases are all proved as before; the new ones are straightforward given Lemma 6.1. □

Our decidability result (Corollary 5.9) is not affected by the addition of modules as first-class values, because it only hinged on the decidability of signature matching.

6.1 A note on first-class modules

Given that our elaboration of modules as first-class values does not actually do much, the reader may be puzzled why it is allegedly so much harder to go the whole way and make modules truly first-class. Can't we just merge the module and core levels into one unified language? For some constructs, such as conditionals, this would probably require type annotations to maintain principal types, and ML-style type inference certainly would not work anymore. But those are limitations that other languages with subtyping (especially object-oriented ones) have always been comfortable with. In the ML module literature, however, it has been frequently claimed that first-class modules result in undecidable typechecking (Lillibridge, 1997), so surely there must be more fundamental problems. What, specifically, would break in the F-ing approach?

A move to first-class modules means collapsing module and term language, as well as signature and type language. Because types can be denoted by type variables, the latter would imply that signatures can then also be denoted by type variables. Our elaboration, on the other hand, is dependent on one fundamental property: for any signature occurring in the rules, the number of abstract types it declares — *i.e.,* the number of quantifiers — is known statically *and* stable under substitution. If this were not the case, then we could not perform the implicit lifting (or "monadic" binding) of existentials that is so central to our approach. Clearly, if we allowed for type variables as signatures, it would no longer work.

Moreover, as Lillibridge (1997) showed, we would lose decidability of subtyping. Looking at our subtyping rules, they substitute type variables along the way. With type variables possibly representing signatures, substitution could change the structure of the signatures we are looking at. Consequently, the subtyping rules would no longer describe an *algorithm* that is inductive on the structure of signatures, and (backwards) application of the rules might indeed diverge (see Lillibridge (1997) for an example). That is, the argument we made regarding Corollary 5.8 (decidability of matching) would no longer hold.

The sort of "predicativity" restriction that results from separating types and signatures (*i.e.,* signatures can only abstract over types, not other signatures) is thus crucial to maintaining decidability of typechecking. It is the real essence of the

```
val p1 = pack {type t = int; val v = 6} : {type t; val v : t}
val p2 = pack {type t = bool; val v = true} : {type t; val v : t}
module Flip = fun X : {} ⇒ unpack (if random() then p1 else p2) : {type t; val v : t}
```

Fig. 23. Example: a statically impure functor.

core/module-language stratification in ML. Without it, the F-ing approach would not work — nor are we aware of any other decidable type system for ML-style modules without a similar limitation.

The same problems would arise if we were to add abstract signature declarations of the form **signature** *X* to the language. Indeed, it is the presence of this additional feature that tips the scales and renders OCaml's module typechecking undecidable (Rossberg, 1999).

7 Applicative functors and static purity

The semantics for functors that we have presented so far follows Standard ML, in that functors are *generative*: if a functor body defines any abstract types, then those types are effectively "generated" anew each time the functor is applied. OCaml employs an alternative, so-called *applicative* semantics for functors, by which a functor will return equivalent types whenever it is applied to the same argument. For example, consider the following use of the Set functor (cf. Figure 3):

```
module IntOrd = {type t = int; val eq = Int.eq; val less = Int.less}
module Set1 = Set IntOrd
module Set2 = Set IntOrd
val s = Set1.add (7, Set2.empty).
```

The last line in this example does not typecheck under generative semantics, because each application of Set yields a "fresh" set type, such that Set_1.set and Set_2.set differ. Under applicative semantics, however, the example *would* typecheck, because the two structures are created by equivalent module applications.

The applicative functor semantics enables the typechecker to recognize that abstract data types generated in different parts of a program are in fact the same type. This is particularly useful when working with functors that implement generic data structures (*e.g.,* sets), but it also supports a more flexible treatment of higher-order functors. For more details about these motivating applications, see Leroy (1995).

Unfortunately, applicative functor semantics is also significantly subtler than generative semantics, and much harder to get right. In particular, there are two major problems:

Type safety: For a functor to be safely given an applicative semantics, it must *at a minimum* satisfy the property that the type components in its body are guaranteed to be implemented in the same way every time the functor is applied to the same argument. In the presence of modules as first-class values (Section 6), this property is not universally satisfied. For example, consider the functor Flip in Figure 23. The first time this functor is applied, it may return a module whose type component t is implemented internally as int, whereas the second time t may

be implemented as bool. It is thus utterly *unsound* (*i.e.,* breaks type safety) to give a functor like Flip an applicative semantics.

Abstraction safety: Even if the type components of a functor are implemented in the same way every time it is applied, treating the functor as applicative may nevertheless constitute a violation of *data abstraction*. That is, for some abstract data types implemented by a functor, applicative semantics breaks the ability to establish representation invariants locally. We will discuss this problem in more detail and see examples in Section 8.

Concerning the first of these two problems, both Moscow ML and (more recently) OCaml provide packaged modules *and* applicative functors, and circumvent the soundness problem only by imposing severe (and rather unsatisfactory) restrictions on the unpacking construct, namely prohibiting its use within functor bodies. In this section, we focus on the first problem and show how to address it properly within the F-ing modules framework. The second problem will be explored in Section 8.

7.1 Understanding applicativity versus generativity in terms of purity

For the purpose of ensuring type safety, the key thing is to ensure that we only project type components out of module expressions whose type components are statically well-determined. Following Dreyer (2005), we refer to such expressions as *statically pure*, which for the remainder of this section we will just shorten to *pure*. (We will consider the role of *dynamic purity* in Section 8.)

In our module language, the expression that introduces static impurity is the unpack E:S construct: the type components of the unpacked module depend essentially on the *term* E, a term which may have computational effects that lead it to produce values with different type components every time it is evaluated. If an unpacked module appears in the body of a functor, the functor will encapsulate the impurity.

Thus, we need to distinguish between *pure functors* and *impure functors*. And it is precisely the pure ones that may behave applicatively, while the impure ones have to behave generatively. Hence, from here on, when talking about functors, we will use "applicative" interchangeably with "pure", and "generative" interchangeably with "impure". (In fact, the correspondence is so natural and intuitive that we are tempted to retire the "applicative" versus "generative" terminology altogether. For historic reasons, however, we will continue to use the traditional terms in the remainder of this article.)

One important point of note: in the case where E is a value (or more generally, free of effects), it would seem that there is nothing unsafe about projecting type components from unpack E:S, since each unpacking will produce modules with the same underlying type components. The trouble with permitting unpack E:S to be treated as statically pure — even in this case — is that, while its type components are well-determined, they are not *statically* well-determined. In the parlance of Harper *et al.* (1990), unpack E:S does not obey *phase separation* because the identity of its type components may depend on the dynamic instantiation of the free (term) variables of E. As a result, supporting projection from unpack E:S would require

$$\text{(signatures)} \quad S \quad ::= \quad \ldots \mid (X{:}S) \Rightarrow S$$

Fig. 24. Extending the syntax of the module language with applicative functor signatures.

full-blown value-dependent types, which we would like to avoid for a variety of pragmatic reasons. The F-ing modules approach, by virtue of its interpretation into the non-dependently typed F_ω, has the benefit of providing automatic enforcement of phase separation, and thus prohibits projection from unpack $E{:}S$.

7.2 Extending the language

In order to distinguish between pure (a.k.a. applicative) and impure (a.k.a. generative) when *specifying* a functor — *e.g.,* in a higher-order setting — we extend the syntax of the external language of signatures with a new form of functor signature, shown in Figure 24. While the original form retains its meaning for specifying impure functors, the new one specifies pure ones. For example, the (pure) Set functor matches the pure functor signature (X : ORD) ⇒ SET, while the (impure) Flip functor will only match the impure signature (X : {}) → {**type** t; **val** v : t}. That said, Set will also continue to match the impure signature (X : ORD) → SET, because pure (applicative) functor signatures are treated as subtypes of impure (generative) ones.

One defining feature of applicative functors is the ability to project types from module paths containing functor applications. For example, given the familiar pure Set functor, (Set IntOrd).set should be a valid type expression, because every application of Set returns the same type. Since our syntax of paths P has been maximally general from the outset, it readily allows such types to be written. In fact, we will see shortly that the existing semantics for paths does not need to change much in order to encompass functor applications.

7.3 Elaboration

The addition of applicative functors, along with the attendant tracking of purity, requires some significant changes to elaboration. We will walk through those changes starting with the simple parts.

Semantic signatures. The main difference between a generative and an applicative functor is the point at which the abstract type components in their bodies get created, and this difference is reflected quite clearly in the placement of existential quantifiers in their semantic signatures. A generative functor has an F_ω type of the form $\forall\overline{\alpha}_1.\Sigma_1 \to \exists\overline{\alpha}_2.\Sigma_2$. Applying such a functor produces an existential package, which must be explicitly unpacked in order to get access to the type components of the package; however, due to the closed-scope nature of existential unpacking, there is no way to associate those type components with the existential package (and thus the generative functor) itself. In contrast, following Russo (1998), we will describe applicative functors with F_ω types of the form $\exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\Sigma_1 \to \Sigma_2$. Such signatures indicate that the existential package is constructed only once, when the functor

$$
\begin{array}{lll}
\text{(effects)} & \varphi & ::= \quad \mathtt{I} \mid \mathtt{P} \\
\text{(concrete signatures)} & \Sigma & ::= \quad \forall\overline{\alpha}.\Sigma \to_{\mathtt{I}} \Xi \mid \forall\overline{\alpha}.\Sigma \to_{\mathtt{P}} \Sigma \mid \ldots
\end{array}
$$

Notation:

$$
\begin{array}{llllll}
\varphi \vee \varphi & := & \varphi & & & \\
\mathtt{I} \vee \mathtt{P} & := & \mathtt{P} \vee \mathtt{I} & := & \mathtt{I} &
\end{array}
$$

Abbreviations:

$$
\begin{array}{lllll}
\text{(types)} & \tau_1 \to_\varphi \tau_2 & := & \tau_1 \to \{l_\varphi : \tau_2\} \\
\text{(expressions)} & \lambda_\varphi x{:}\tau.e & := & \lambda x{:}\tau.\{l_\varphi = e\} \\
 & (e_1\ e_2)_\varphi & := & (e_1\ e_2).l_\varphi
\end{array}
$$

Fig. 25. (Colour online) Semantic signatures for applicative functors.

is *defined*, not every time it is applied, thus enabling the abstract types $\overline{\alpha}_2$ to be associated with the functor itself. The return type of an applicative functor is always a concrete signature Σ_2, with no local existential variables.

Consequently, the introduction of applicative functors does not require any significant change to our definition of semantic signatures — our existing notion of abstract signature Ξ already subsumes the kind of quantification that expresses an applicative functor! We merely extend functor signatures with a simple effect annotation. As defined in Figure 25, an effect φ can either be pure (P) or impure (I). These form a trivial two-point lattice with $\mathtt{P} < \mathtt{I}$, and there is a straightforward definition of join ($\vee$) on effect annotations (we won't need meet). To encode effect annotations in our F_ω representation of functors, we assume that there are two distinct record labels $l_\mathtt{P}$ and $l_\mathtt{I}$.

The important point, though, is that a pure functor type may only have a concrete result signature Σ, which is why we give it as a separate production in the syntax of Σ in Figure 25. Nevertheless, we will often write $\forall\overline{\alpha}.\Sigma \to_\varphi \Xi$ to range over both kinds of functor signature, implicitly understanding that Ξ has to be a concrete Σ' when $\varphi = \mathtt{P}$.

Signature elaboration. Figure 26 shows the new elaboration rules for dealing with functor signatures (we have highlighted the differences from the original rules from Figure 11). The rule S-FUNCT-I for impure functor signatures leaves the original rule S-FUNCT almost unchanged, except for adding the effect annotation I on the signature in the conclusion.

In order to match the description of applicative functor signatures we just gave, the new rule S-FUNCT-P for applicative functors must produce a signature where all existential quantifiers are "lifted" out of the functor type. It does so by replacing the original $\overline{\alpha}_2$ inferred for the result signature with fresh $\overline{\alpha}'_2$ that are quantified *outside* the functor signature.

But abstract types defined inside a functor might have functional dependencies on the functor's parameters. The trick, discovered by Biswas (1995) and Russo (1998), is to capture such potential dependencies by skolemizing the lifted variables over the universally quantified types from the functor's parameter. That is, we raise the kind of each of the $\overline{\alpha}'_2$ so as to generalize it over all the type parameters $\overline{\alpha}_1$;

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

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2}{\Gamma \vdash (X{:}S_1) \to S_2 \rightsquigarrow \forall \overline{\alpha}_1.\, \Sigma_1 \to_{\mathrm{I}} \exists \overline{\alpha}_2.\Sigma_2}\ \text{S-FUNCT-I}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2 \qquad \overline{\kappa_{\alpha'_2} = \kappa_{\alpha_1} \to \kappa_{\alpha_2}}}{\Gamma \vdash (X{:}S_1) \Rightarrow S_2 \rightsquigarrow \exists \overline{\alpha}'_2.\forall \overline{\alpha}_1.\, \Sigma_1 \to_{\mathrm{P}} \Sigma_2[\overline{\alpha'_2\ \alpha_1/\alpha_2}]}\ \text{S-FUNCT-P}$$

Subtyping $\boxed{\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f}$

$$\frac{\Gamma, \overline{\alpha}' \vdash \Sigma' \leqslant \exists \overline{\alpha}.\Sigma \uparrow \overline{\tau} \rightsquigarrow f_1 \qquad \Gamma, \overline{\alpha}' \vdash \Xi[\overline{\tau}/\overline{\alpha}] \leqslant \Xi' \rightsquigarrow f_2 \qquad \varphi \leqslant \varphi'}{\Gamma \vdash (\forall \overline{\alpha}.\Sigma \to_{\varphi} \Xi) \leqslant (\forall \overline{\alpha}'.\Sigma' \to_{\varphi'} \Xi') \rightsquigarrow \lambda f{:}(\forall \overline{\alpha}.\Sigma \to_{\varphi} \Xi).\lambda \overline{\alpha}'.\, \lambda_{\varphi'} x{:}\Sigma'.\, f_2(f\ \overline{\tau}\ (f_1\ x))_{\varphi}}\ \text{U-FUNCT}$$

Subeffects $\boxed{\varphi \leqslant \varphi'}$

$$\frac{}{\varphi \leqslant \varphi}\ \text{F-REFL} \qquad \frac{}{\mathtt{P} \leqslant \mathtt{I}}\ \text{F-SUB}$$

Fig. 26. (Colour online) New rules for applicative functor signatures.

```
(Elem : ORD) ⇒ (SET where type t = Elem.t)
 ⇝ ∃β:(Ω →Ω).
        ∀α:Ω.{t : [= α : Ω],
              eq : [α × α → bool],
              less : [α × α → bool]}
             →P {set : [= β α : Ω],
                 elem : [= α : Ω],
                 empty : [β α],
                 add : [α × β α → β α],
                 mem : [α × β α → bool]}
```

Fig. 27. (Colour online) Example: applicative signature elaboration.

correspondingly, all occurrences of an $\alpha \in \overline{\alpha}_2$ are substituted by the application of the corresponding $\alpha' \in \overline{\alpha}'_2$ to the actual parameter vector $\overline{\alpha}_1$. (At this point, clearly, we require not just System F, but the full power of F_ω, to model our semantics.)

To better understand what's going on here, let us revisit the signature of the Set functor (cf. Figure 12), and its elaboration into a semantic signature. Figure 27 shows how the analogous *applicative* functor signature will be represented semantically. The new elaboration rule places the existential quantifier for β outside the functor, and it raises the original kind Ω of β to $\Omega \to \Omega$, in order to reflect the functional dependency on α. Everywhere we originally had a β, we now find $\beta\,\alpha$ in the result.

Where such a functor is later applied, β remains as is; only α gets substituted by the concrete argument type. If that is, say, int, then the resulting structure signature will equate the type set to β int. Any further application of the functor to arguments with a type component t = int will yield the same type set = β int.

$$
\begin{array}{llll}
\text{(kinds)} & (\cdot) \to \kappa & := \kappa & \\
& (\Gamma, \alpha) \to \kappa & := \Gamma \to \kappa_\alpha \to \kappa & \\
& (\Gamma, x{:}\tau) \to \kappa & := \Gamma \to \kappa & \\
\text{(types)} & \lambda(\cdot).\tau' & := \tau' & \\
& \lambda(\Gamma, \alpha).\tau' & := \lambda\Gamma.\lambda\alpha.\tau' & \\
& \lambda(\Gamma, x{:}\tau).\tau' & := \lambda\Gamma.\tau' & \\
& \tau'\ (\cdot) & := \tau' & \\
& \tau'\ (\Gamma, \alpha) & := \tau'\ \Gamma\ \alpha & \\
& \tau'\ (\Gamma, x{:}\tau) & := \tau'\ \Gamma & \\
\text{(types)} & \forall(\cdot).\tau' & := \tau' & \\
& \forall(\Gamma, \alpha).\tau' & := \forall\Gamma.\forall\alpha.\tau' & \\
& \forall(\Gamma, x{:}\tau).\tau' & := \forall\Gamma.\tau \to_{\mathrm{P}} \tau' & \\
\text{(expressions)} & \lambda(\cdot).e & := e & \\
& \lambda(\Gamma, \alpha).e & := \lambda\Gamma.\lambda\alpha.e & \\
& \lambda(\Gamma, x{:}\tau).e & := \lambda\Gamma.\lambda_{\mathrm{P}} x{:}\tau.e & \\
& e\ (\cdot) & := e & \\
& e\ (\Gamma, \alpha) & := e\ \Gamma\ \alpha & \\
& e\ (\Gamma, x{:}\tau) & := (e\ \Gamma\ x)_{\mathrm{P}} &
\end{array}
$$

$$
\begin{array}{lll}
\Gamma^{\mathrm{I}} & := & \cdot \\
\Gamma^{\mathrm{P}} & := & \Gamma
\end{array}
$$

Fig. 28. Environment abstraction.

Subtyping. Because the definition of semantic signatures barely changed, only a minor extension is required to define functor subtyping, namely to allow pure functor types to be subtypes of impure ones. We do not need to change the definition of matching at all. Abstract types lifted from a functor body act as if they were abstract type constructors defined outside the functor, and the original matching rule (cf. Figure 13) handles them just fine. (However, an algorithmic implementation of the rules will require non-trivial extensions to the type lookup algorithm, as we will discuss in Section 9.2.)

In other words, the correct subtyping relation between applicative and generative functor signatures falls out almost for free. The F-ing method provides an immediate explanation of such subtyping and why it is sound.

Modules. The rule M-SEAL defined in Section 4, when used with an applicative functor signature, allows one to introduce applicative functor types. But the circumstances are limited: the definition of matching requires that the sealed functor may not itself contain any non-trivial sealing, because a functor creating abstract types would be considered generative, *i.e.,* impure, under the module elaboration rules from Section 4. Shao's system (Shao, 1999), which introduces applicative functor signatures solely through sealing, suffers from this limitation, a point we return to in Section 11. In contrast, the system we will present is designed to support sealing within applicative functors, a feature shared by all other accounts besides Shao's. That requires refining our module elaboration rules.

While signatures for applicative functors are (relatively) easy to elaborate, modules require more extensive changes to their elaboration rules to account for applicativity and purity. Superficially, the only extension to the module elaboration judgment is the inclusion of an effect annotation φ, which specifies whether the module is deemed pure or not. However, the invariants associated with pure and impure module elaboration are quite different from each other, as we explain below. Figure 29 gives the modified rules (we have again highlighted the changes relative to the original rules, cf. Figure 14).

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

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X :_{\mathrm{P}} \Sigma \rightsquigarrow \lambda\Gamma.X}\ \textsc{M-var} \qquad \frac{\Gamma \vdash B :_{\varphi} \Xi \rightsquigarrow e}{\Gamma \vdash \{B\} :_{\varphi} \Xi \rightsquigarrow e}\ \textsc{M-struct}$$

$$\frac{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\{l_X : \Sigma, \overline{l : \Sigma'}\} \rightsquigarrow e}{\Gamma \vdash M.X :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \lambda\Gamma^{\varphi}.(y\ \Gamma^{\varphi}).l_X\rangle}\ \textsc{M-dot}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash M :_{\mathrm{I}} \Xi \rightsquigarrow e}{\Gamma \vdash \mathbf{fun}\ X{:}S \Rightarrow M :_{\mathrm{P}} \forall\overline{\alpha}.\Sigma \to_{\mathrm{I}} \Xi \rightsquigarrow \lambda\Gamma.\lambda\overline{\alpha}.\lambda_{\mathrm{I}}X{:}\Sigma.e}\ \textsc{M-funct-I}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash M :_{\mathrm{P}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \mathbf{fun}\ X{:}S \Rightarrow M :_{\mathrm{P}} \exists\overline{\alpha}_2.\forall\overline{\alpha}.\Sigma \to_{\mathrm{P}} \Sigma_2 \rightsquigarrow e}\ \textsc{M-funct-P}$$

$$\frac{\Gamma(X_1) = \forall\overline{\alpha}.\Sigma_1 \to_{\varphi} \Xi \qquad \Gamma(X_2) = \Sigma_2 \qquad \Gamma \vdash \Sigma_2 \leqslant \exists\overline{\alpha}.\Sigma_1 \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X_1\ X_2 :_{\varphi} \Xi[\overline{\tau}/\overline{\alpha}] \rightsquigarrow \lambda\Gamma^{\varphi}.(X_1\ \overline{\tau}\ (f\ X_2))_{\varphi}}\ \textsc{M-app}$$

$$\frac{\Gamma(X) = \Sigma' \qquad \Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma \vdash \Sigma' \leqslant \exists\overline{\alpha}.\Sigma \uparrow \overline{\tau} \rightsquigarrow f \qquad \overline{\kappa_{\alpha'} = \Gamma \to \kappa_{\alpha}}}{\Gamma \vdash X :> S :_{\mathrm{P}} \exists\overline{\alpha'}.\Sigma[\overline{\alpha'\ \Gamma/\alpha}] \rightsquigarrow \mathsf{pack}\ \langle\overline{\lambda\Gamma.\tau}, \lambda\Gamma.f\ X\rangle}\ \textsc{M-seal}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash E : \mathrm{norm}(\Xi) \rightsquigarrow e}{\Gamma \vdash \mathbf{unpack}\ E{:}S :_{\mathrm{I}} \mathrm{norm}(\Xi) \rightsquigarrow e}\ \textsc{M-unpack}$$

Bindings $\boxed{\Gamma \vdash B :_{\varphi} \Xi \rightsquigarrow e}$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e}{\Gamma \vdash \mathbf{val}\ X{=}E :_{\mathrm{P}} \{l_X : [\tau]\} \rightsquigarrow \lambda\Gamma.\{l_X = [e]\}}\ \textsc{B-val}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \mathbf{type}\ X{=}T :_{\mathrm{P}} \{l_X : [= \tau : \kappa]\} \rightsquigarrow \lambda\Gamma.\{l_X = [\tau : \kappa]\}}\ \textsc{B-typ}$$

$$\frac{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e \qquad \Sigma\ \text{not atomic}}{\Gamma \vdash \mathbf{module}\ X{=}M :_{\varphi} \exists\overline{\alpha}.\{l_X : \Sigma\} \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \lambda\Gamma^{\varphi}.\{l_X = x\ \Gamma^{\varphi}\}\rangle}\ \textsc{B-mod}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \Xi}{\Gamma \vdash \mathbf{signature}\ X{=}S :_{\mathrm{P}} \{l_X : [= \Xi]\} \rightsquigarrow \lambda\Gamma.\{l_X = [\Xi]\}}\ \textsc{B-sig}$$

$$\frac{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\} \rightsquigarrow e}{\Gamma \vdash \mathbf{include}\ M :_{\varphi} \exists\overline{\alpha}.\{\overline{l_X : \Sigma}\} \rightsquigarrow e}\ \textsc{B-incl} \qquad \frac{}{\Gamma \vdash \epsilon :_{\mathrm{P}} \{\} \rightsquigarrow \lambda\Gamma.\{\}}\ \textsc{B-emt}$$

$$\frac{\begin{array}{rl}\Gamma \vdash B_1 :_{\varphi_1} \exists\overline{\alpha}_1.\{\overline{l_{X_1} : \Sigma_1}\} \rightsquigarrow e_1 & \overline{l'_{X_1} = l_{X_1} - l_{X_2}} \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash B_2 :_{\varphi_2} \exists\overline{\alpha}_2.\{\overline{l_{X_2} : \Sigma_2}\} \rightsquigarrow e_2 & \overline{l'_{X_1} : \Sigma'_1} \subseteq \overline{l_{X_1} : \Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash B_1;B_2 :_{\varphi_1 \vee \varphi_2} \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{l'_{X_1} : \Sigma'_1}, \overline{l_{X_2} : \Sigma_2}\} \\ \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in} \\ \quad \mathsf{unpack}\ \langle\overline{\alpha}_2, y_2\rangle = (\mathsf{let}\ \overline{X_1 = \lambda\Gamma^{\varphi_1 \vee \varphi_2}.(y_1\ \Gamma^{\varphi_1}).l_{X_1}}\ \mathsf{in}\ e_2)\ \mathsf{in} \\ \quad\quad \mathsf{pack}\ \langle\overline{\alpha}_1\overline{\alpha}_2,\ \lambda\Gamma^{\varphi_1 \vee \varphi_2}.\mathsf{let}\ \overline{X_1 = (y_1\ \Gamma^{\varphi_1}).l_{X_1}}\ \mathsf{in} \\ \quad\quad\quad\quad \mathsf{let}\ \overline{X_2 = (y_2\ (\Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1})^{\varphi_2}).l_{X_2}}\ \mathsf{in}\{\overline{l'_{X_1} = X_1}, \overline{l_{X_2} = X_2}\}\rangle\end{array}}\ \textsc{B-seq}$$

Fig. 29. (Colour online) New rules for applicative functors and modules.

Functors We begin by explaining how we handle functors, since this motivates the form and associated invariants of the module elaboration judgment. We now have two rules: M-FUNCT-I, which yields a generative functor (as before) if the body M is impure, and M-FUNCT-P, which yields an applicative functor if M is pure. In both cases, the functor expression itself is pure, because it is a *value* form that suspends any effects of M.

For applicative functors, we need to follow what we did for signatures, and implement $\exists$-lifting. The difficulty, though, is doing it in a way that still allows a compositional translation of sealing inside an applicative functor.

What is the problem? Consider the following example:

$$\textbf{fun}\ (\mathsf{X} : \{\textbf{type}\ \mathsf{t}\}) \Rightarrow \{\textbf{type}\ \mathsf{u} = \mathsf{X.t} \times \mathsf{X.t}\} :> \{\textbf{type}\ \mathsf{u}\}$$

If the body of this functor were impure (like the body of Flip from Figure 23), the impure functor rule M-FUNCT-I would delegate translation of the functor body to a subderivation, which, in this example, would yield a signature $\Xi = \exists\beta.\{\mathsf{u} : [= \beta : \Omega]\}$ and some term $e : \Xi$. We would then λ-abstract e over the functor argument to produce a function of type $\forall\alpha.\{\mathsf{t} : [= \alpha : \Omega]\} \to_{\mathrm{I}} \Xi$. Now, if we wanted to adapt this situation for pure functors by applying the same lifting trick we used for pure functor signatures, then we would have to somehow take $e : \Xi$ and retroactively lift its hidden type components over α to derive a term of type $\exists\beta' : \Omega \to \Omega.\forall\alpha : \Omega.\{\mathsf{t} : [= \alpha : \Omega]\} \to_{\mathrm{P}} \{\mathsf{u} : [= \beta'\alpha : \Omega]\}$. In general, such retroactive lifting is not possible.

To avoid this dilemma, we employ a different trick: we design the translation of a pure module (which the body of an applicative functor must be) so that it consistently constructs an existential package with the necessary lifting already built in!

In fact, for simplicity, the translation of a pure module abstracts over the *entire* environment Γ. More precisely, whereas the impure judgment $\Gamma \vdash M :_{\mathrm{I}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$ guarantees that $\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$, the pure judgment $\Gamma \vdash M :_{\mathrm{P}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$ instead guarantees that e is a *closed* term satisfying $\cdot \vdash e : \exists\overline{\alpha}.\forall\Gamma.\Sigma$, where the notation $\forall\Gamma.\Sigma$ is defined in Figure 28. This idea is borrowed from Shan (2004), who used a similar approach for a translation of the module calculus of Dreyer *et al.* (2003) into System F_ω.

The pure functor rule M-FUNCT-P then becomes fairly trivial: it just computes the translation of its body and returns that directly. This means the translation of the functor will not only abstract over the functor's parameters as required, but over the rest of the current environment Γ, too (because $\exists\overline{\alpha}_2.\forall(\Gamma, \overline{\alpha}, X{:}\Sigma).\Sigma_2$ is just an alternative way of writing $\exists\overline{\alpha}_2.\forall\Gamma.\forall\overline{\alpha}.\Sigma \to_{\mathrm{P}} \Sigma_2$). But that is fine, because the functor is itself a pure module, so according to the elaboration invariant for pure modules, it *has to* abstract over Γ anyway.

It turns out that the rule M-APP for functor application can remain largely unchanged — it can handle both kinds of functors. In both cases, the effect φ on the functor's type is unleashed and determines the effect of the application. Note that, applicative application is always degenerate with Ξ being some concrete signature Σ_3, so that there are no existential quantifiers in the result to lift over.

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

$$\frac{\Gamma \vdash P :_{\varphi} \exists \overline{\alpha}.\Sigma \rightsquigarrow e \qquad \Gamma \vdash \Sigma : \Omega}{\Gamma \vdash P : \Sigma \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ x\ \Gamma^{\varphi}}\ \textsc{P-mod}$$

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

$$\frac{\Gamma \vdash M :_{\varphi} \exists \overline{\alpha}.\Sigma \rightsquigarrow e \qquad \Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash \exists \overline{\alpha}.\Sigma \leqslant \mathrm{norm}(\Xi) \rightsquigarrow f}{\Gamma \vdash \mathbf{pack}\ M{:}S : \mathrm{norm}(\Xi) \rightsquigarrow f\ (\mathsf{unpack}\ \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\alpha}, x\ \Gamma^{\varphi} \rangle)}\ \textsc{E-pack}$$

Fig. 30. (Colour online) New rules for applicative paths and packages.

Pure modules and bindings. The real "heavy lifting" (so to speak) happens in M-SEAL. It abstracts the witness types $\overline{\tau}$ over all type variables from Γ, thereby lifting their kinds in a manner similar to what happens in the elaboration of applicative functor signatures (except that Γ generally contains more than just the functor's parameters). Similarly, the rule abstracts the term component over all of Γ, thereby constructing the desired functor representation *inside* the package. Both these abstractions together cause the rule to yield a lifted existential type, as desired for an applicative functor.

But using a different elaboration invariant for pure modules has implications on the translation of other module constructs as well. In all places where the original, impure rules had to unpack and re-pack existential packages in the translated term, the pure ones also have to apply and re-abstract Γ (rules M-DOT, B-MOD, and B-SEQ). To avoid the need for a separate set of rules for pure and impure elaboration, we use the Γ^{φ} notation defined in Figure 28 to make these steps conditional on the effect φ. Rules that return concrete signatures do not need to shuffle around Γ, but simply insert the expected abstraction (rules M-VAR, M-FUNCT-I, M-APP, B-VAL, B-TYP, B-SIG, B-EMT). Rule B-SEQ on the other hand is somewhat trickier, because it has to handle all possible combinations of effects φ_1 and φ_2. (The let-expression around e_2 in this rule is actually redundant when $\varphi_2 = \mathrm{P}$ — because e_2 is a closed expression in that case — but we leave it alone for the sake of simplicity of the rule.)

Interestingly, sealing is always pure according to the rules. That is because the syntax of our module language only permits sealing of module *variables*, which are values. When expanding the derived syntax for $M :> S$ (Figure 2), however, for an M that is impure, the overall expression will be regarded impure as advertised, thanks to the rules M-DOT and B-SEQ that are needed to type the expansion.

Rule M-UNPACK is the only source of unconditional impurity. First of all, an unpacked expression *must* be considered impure if the expression being unpacked might compute to package values with different type components (as in the body of Flip). But second, even if the expression being unpacked is already a value, it is not possible to treat its unpacking as a pure module expression because doing so would require us to be able to somehow project out its type components as type-level expressions. (This is necessary if we want to be able to lift the type components of the

```
Set ⇝
  pack ⟨λα.list α,
        λα.λP Elem : {t : [= α : Ω],
                      eq : [α × α → bool],
                      less : [α × α → bool]}.
          f ((let y1 = λα.λP Elem : {...}.{elem = [α : Ω]} in
              let y2 =
                let y21 = (let elem = ... in λα.λP Elem : {...}.{set = [list α : Ω]}) in
                let y22 =
                  ...
              in λα.λP Elem : {...}.
                let elem = (y1 α Elem)P.elem in
                let set = ((y2 α Elem)P elem)P.set in
                let empty = ((y2 α Elem)P elem)P.empty in
                let add = ((y2 α Elem)P elem)P.add in
                let mem = ((y2 α Elem)P elem)P.mem in
                {elem = elem, set = set, empty = empty, add = add, mem = mem}
              ) α Elem)P
        ⟩∃β:(Ω→Ω).∀α.{t:[=α:Ω],... }→P{set:[=β α:Ω], elem:[=α:Ω], empty:[β α], add:[...], mem:[...]}
```

Fig. 31. Example: applicative functor elaboration.

unpack over the context Γ.) If we were interpreting ML modules into a dependent type theory, this might be possible; however, as discussed in Section 7.1, given that we are interpreting into F_ω, with packaged modules represented as existentials, there is no way to project out their abstract type components as type-level expressions, so we treat all unpacked expressions as impure.

Figure 31 shows the translation of the Set functor as an applicative functor according to our rules. Compared to the elaboration previously given in Figure 15, the main difference is that packing and λ-abstractions have switched order, and that the existential witness type has been abstracted over α accordingly. Moreover, the nested local let-bindings in the sequence rule have been replaced by applications of the functor parameters inside the abstraction. As before, the translation produces many administrative redexes that can be optimized via some fairly obvious partial evaluation scheme. Figure 32 shows the translated Set functor after eliminating all intermediate structures and functors this way, for easier comparison with the analogous generative implementation in Figure 16.

Obviously, always abstracting over Γ in its entirety, as our rules do for pure modules, also leads to over-abstraction (although that is not visible in the example, where we assume the initial Γ to be empty). In particular, it would be sufficient to abstract only over the part of Γ that is bound by, or local to, the outermost applicative functor surrounding a pure module, if any. However, semantically the difference does not matter much. It is not difficult to refine the translation so that it avoids redundant abstractions, but the bureaucracy for tracking the necessary extra information would unnecessarily clutter the rules, so for presentational purposes we chose the simpler path. A real-world implementation can easily optimize the redundant abstractions by what amounts to (fairly straightforward) local partial reductions. We would also expect an implementation to present types in a more

$$
\begin{array}{l}
\mathsf{Set} \rightsquigarrow \\
\quad \mathsf{pack}\ \langle \lambda\alpha.\mathsf{list}\,\alpha, \\
\qquad \lambda\alpha.\lambda_{\mathrm{P}} \mathit{Elem} : \{\mathsf{t} : [= \alpha : \Omega], \\
\qquad\qquad\qquad\quad \mathsf{eq} : [\alpha \times \alpha \to \mathsf{bool}], \\
\qquad\qquad\qquad\quad \mathsf{less} : [\alpha \times \alpha \to \mathsf{bool}]\}. \\
\qquad\quad f\,(\mathsf{let}\ \mathit{elem} = [\alpha : \Omega]\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathit{set} = [\mathsf{list}\,\alpha : \Omega]\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathit{empty} = [\mathsf{nil}]\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathit{add} = [\ldots \mathit{Elem}.\mathsf{eq} \ldots \mathit{Elem}.\mathsf{less} \ldots]\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathit{mem} = [\ldots \mathit{Elem}.\mathsf{eq} \ldots \mathit{Elem}.\mathsf{less} \ldots]\ \mathsf{in} \\
\qquad\qquad \{\mathsf{elem} = \mathit{elem},\ \mathsf{set} = \mathit{set},\ \mathsf{empty} = \mathit{empty},\ \mathsf{add} = \mathit{add},\ \mathsf{mem} = \mathit{mem}\}) \\
\quad \rangle_{\exists\beta:(\Omega\to\Omega).\forall\alpha.\{\mathsf{t}:[=\alpha:\Omega],\ldots\}\to_{\mathrm{P}}\{\mathsf{set}:[=\beta\,\alpha:\Omega],\ \mathsf{elem}:[=\alpha:\Omega],\ \mathsf{empty}:[\beta\,\alpha],\ \mathsf{add}:[\ldots],\ \mathsf{mem}:[\ldots]\}}
\end{array}
$$

Fig. 32. Example: applicative functor elaboration, simplified.

readable way to the user (*e.g.*, as module paths), but such concerns are outside the scope of this article.

Paths and packages. Finally, Figure 30 shows the modified rules for paths and packages. They should not reveal any surprises at this point, because all that changes is the insertion of the right Γ-abstraction/application necessary to match the module rules.

Importantly, the path rule now fully supports functor applications in type paths. For example, the type expression (Set IntOrd).set is well-formed when Set is an appropriate applicative functor. This is simply a consequence of our semantic treatment of paths: when Set is bound to a functor with the signature given in Figure 27, its outer $\exists\beta$ is separated in the environment (according to rule B-SEQ) and the module (Set IntOrd).set simply has the atomic signature $[= \beta\ \mathsf{int} : \Omega]$. Since this signature contains no existentials, it is trivially a legal path.

Contrast that to the behavior under a generative signature for Set, like the one originally given in Figure 12. Under that typing, (Set IntOrd).set has the type $\exists\beta.[= \beta : \Omega]$, with a fresh local β that prevents it from typechecking as a path in rule P-MOD. The same applies to any other path to an abstract type defined inside a generative functor.

Our semantics does, however, allow functor paths with applications of generative functors if they do *not* refer to such abstract types. For example, (Set IntOrd).elem yields signature $\exists\beta.[= \mathsf{int} : \Omega]$, which *can* be used as a path — even in the basic system of Section 4! In the extended system presented in this section, we could easily rule out such corner cases by requiring P to be a *pure* module in rule P-MOD, but there is no real reason to do so.

8 Abstraction safety, dynamic purity, and sharing

The elaboration rules for applicative functors that we presented in the previous section are type-safe in the basic syntactic sense that they produce well-typed F_ω terms and types, but they are not *abstraction-safe*. By "abstraction safety", we are referring to the ability to impose local representation invariants on the

```
signature NAME = {
  type name
  val new : unit → name
  val equal : name × name → bool
}
module Name = fun X:{} ⇒ {
  type name = int
  val counter = ref 0
  val new () = (counter := !counter + 1; !counter)
  val equal (x, y) = (x = y)
} :> NAME

module Empty = {}
module Name1 = Name Empty
module Name2 = Name Empty
```

Fig. 33. Problems with abstraction safety in applicative functors: dynamic impurity.

abstract types defined by a sealed module expression, and to reason locally about the implementation of the sealed module under the assumption that all enclosing program contexts will preserve the imposed invariants.[9]

The failure to provide abstraction safety is not a peculiar fault of our semantics: contrary to popular belief, *none* of the existing accounts of applicative functors in the literature (or in ML compilers) provide abstraction safety either (Harper *et al.*, 1990; Leroy, 1995; Russo, 1998; Shao, 1999; Dreyer *et al.*, 2003). The reason, in short, is that tracking only *static* purity of module expressions — as we have done in the previous section, and as other approaches have done before us — is not sufficient: it is important for the purpose of abstraction safety to track *dynamic* purity as well. In a similar vein, it is not sufficient to consider only *static* module equivalence — *i.e.,* the equivalence of type components — to decide the equivalence of types resulting from pure functor applications: we also need to consider *dynamic* module equivalence, *i.e.,* the equivalence of value components, as well.

To see what the issue with abstraction safety is, let us turn to the illustrative set of examples in Figures 33 and 34. The first example, concerning the functor Name and its instantiations Name_1 and Name_2, demonstrates why we may want to require a functor that is statically pure, but not dynamically pure, to be treated as generative. The remaining examples, concerning various applications of the Set functor, show how ensuring abstraction safety can even be quite tricky when working

[9] The term "abstraction-safe" (or "abstraction-secure") has appeared in the literature a number of times, but as far as we know without a clear formal definition. The informal description we have given here matches the use of the term in various papers by Sewell et al. (Leifer *et al.*, 2003; Sewell *et al.*, 2007). To make this precise, we would need to build a parametric model of the language and use it to establish interesting invariants for abstract data types. This is clearly beyond the scope of the present article and would in fact constitute new research, since as far as we know no one has yet attempted to build parametric models for full-fledged ML-style modules. If anything, though, our F-ing semantics may help point the way forward in this regard, since we show how to understand modules in terms of System F_ω, for which parametric models do exist (*e.g.,* Atkey (2012)).

```
module IntOrd = {type t = int; val eq = Int.eq; val less = Int.less}
module IntOrd' = IntOrd
module Set0 = Set IntOrd
module Set1 = Set IntOrd'
module Set2 = Set {type t = int; val eq = Int.eq; val less = Int.less}
module Set3 = Set {type t = int; val eq = Int.eq; val less = Int.greater}

module F = fun X : {} ⇒
             {type t = int; val eq = Int.eq;
              val less = if random() then Int.less else Int.greater}
module Set4 = Set (F Empty)
module Set5 = Set (F Empty)
```

Fig. 34. Problems with abstraction safety in applicative functors: dynamic module inequivalence.

with a functor that *is* dynamically pure, as long as we do not track dynamic module equivalence.

First, consider the functor Name, which implements an ADT of fresh names. Every time Name is instantiated, it will return a module with its own abstract type name, along with its own private integer counter (of type ref int) — initially set to 0 — which can be incremented to generate a fresh value of type name every time its new operation is invoked. In order to ensure that new produces a fresh name every time it is applied, it is crucial that each instantiation of Name have a distinct name type — *i.e.,* that we treat Name as a generative functor. Otherwise, calling Name_1.new might produce a name that Name_2.new had already produced.[10] However, since Name does not involve any uses of unpacking — *i.e.,* it is statically pure — our semantics from Section 7 would consider it to be applicative, as would OCaml (since in OCaml all functors are applicative) and Moscow ML (in which, even if Name were declared as generative, it could be subsequently coerced to an applicative signature by eta-expansion, thus violating abstraction safety). In the case of our semantics from Section 7, one could induce Name to be considered generative by replacing the sealing in its body with a pack at NAME followed by an unpack, but this is a rather indirect approach, and it does not work in OCaml or Moscow ML due to their restrictions on the use of the unpack construct.

Second, consider the set types defined by modules Set_0 through Set_5 in Figure 34. The set implementation is purely functional, so it may be more surprising to some readers that abstraction safety can still be a problem with this functor! The types Set_0.set, Set_1.set, and Set_2.set should clearly be equivalent, since they are constructed by passing Set the exact same argument IntOrd, just written three different ways. To ensure abstraction safety, however, Set_3.set should be

[10] One can, of course, engender *use-site* generativity by explicitly sealing each application of Name with the signature NAME. However, this is no substitute for true abstraction safety, since it demands disciplined use of sealing on the part of clients of the Name functor — it does not ensure that any local invariants on the abstract name type will be preserved under linking with arbitrary clients. For a more detailed semantic explanation of the importance of generativity in this example, see Ahmed, Dreyer & Rossberg (2009).

considered distinct from the others: the argument passed to Set in the definition of Set_3 provides a different ordering on integers (Int.greater), thus rendering the representation of Set_3.set incompatible with the representation of sets ordered by Int.less. If we were to treat Set_2.set and Set_3.set as equivalent, the definition **val** s = Set_2.add(1, Set_3.add(2, Set_2.add(3, Set_2.empty))) would become well-typed. That would be disastrous, because it would yield a set value represented internally by the list [1,3,2], which violates the internal ordering invariants of *both* Set_2 and Set_3's list-based set representations. This would result in unpredictable behavior from any further interactions with Set_2 and Set_3's operations; for instance, Set_2.mem(2, s) and Set_3.mem(2, s) would both return false!

As for Set_4.set and Set_5.set, it is important to distinguish them from each other (and from all the other set types), for the following reason. Depending on the result of a random coin flip, the expression F Empty used in the definition of Set_4 and Set_5 will evaluate to a module that is dynamically equivalent to *one* of the argument modules used in the definitions of Set_2 and Set_3. Consequently, each of the types Set_4.set and Set_5.set will end up dynamically being compatible with either Set_2.set or Set_3.set, but statically we have no way of knowing which will be equivalent to which! We must therefore conservatively insist that they are both fresh types, even though they are defined using the exact same module expression Set (F Empty).[11]

Getting abstraction-safe applicative behavior on these Set examples seems to be hard, as indeed all previous accounts of applicative functors are unsafe and/or overly conservative in one way or another. Assuming that the Set functor has been assigned an applicative signature, the type system of Section 7, as well as those of Moscow ML, Shao (1999), and Dreyer *et al.* (2003), all consider Set_0 through Set_5 to have equivalent set components. The reason is that they employ a "static" notion of module equivalence: they pretend that the meaning of abstract types created by a functor only depends on the *types* from the functor's parameters, while ignoring any dependency on parameter *values*. Consequently, they consider the type components of Set(M_1) and Set(M_2) to be equivalent so long as M_1 and M_2 have equivalent type components. As one can plainly see, though, this approach is demonstrably unsafe: since sets ordered one way are not compatible with sets ordered a different way, the semantics of the type component set in the body of the Set functor clearly depends on the value component less of the functor argument. A correct treatment of abstraction safety thus demands capturing the dependency of abstract types on *entire* modules, *i.e.,* both type and value components — which is completely natural from the point of view of dependent type systems.

OCaml is closest to this ideal: it only considers Set(M_1) and Set(M_2) to be equivalent if $M_1 = M_2$ syntactically. However, this is quite restrictive, with the consequence that Set_0.set, Set_1.set, and Set_2.set are all considered distinct for no

[11] As in the case of the Name functor, one could try to rely on disciplined *use-site* sealing to work around this problem — *e.g.,* by sealing the results of all applications of the Set functor appropriately, or by introducing phantom types into the functor parameter, instantiated to fresh abstract types associated with an ordering as necessary. But once more, this would wrongly place the burden of protecting the abstraction on (all) *clients* of the functor, while depriving its *implementer* of the ability to perform local reasoning about the correctness of the abstraction.

good reason. Moreover, OCaml deems $\mathsf{Set}_4.\mathsf{set}$ and $\mathsf{Set}_5.\mathsf{set}$ equivalent just because they are constructed from syntactically identical module expressions, even though doing so constitutes a clear violation of abstraction safety.

8.1 Elaboration

In this section, we refine our elaboration from Section 7 in order to arrive at a semantics that achieves abstraction safety in a satisfactory manner.[12] Our approach is as follows.

First, in order to deal with examples like the Name functor, which ought not to be applicative, we now take into account not only static purity, but also *dynamic purity*. That is, in the elaboration of pure modules, we only permit value bindings that we can prove to have no side effects. The intuition behind this restriction is simple: if a module defines abstract types and also has computational effects, then it is only safe to assume that the semantic meanings of the abstract types are tied up with the effects. For example, the meaning of the name type in the Name functor is semantically tied to the stateful counter — in particular, it represents the set of natural numbers less than the current value of counter (which may only grow over time).

Second, we observe that it is only abstraction-safe to equate the types returned by applicative functors if the arguments passed to them are dynamically (as well as statically) equivalent. This explains why Set_0, Set_1, and Set_2 produce equivalent set types, but they are distinct from $\mathsf{Set}_3.\mathsf{set}$. In order to check for dynamic equivalence of functor arguments, we thus refine our semantics to (conservatively) track the "identity" of values. In essence, we emulate a simple form of dependent typing without actually requiring dependent types.

Dynamic purity. Determining whether an expression is dynamically pure is undecidable. As a conservative approximation, we piggyback on a notion that already exists in ML: the syntactic classification of *non-expansive* expressions — essentially, syntactic values. In ML, this notion is used in the core language to prevent unsound implicit polymorphism, the so-called *value restriction* (Wright, 1995). It makes perfect sense to reuse it here, because an applicative functor can be thought of as a polymorphic function on steroids.

Figure 35 gives a suitable grammar for non-expansive expressions $\mathbb{E}$ that accounts for paths and packages. The "..." in the grammar for $\mathbb{E}$ will typically define a sublanguage of what is templated as "..." in the grammar for E (cf. Figure 1), but the specifics obviously depend on the concrete core language. For module expressions $\mathbb{M}$ contained in $\mathbb{E}$, the only constructs disallowed are functor application and unpacking.

[12] As explained in footnote 9, the notion of abstraction safety is somewhat informal. The claim that the semantics described in this section regains abstraction safety is likewise informal, and to justify it formally would take us beyond the scope of this article. At the very least, we believe it is clear that our semantics does not suffer from the same problems with abstraction safety that afflict previous approaches.

$$
\begin{array}{lcl}
\mathbb{E} & ::= & \ldots \mid \mathbb{P} \mid \textbf{pack}\ \mathbb{M}{:}S \\
\mathbb{P} & ::= & \mathbb{M} \\
\mathbb{M} & ::= & X \mid \{\mathbb{B}\} \mid \mathbb{M}.X \mid \textbf{fun}\ X{:}S \Rightarrow M \mid X{:>}S \\
\mathbb{B} & ::= & \textbf{val}\ X{=}\mathbb{E} \mid \textbf{type}\ X{=}T \mid \textbf{module}\ X{=}\mathbb{M} \mid \textbf{signature}\ X{=}S \mid \textbf{include}\ \mathbb{M} \mid \epsilon \mid \mathbb{B};\mathbb{B}
\end{array}
$$

Fig. 35. Non-expansive expressions.

$$
\begin{array}{lcl}
\text{(paths)} & \pi & ::= \quad \alpha \mid \pi\,\overline{\tau} \\
\text{(concrete signatures)} & \Sigma & ::= \quad [= \pi : \tau] \mid \ldots
\end{array}
$$

Abbreviations:

$$
\begin{array}{lcl}
\text{(types)} \quad [= \pi : \tau] & := & \{\mathsf{val} : \tau, \mathsf{nam} : \pi\} \\
\text{(expressions)} \quad [e\ \mathsf{as}\ e'] & := & \{\mathsf{val} = e, \mathsf{nam} = e'\}
\end{array}
$$

Fig. 36. (Colour online) Semantic signatures for tracking sharing.

Depending on the details of the core language and its type system, more refined strategies are possible for classifying pure value bindings. Fortunately, this does not affect anything else in our development, so we stick with the simple notion of non-expansiveness for simplicity; adopting something more sophisticated should be straightforward.

Dynamic module equivalence and semantic paths. We have demonstrated above that abstraction safety requires type equivalence to take dynamic module equivalence into account. As we have mentioned already, our approach relies on the tracking of "identities" for value components of modules. Since equivalence of values is obviously undecidable in general, and because we also want to avoid the need for true dependent types, we again use a conservative approximation: our new typing rules employ "phantom types" to identify values, *i.e.,* abstract type expressions that we call *semantic paths* π. Usually, such a path is just a type variable, but due to the lifting that happens with applicative functors, it can actually take the more general form defined in Figure 36.

Paths are recorded in an extended definition of atomic value signature, also given in Figure 36. Consequently, every value binding or declaration will be associated with a semantic path. As with abstract types, we can quantify over path variables (existentially and universally), and thus abstract over value identities.

Semantic paths can be viewed as a refinement of the concept of *structure stamps*, which tracked structure identity in SML'90 (Milner *et al.*, 1990). Here, we reinterpret the *ad hoc* operational notion of "stamp" as a phantom type introduced via System F quantification, and we use it to stamp individual values rather than whole structures, thus enabling the tracking of identities at a finer granularity. (We could reconstruct "real" structure stamps, essentially by tracking module identities in addition to value identities. But in the presence of fine-grained value paths we see no additional benefit in also having structure stamps.)

Obviously, our notion of semantic paths could be refined in various ways. For example, certain values, such as scalar constants, could be captured more precisely

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

$$\frac{\Gamma \vdash T : \Omega \rightsquigarrow \tau \qquad \kappa_\alpha = \Omega}{\Gamma \vdash \mathbf{val}\ X{:}T \rightsquigarrow \exists \alpha.\{l_X : [= \alpha : \tau]\}}\ \text{D-VAL}$$

Subtyping $\boxed{\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f}$

$$\frac{\pi = \pi' \qquad \Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f}{\Gamma \vdash [= \pi : \tau] \leqslant [= \pi' : \tau'] \rightsquigarrow \lambda x{:}[= \pi : \tau].[f\,(x.\mathsf{val})\ \mathsf{as}\ x.\mathsf{nam}]}\ \text{U-VAL}$$

Bindings $\boxed{\Gamma \vdash B :_\varphi \Xi \rightsquigarrow e}$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e \qquad \kappa_\alpha = \Omega \qquad \forall \mathbb{E}.\ E \neq \mathbb{E} \qquad \forall P.\ E \neq P}{\Gamma \vdash \mathbf{val}\ X{=}E :_{\mathrm{I}} \exists \alpha.\{l_X : [= \alpha : \tau]\} \rightsquigarrow \mathsf{pack}\ \langle \{\}, \{l_X = [e\ \mathsf{as}\ \{\}]\}\rangle}\ \text{B-VAL-I}$$

$$\frac{\Gamma \vdash \mathbb{E} : \tau \rightsquigarrow e \qquad \kappa_\alpha = \Gamma \to \Omega \qquad \forall P.\ \mathbb{E} \neq P}{\Gamma \vdash \mathbf{val}\ X{=}\mathbb{E} :_{\mathrm{P}} \exists \alpha.\{l_X : [= \alpha\ \Gamma : \tau]\} \rightsquigarrow \mathsf{pack}\ \langle \lambda\Gamma.\{\}, \lambda\Gamma.\{l_X = [e\ \mathsf{as}\ \{\}]\}\rangle}\ \text{B-VAL-P}$$

$$\frac{\Gamma \vdash P :_\varphi \exists \overline{\alpha}.[= \pi : \tau] \rightsquigarrow e}{\Gamma \vdash \mathbf{val}\ X{=}P :_\varphi \exists \overline{\alpha}.\{l_X : [= \pi : \tau]\} \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\alpha}, \lambda\Gamma^\varphi.\{l_X = x\}\rangle}\ \text{B-VAL-ALIAS}$$

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

$$\frac{\Gamma \vdash P :_\varphi \exists \overline{\alpha}.[= \pi : \tau] \rightsquigarrow e \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash P : \tau \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x\rangle = e\ \mathsf{in}\ (x\ \Gamma^\varphi).\mathsf{val}}\ \text{E-PATH}$$

Fig. 37. (Colour online) Elaboration of value sharing.

by reflecting them on the type level (equating more values and hence allowing more programs to typecheck). However, such details are beyond the scope of this article.

Elaboration. The new and modified rules for value declarations and bindings are shown in Figure 37. We once more have highlighted the relevant changes.

For a value *declaration* (rule D-VAL), we always introduce a fresh path variable (of kind Ω) as a place-holder for the actual value's identity. For value *bindings*, there are now three rules. If the binding just rebinds a suitable path P, then we actually know the value's identity, and can retain it (rule B-VAL-ALIAS). Otherwise, we treat the value as "new" and introduce a fresh path variable representing it; the witness type for the variable does not matter, so we simply pick $\{\}$. The binding can be treated as pure if the expression is non-expansive (rule B-VAL-P), in which case we have to abstract over Γ inside the package, in the same way we did in the sealing rule M-SEAL (Figure 29).

Subtyping requires atomic value signatures to have matching paths (rule U-VAL). For now, this condition is trivial to meet, because a rule D-VAL always produces a

$$
\begin{array}{l}
(\mathsf{Elem} : \mathsf{ORD}) \Rightarrow (\mathsf{SET}\ \textbf{where type}\ \mathsf{elem} = \mathsf{Elem.t}) \\
\quad \rightsquigarrow\ \exists \beta\!:\!(\Omega^3 \to \Omega), \beta_1\!:\!(\Omega^3 \to \Omega), \beta_2\!:\!(\Omega^3 \to \Omega), \beta_3\!:\!(\Omega^3 \to \Omega). \\
\qquad \forall \alpha\!:\!\Omega, \alpha_1\!:\!\Omega, \alpha_2\!:\!\Omega.\{\mathsf{t} : [= \alpha : \Omega], \\
\qquad\qquad\qquad\qquad \mathsf{eq} : [= \alpha_1 : \alpha \times \alpha \to \mathsf{bool}], \\
\qquad\qquad\qquad\qquad \mathsf{less} : [= \alpha_2 : \alpha \times \alpha \to \mathsf{bool}]\} \\
\qquad\quad \to_{\mathrm{P}} \{\mathsf{set} : [= \beta\ \alpha\ \alpha_1\ \alpha_2 : \Omega], \\
\qquad\qquad \mathsf{elem} : [= \alpha : \Omega], \\
\qquad\qquad \mathsf{empty} : [= \beta_1\ \alpha\ \alpha_1\ \alpha_2 : \beta\ \alpha\ \alpha_1\ \alpha_2], \\
\qquad\qquad \mathsf{add} : [= \beta_2\ \alpha\ \alpha_1\ \alpha_2 : \alpha \times \beta\ \alpha\ \alpha_1\ \alpha_2 \to \beta\ \alpha\ \alpha_1\ \alpha_2], \\
\qquad\qquad \mathsf{mem} : [= \beta_3\ \alpha\ \alpha_1\ \alpha_2 : \alpha \times \beta\ \alpha\ \alpha_1\ \alpha_2 \to \mathsf{bool}]\} \\
\qquad (\text{where } \Omega^3 \to \Omega \ := \ \Omega \to \Omega \to \Omega \to \Omega)
\end{array}
$$

Fig. 38. (Colour online) Example: signature elaboration with value tracking.

separate, existentially quantified path for every single value declaration, so that the matching rule U-MATCH can pick them freely before descending into the subtyping check. In Section 8.2 below, we present another small language extension that makes the condition more interesting, though.

Finally, in the premise of the modified rule E-PATH, P is elaborated as a full module. This is more permissive than going through the generic path rule P-MOD as before (cf. Figure 30), because the new rule also allows dropping any quantified variable that only occurs in the path π. Without the modified rule, our encoding of **let**-expressions would no longer work, since every local value definition (that is not a mere alias) introduces an existential quantifier as its path. (Consider **let val** x = 1 **in** x+x, which desugars into {**val** x = 1; **val** it = x+x}.it — as a module, its type is $\exists \alpha_1 \alpha_2.[= \alpha_2 : \mathsf{int}]$, so that α_2 cannot be avoided by the path rule P-MOD. Rule E-PATH, on the other hand, can drop both variables.)

Example. Figure 38 shows the result of elaborating the (applicative) functor signature describing Set, previously shown in Figure 27, under the updated rules. Differences to the previous result are highlighted: atomic value signatures now carry path information, the signature abstracts the path variables α_1, α_2 and β_1 to β_3, and the export type β has to be applied not just to the argument type α but also to the argument paths α_1, α_2, accordingly.

Given a Set functor with the semantic signature from Figure 38, the types $\mathsf{Set}_0.\mathsf{set}$, $\mathsf{Set}_1.\mathsf{set}$, and $\mathsf{Set}_2.\mathsf{set}$ (from the beginning of the section) will be seen as equivalent: they all elaborate to the semantic type $\beta\ \mathsf{int}\ \pi_{\mathsf{eq}}\ \pi_{\mathsf{less}}$, with the two paths π_{eq} and π_{less} referring to the respective members of structure Int. They are distinguished from type $\mathsf{Set}_3.\mathsf{set}$, which elaborates to $\beta\ \mathsf{int}\ \pi_{\mathsf{eq}}\ \pi_{\mathsf{greater}}$.

Types $\mathsf{Set}_4.\mathsf{set}$ and $\mathsf{Set}_5.\mathsf{set}$ are also fresh, because the functor F will be deemed impure under the new rules, due to its binding for less, which features an expansive application (random()). Its semantic signature looks as follows (highlighting the pieces that have been added or changed with the refined rules):

$$
\begin{array}{l}
\mathsf{F} \ : \ \{\} \ \to_{\mathrm{I}} \ \exists \beta_1\!:\!\Omega.\{\mathsf{t} : [= \mathsf{int} : \Omega], \\
\qquad\qquad\qquad\qquad \mathsf{eq} : [= \pi_{\mathsf{eq}} : \mathsf{int} \times \mathsf{int} \to \mathsf{bool}], \\
\qquad\qquad\qquad\qquad \mathsf{less} : [= \beta_1 : \mathsf{int} \times \mathsf{int} \to \mathsf{bool}]\}.
\end{array}
$$

Hence, F delivers a fresh path for less with every application, and so each application of the Set functor to F Empty will produce different set types.

The Name functor will be considered impure under the new rules as well, because of the local effectful binding for counter. Here is its signature according to the refined rules:

$$\begin{aligned}\mathsf{Name} \;:\; \{\} \;\to_{\mathrm{I}}\; \exists \beta{:}\Omega, \beta_1{:}\Omega, \beta_2{:}\Omega.\{&\mathsf{name} : [= \beta : \Omega],\\ &\mathsf{new} : [= \beta_1 : \{\} \to \beta],\\ &\mathsf{equal} : [= \beta_2 : \beta \times \beta \to \mathsf{bool}]\}\end{aligned}$$

Consequently, the functor will behave generatively, with Name_1.name and Name_2.name elaborating to distinct fresh abstract types.

8.2 Sharing specifications

Once value identities matter for determining type equivalences, it can be useful to give the programmer the ability to explicitly specify sharing constraints between values. For example, consider a functor that takes two arguments, both with a sub-module Ord:

```
signature A = {module Ord : ORD; val v : Set(Ord).t; . . . }
signature B = {module Ord : ORD; val f : Set(Ord).t → int; . . . }
module F (X : A) (Y : B) = { . . . Y.f (X.v) . . . }
```

Clearly, the application in the functor's body cannot typecheck without knowing that X.Ord and Y.Ord are statically *and* dynamically equal. For that, we need to be able to impose sufficient constraints on the parameters.

Figure 39 presents syntax for manifest value specifications (using module paths P) and a related signature refinement using **where**. It also introduces similar forms to specify sharing between entire modules, which serves as an abbreviation for sharing all type and value components. Finally, we add a construct, "**like** P", which yields the signature of the module P, and thus can only be matched by modules that provide the same definitions as P. In essence, this describes a higher-order *singleton signature* in the manner introduced by Dreyer *et al.* (2003).[13] A manifest specification **module** $X{=}P$ is equivalent to the specification **module** X : **like** P. With these extensions, we can, for example, define the functor F properly as follows:

```
module F (X : A) (Y : B where module Ord = X.Ord) = { . . . Y.f (X.v) . . . }.
```

One subtlety to point out here is that the design of these constructs depends on the fact that our elaboration is deterministic, and so any path P trivially has a *unique type* in our system. If that were not the case — *e.g.,* if modules only had *principal* types — then the "**where module**" and the "**like**" construct would not yield a unique signature specification, *i.e.,* their meaning would be ambiguous. To compensate, it would be necessary to require the programmer to disambiguate those

[13] It is also very similar to the "**module type of**" operator that was introduced in recent versions of OCaml. The difference is that OCaml's operator does not propagate the identities of abstract types defined by the module, which we find rather surprising.

$$\begin{array}{llll} \text{(signatures)} & S & ::= & \ldots \mid S \textbf{ where val } \overline{X}{=}P \mid S \textbf{ where module } \overline{X}{=}P \mid \textbf{like } P \\ \text{(declarations)} & D & ::= & \ldots \mid \textbf{val } X{=}P \mid \textbf{module } X{=}P \end{array}$$

Fig. 39. Extension with value and module sharing specifications.

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

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}_1 \alpha \overline{\alpha}_2.\Sigma \qquad \Gamma \vdash P : [= \pi : \tau'] \rightsquigarrow e \qquad \Sigma.\overline{l_X} = [= \alpha : \tau] \qquad \Gamma \vdash \tau' \leqslant \tau \rightsquigarrow f}{\Gamma \vdash S \textbf{ where val } \overline{X}{=}P \rightsquigarrow \exists \overline{\alpha}_1 \overline{\alpha}_2.\Sigma[\pi/\alpha]} \text{ S-WHERE-VAL}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad \Gamma \vdash P : \Sigma' \rightsquigarrow e \qquad \Sigma.\overline{l_X} = \Sigma'' \qquad \overline{\alpha} = \overline{\alpha}_1 \uplus \overline{\alpha}_2 \qquad \exists \overline{\alpha}_2.\Sigma'' \text{ explicit} \qquad \Gamma, \overline{\alpha}_1 \vdash \Sigma' \leqslant \exists \overline{\alpha}_2.\Sigma'' \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash S \textbf{ where module } \overline{X}{=}P \rightsquigarrow \exists \overline{\alpha}_1.\Sigma[\overline{\tau}/\overline{\alpha}_2]} \text{ S-WHERE-MOD}$$

$$\frac{\Gamma \vdash P : \Sigma \rightsquigarrow e \qquad \Sigma \text{ explicit}}{\Gamma \vdash \textbf{like } P \rightsquigarrow \Sigma} \text{ S-LIKE}$$

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

$$\frac{\Gamma \vdash P : [= \pi : \tau] \rightsquigarrow e}{\Gamma \vdash \textbf{val } X{=}P \rightsquigarrow \{l_X : [= \pi : \tau]\}} \text{ D-VAL-EQ}$$

$$\frac{\Gamma \vdash P : \Sigma \rightsquigarrow e \qquad \Sigma \text{ explicit}}{\Gamma \vdash \textbf{module } X{=}P \rightsquigarrow \{l_X : \Sigma\}} \text{ D-MOD-EQ}$$

Fig. 40. Elaboration of value and module sharing specifications.

constructs with explicit signature annotations "$:S$" on the paths. A deterministic type system avoids any such nuisance.

Elaboration. The respective elaboration rules are shown in Figure 40. Rule S-WHERE-VAL is analogous to S-WHERE-TYP (cf. Figure 11).

Module refinement (rule S-WHERE-MOD) is slightly more involved. It is defined as refining every individual abstract value and type specification in submodule $\overline{X}$ of S. This module has the signature Σ'', and the type variables $\overline{\alpha}_2$ identify its abstract entities; the remaining $\overline{\alpha}_1$ are used elsewhere in Σ and remain untouched. The concrete signature Σ' of the refining path P has to match $\exists \overline{\alpha}_2.\Sigma''$. (Typically, $\overline{\alpha}_2$ will coincide with the subset of $\overline{\alpha}$ that are free in Σ'', because only in rare circumstances can matching succeed with an unquantified $\alpha \in \overline{\alpha}_1$ left over in Σ''.[14])

The rules for manifest value and module declarations are straightforward, as is the rule for singletons.

[14] With ML as a core language, one such example would be if Σ'' contained a value component of type t int → t int. This type could be matched by a Σ' in which the corresponding component had type $\forall \alpha.\alpha \to \alpha$, which does not mention t but can nonetheless be *instantiated* to t int → t int.

In all the module forms, a side condition about explicitness is necessary to maintain the elaboration invariant that is required for decidability (cf. Section 5.2). Inductively, we only know that the respective signatures are *valid*, but because they can occur on the right-hand side of a match, we would lose decidability (which we will prove in Section 9.2) if we did not require them to also be *explicit*. In practice, the signature of a path (or any module, for that matter) can always be enforced to be explicit by imposing a signature annotation. Alternatively, any "classic" syntactic path consisting only of variables, projection, and pure functor application will satisfy the explicitness criterion, as long as those variables in turn are bound to definitions with explicit signature annotations.

In the case of rule S-WHERE-MOD, however, $\exists\overline{\alpha}_2.\Sigma''$ can only be made explicit (and the refinement made well-formed) by ensuring that the signature of the specialized submodule is sufficiently self-contained, *i.e.,* none of its type components refers to any of the $\overline{\alpha}_1$ from the surrounding signature. It is not merely decidability concerns that demand this. For example, the refinement in

```
signature S = {type t : ⋆ → ⋆; module A : {type u = t int; ...}}
module B = {type u = int; ...}
signature T = S where module A = B
```

would require higher-order unification to find a t such that t int = int. Not only is that an undecidable problem in the general case, it also has more than one "solution" for this example, and the signature T would therefore have an ambiguous meaning. Consequently, the above example is disallowed by the rule — t is not rooted in the inner signature of A, although it mentions it. But the example can be disambiguated by splitting the refinement into stages:

```
signature T = (S where type t = fun a ⇒ a) where module A = B.
```

If all types from the surrounding signature have an alias in the submodule, however, then our system accepts the direct refinement:

```
signature S = {type t : ⋆ → ⋆; module A : {type u = t; ...}}
module B = {type u = fun a ⇒ list a; ...}
signature T = S where module A = B.
```

(And because we always $\beta\eta$-normalize all types, this even works when u is specified as **fun** a ⇒ t a in signature S.)

The "**where module**" construct has been a rather dark corner of ML-style modules. While it is often available in one form or another, its semantics tends to be either vague or over-restrictive (or both), and rarely is it properly specified. The structure sharing specifications of SML'90 (Milner *et al.*, 1990) were the earliest form of a comparable construct, but they were both relatively restricted and semantically complicated, resorting to global "admissibility" conditions. In SML'97 (Milner *et al.*, 1997), they were hence degraded to a form of syntactic sugar, but this is arguably not quite the right thing either, since their desugaring in fact relies on type information. As has been observed repeatedly by SML implementers, the SML'97 semantics has a severe limitation: it prevents the placement of structure sharing constraints on any signatures that export a single transparent type specification! Generalizations and improvements, including the complementary "**where module**"

(or "**where structure**") mechanism, have been discussed in online forums and implemented in some compilers (*e.g.*, SML/NJ (SML/NJ Development Team, 1993) and Alice ML (Rossberg *et al.*, 2004)), but have never been formalized as far as we are aware. In OCaml, "**with module**" is superficially similar, but actually *extends* a signature instead of just refining types, which apparently is considered a bug.[15] Our elaboration rule S-WHERE-MOD may thus be viewed as a novel step in the right direction.

9 Meta-theory revisited

Having made non-trivial extensions to our system in the last two sections, we need to revisit the meta-theoretical properties that we proved about the initial system in Section 5.

9.1 Soundness

The soundness statement for the new elaboration rules has to cover the elaboration of pure modules now. But first a helpful lemma about typing environment abstractions:

Lemma 9.1 (*Typing of environment abstraction*)
Let $\Gamma \vdash \square$ and $\Gamma_1, \Gamma, \Gamma_2 \vdash \square$.

1. If and only if $\Gamma \vdash \tau : \kappa$, then $\cdot \vdash \lambda\Gamma.\tau : \Gamma \to \kappa$.
2. If and only if $\Gamma_1, \Gamma, \Gamma_2 \vdash \tau : \Gamma \to \kappa$, then $\Gamma_1, \Gamma, \Gamma_2 \vdash \tau\ \Gamma : \kappa$.
3. If and only if $\Gamma \vdash \tau : \Omega$, then $\cdot \vdash \forall\Gamma.\tau : \Omega$.
4. If and only if $\Gamma \vdash e : \tau$, then $\cdot \vdash \lambda\Gamma.e : \forall\Gamma.\tau$.
5. If and only if $\Gamma_1, \Gamma, \Gamma_2 \vdash e : \forall\Gamma.\tau$, then $\Gamma_1, \Gamma, \Gamma_2 \vdash e\ \Gamma : \tau$.
6. $(\lambda\Gamma.\tau)\ \Gamma \equiv \tau$.

In the actual soundness statement, pure module elaboration has a somewhat more intricate invariant than its impure version, as given by part 7 of the following theorem (all other parts read as before):

Theorem 9.2 (*Soundness of elaboration with applicative functors*)
Let $\Gamma \vdash \square$.

1. If $\Gamma \vdash T : \kappa \rightsquigarrow \tau$, then $\Gamma \vdash \tau : \kappa$.
2. If $\Gamma \vdash E : \tau \rightsquigarrow e$, then $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash e : \tau$.
3. If $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f$ and $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$, then $\Gamma \vdash f : \tau \to \tau'$.
4. If $\Gamma \vdash P : \Sigma \rightsquigarrow e$, then $\Gamma \vdash \Sigma : \Omega$ and $\Gamma \vdash e : \Sigma$.
5. If $\Gamma \vdash S/D \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
6. If $\Gamma \vdash M/B :_{\mathrm{I}} \Xi \rightsquigarrow e$, then $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash e : \Xi$.
7. If $\Gamma \vdash M/B :_{\mathrm{P}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$, then $\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$ and $\cdot \vdash e : \exists\overline{\alpha}.\forall\Gamma.\Sigma$.
8. If $\Gamma \vdash \Xi \leqslant \Xi' \rightsquigarrow f$ and $\Gamma \vdash \Xi : \Omega$ and $\Gamma \vdash \Xi' : \Omega$, then $\Gamma \vdash f : \Xi \to \Xi'$.

[15] See the bug report at `http://caml.inria.fr/mantis/view.php?id=5514`.

9. If $\Gamma \vdash \Sigma \leqslant \exists\overline{\alpha}.\Sigma' \uparrow \overline{\tau} \rightsquigarrow f$ 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
By simultaneous induction on the derivations. Most cases are proved as before (Theorem 5.1), except that some use additional abstraction over Γ, and we have added a number of new rules, most of which are fairly straightforward. We give the two most relevant cases for elaborating applicative functors and pure modules:

- Case M-FUNCT-P: By induction on the first premise we know that $\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$, and by iterated inversion this implies (1) $\Gamma, \overline{\alpha} \vdash \Sigma : \Omega$. Hence we can show that $\Gamma, \overline{\alpha}, X{:}\Sigma \vdash \square$. By induction on the second premise it follows that (2) $\Gamma, \overline{\alpha}, X{:}\Sigma \vdash \exists\overline{\alpha}_2.\Sigma_2 : \Omega$ and (3) $\Gamma \vdash e : \exists\overline{\alpha}_2.\forall(\Gamma, \overline{\alpha}, X{:}\Sigma).\Sigma_2$. Statement (3) already proves the second goal, because $\exists\overline{\alpha}_2.\forall(\Gamma, \overline{\alpha}, X{:}\Sigma).\Sigma_2 = \exists\overline{\alpha}_2.\forall\Gamma.\forall\overline{\alpha}.\Sigma \to_{\mathrm{P}} \Sigma_2$ by the definition of environment abstraction.
  To prove the first goal, inverting (2) gives $\Gamma, \overline{\alpha}, X{:}\Sigma, \overline{\alpha}_2 \vdash \Sigma_2 : \Omega$, which can be trivially strengthened and reordered to $\Gamma, \overline{\alpha}_2, \overline{\alpha} \vdash \Sigma_2 : \Omega$. By weakening (1) to $\Gamma, \overline{\alpha}, \overline{\alpha}_2 \vdash \Sigma : \Omega$, applying F_ω typing rules, and induction over the length of $\overline{\alpha}_1$ and then $\overline{\alpha}_2$, we arrive at $\Gamma \vdash \exists\overline{\alpha}_2.\forall\overline{\alpha}.\Sigma \to_{\mathrm{P}} \Sigma_2 : \Omega$.
- Case M-SEAL: Since we assume that Γ is well-formed, the first premise implies (1) $\Gamma \vdash \Sigma' : \Omega$. By induction on the second premise we get $\Gamma \vdash \exists\overline{\alpha}.\Sigma$, which can be inverted to (2) $\Gamma, \overline{\alpha} \vdash \Sigma : \Omega$. By induction (part 9) we can conclude (3) $\overline{\Gamma \vdash \tau : \kappa_\alpha}$ and (4) $\Gamma \vdash f : \Sigma' \to \Sigma[\overline{\tau}/\overline{\alpha}]$.
  Consider the first goal first. By Lemma 9.1 and F_ω kinding, we get $\overline{\Gamma, \overline{\alpha'} \vdash \alpha'\ \Gamma' : \kappa_\alpha}$, and accordingly, $\Gamma, \overline{\alpha'} \vdash \overline{[\alpha'\ \Gamma/\alpha]} : \Gamma, \overline{\alpha}$, so that the substitution lemma applied to (2) yields $\Gamma, \overline{\alpha'} \vdash \Sigma\overline{[\alpha'\ \Gamma/\alpha]} : \Omega$. By induction over the length of $\overline{\alpha'}$, F_ω typing rules then give $\Gamma \vdash \exists\overline{\alpha'}.\Sigma\overline{[\alpha'\ \Gamma/\alpha]} : \Omega$ as desired.
  For the second goal, first derive (5) $\Gamma \vdash f\ X : \Sigma\overline{[\tau/\alpha]}$ by simple application of F_ω typing rules to (1) and (4). Lemma 9.1 then gives $\cdot \vdash \lambda\Gamma.f\ X : \forall\Gamma.\Sigma\overline{[\tau/\alpha]}$. Likewise, $\overline{\cdot \vdash \lambda\Gamma.\tau : \Gamma \to \kappa_\alpha}$ follows from (3). The lemma also gives $\overline{(\lambda\Gamma.\tau)\ \Gamma = \tau}$, and hence it holds that $\Sigma\overline{[\tau/\alpha]} = \Sigma\overline{[(\lambda\Gamma.\tau)\ \Gamma/\alpha]}$ and we can apply the conversion rule and Lemma 9.1 to (5) to get $\cdot \vdash \lambda\Gamma.f\ X : \forall\Gamma.\Sigma\overline{[(\lambda\Gamma.\tau)\ \Gamma/\alpha]}$. Since we assume that $\overline{\alpha'}$ are fresh by convention, this is the same type as $\forall\Gamma.\Sigma[\alpha'\ \Gamma/\alpha][(\lambda\Gamma.\tau)/\alpha']$, and induction over $\overline{\alpha'}$ for application of the pack typing rule gives the wanted result. □

9.2 Decidability

Recall from Section 5.2 that the decidability of our type system solely hinged on the decidability of subtyping — more specifically, type lookup for the matching rule U-MATCH. This has not changed with any of the extensions we made. In fact, except for the trivial incorporation of effect subtyping, the addition of applicative functors did not change the declarative subtyping and matching rules at all!

However, the presence of applicative functors *does* necessitate fundamental changes to their algorithmic implementation. In particular, type lookup now has to look into pure functor signatures in order to find suitable types for matching,

and the contravariance of functor parameters results in a significantly more complex definition of the lookup function. That also makes the surrounding definitions and proofs more involved than what we have seen so far. (The end of this section has a few remarks concerning this complexity.)

Validity and rootedness. First, we observe that our previous definition of signature validity and, specifically, rootedness (cf. Figure 18) is no longer appropriate — it is violated by the new rules for pure functors (S-FUNCT-P and M-FUNCT-P), where we lift an existential quantifier over a universal one, and thus separate the existential quantifier from the structure that roots its variables. To deal with the additional extensions from Section 8, we must also account for abstract value paths— however, they are treated like any other abstract type variable, so do not affect the definitions and proofs much. (That is, the essential meta-theoretical complexity encountered in this section already comes up for the simpler system from Section 7 alone.)

Let us consider a couple of simple examples first. An abstract type $\beta_1 : \Omega$ is rooted in a structure signature $\{\mathsf{t}_1\colon [= \beta_1 : \Omega]\}$ (as before), so that $\exists\beta_1.\{\mathsf{t}_1\colon [= \beta_1 : \Omega]\}$ is a valid (and explicit) signature. Likewise, structures can be roots for higher-kinded types, if they specify them at their higher kind — for example, $\beta_2 : \Omega \to \Omega$ is rooted in $\{\mathsf{t}_2\colon [= \beta_2 : \Omega \to \Omega]\}$ (still as before). What's new now is that types may also be rooted in a pure functor signature. For example, a higher-kinded $\beta_3 : \Omega \to \Omega$ can now be rooted in

$$\forall\alpha_1,\alpha_2.\{\mathsf{u}\colon [= \alpha_1 : \Omega], \mathsf{v}\colon [= \alpha_2 : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}_3\colon [= \beta_3\,\alpha_1\,\alpha_2 : \Omega]\}$$

if the *path* $\beta\,\alpha_1\,\alpha_2$ — with α_1,α_2 being exactly the list of abstract types that the functor quantifiers over — is rooted in the functor's result signature. Consequently,

$$\exists\beta_3.\forall\alpha_1,\alpha_2.\{\mathsf{u}\colon [= \alpha_1 : \Omega], \mathsf{v}\colon [= \alpha_2 : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}_3\colon [= \beta_3\,\alpha_1\,\alpha_2 : \Omega]\}$$

is a valid (and explicit) signature. (As a degenerate case, the universal quantifier in a functor signature can actually be empty; such functors can be roots even for abstract types of ground kind Ω — *e.g.*, β_4 is rooted in $\{\} \to_{\mathrm{P}} \{\mathsf{t}_4\colon [= \beta_4 : \Omega]\}$.)

Figure 41 gives an extended definition of validity and related properties. Rootedness takes applicative functors into account: a variable may now be rooted in a pure functor's codomain. As a side effect, the definition no longer is concerned with plain type variables only, but generalizes to semantic paths π. In the functor case, we extend the current path by applying the functor's universal variables before descending into the codomain, mirroring the kind-raising substitution performed by rule S-FUNCT-P. The path π in the rootedness relation is always "abstract", in the sense that it is restricted to the form $\alpha\ \overline{\alpha}'$. We write head($\pi$) to denote the head variable α in such a path.

However, we have to be careful not to treat variable occurrences inside a functor as a root when that functor's *argument* already mentions that variable. For example, the (valid) signature

$$\forall\alpha.\{\mathsf{u}\colon [= \alpha : \Omega], \mathsf{v}\colon [= \beta\,\alpha : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}\colon [= \beta\,\alpha : \Omega]\}$$

$$
\begin{array}{lcl}
\epsilon \text{ rooted in } \Sigma & :\Leftrightarrow & \text{always} \\
\alpha, \overline{\alpha} \text{ rooted in } \Sigma & :\Leftrightarrow & \alpha \text{ rooted in } \Sigma \text{ avoiding } \alpha, \overline{\alpha} \ \wedge \ \overline{\alpha} \text{ rooted in } \Sigma \\
\pi \text{ rooted in } [= \pi' : \tau] \text{ avoiding } \overline{\beta} \text{ (at } \epsilon) & :\Leftrightarrow & \pi = \pi' \\
\pi \text{ rooted in } [= \tau : \kappa] \text{ avoiding } \overline{\beta} \text{ (at } \epsilon) & :\Leftrightarrow & \pi = \tau \\
\pi \text{ rooted in } \{\overline{l : \Sigma}\} \text{ avoiding } \overline{\beta} \text{ (at } l.\overline{l}) & :\Leftrightarrow & \pi \text{ rooted in } \{\overline{l : \Sigma}\}.l \text{ avoiding } \overline{\beta} \text{ (at } \overline{l}) \\
\pi \text{ rooted in } \forall \overline{\alpha}.\Sigma_1 \to_{\mathrm{P}} \Sigma_2 \text{ avoiding } \overline{\beta} \text{ (at } \overline{l}) & :\Leftrightarrow & \pi\, \overline{\alpha} \text{ rooted in } \Sigma_2 \text{ avoiding } \overline{\beta} \text{ (at } \overline{l}) \\
 & & \wedge\ \overline{\beta} \cap \mathrm{fv}(\Sigma_1) = \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
[= \pi : \tau] \text{ explicit} & & \text{(always)} \\
\forall \overline{\alpha}.\Sigma \to_{\varphi} \Xi \text{ explicit} & :\Leftrightarrow & \exists \overline{\alpha}.\Sigma \text{ explicit} \wedge \Xi \text{ explicit} \\
\cdots & & \\
[= \pi : \tau] \text{ valid} & & \text{(always)} \\
\forall \overline{\alpha}.\Sigma \to_{\varphi} \Xi \text{ valid} & :\Leftrightarrow & \exists \overline{\alpha}.\Sigma \text{ explicit} \wedge \Xi \text{ valid} \\
\cdots & &
\end{array}
$$

Fig. 41. (Colour online) Validity for applicative functors.

cannot possibly be a root for β, even though the path $\beta\,\alpha$ has the right form in its codomain. Intuitively, with β already occurring in its argument, this functor cannot be the origin of the abstract type β. Rather, it represents a functor signature like $(\mathsf{X} : \{\textbf{type}\ \mathsf{u};\ \textbf{type}\ \mathsf{v} = \mathsf{b}\ \mathsf{u}\}) \Rightarrow \{\textbf{type}\ \mathsf{t} = \mathsf{b}\ \mathsf{X.u}\}$, where the type b that β corresponds to is bound somewhere else. (Technically, the refined type lookup algorithm that we are going to define in a moment could produce cyclic results if we allowed examples like this as input.) The problem extends to multiple variables. Imagine:

$$
\begin{array}{l}
\exists \beta_1 \beta_2.\{\mathsf{F} : \forall \alpha.\{\mathsf{t} : [= \alpha : \Omega], \mathsf{u}\colon [= \beta_2 : \Omega \to \Omega]\} \to_{\mathrm{P}} \{\mathsf{v} : [= \beta_1\,\alpha : \Omega]\}, \\
\qquad \mathsf{G} : \forall \alpha.\{\mathsf{t} : [= \alpha : \Omega], \mathsf{v}\colon [= \beta_1 : \Omega \to \Omega]\} \to_{\mathrm{P}} \{\mathsf{u} : [= \beta_2\,\alpha : \Omega]\}\}.
\end{array}
$$

We cannot allow such a signature to be regarded explicit, because β_1 and β_2 would then have a cyclic dependency.

The new rootedness judgment excludes such cyclic examples, by (1) enforcing that each rooted variable is "avoided" by any functor parameter signature its root is under, and (2) inductively requiring that for multiple variables, each root not only avoids the variable itself, but also any of the following ones, thereby imposing sequential dependencies. Intuitively, then, the order of the quantified variables has to reflect the order of the respective declarations from which they originate. (This means that we are no longer as free to reorder quantified variables as we were before. We can only pick an order that represents a topological sorting with respect to the (non-cyclic) dependency graph of the declarations. Our definition of signature normalization (Section 6) hence is in need of refinement. However, the details are not very interesting, so we omit them here.)

With the new and improved definition of rootedness, the validity lemma is valid again, and we can extend it to the pure judgments:

Lemma 9.3 (*Simple properties of validity with applicative functors*)

1. If and only if π rooted in Σ avoiding $\overline{\beta}_1$ and π rooted in Σ avoiding $\overline{\beta}_2$, then π rooted in Σ avoiding $\overline{\beta}_1, \overline{\beta}_2$.

$$
\begin{array}{lcll}
\mathrm{lookup}_{\epsilon}(\Sigma, \Sigma') & \uparrow & \epsilon & \text{always} \\
\mathrm{lookup}_{\alpha.\overline{\alpha}}(\Sigma, \Sigma') & \uparrow & \tau, \overline{\tau} & \text{if } \mathrm{lookup}_{\alpha}(\Sigma, \Sigma') \uparrow \tau \wedge \ \mathrm{fv}(\tau) \cap \overline{\alpha} = \emptyset \\
 & & & \quad \wedge \mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma'[\tau/\alpha]) \uparrow \overline{\tau} \\
\mathrm{lookup}_{\pi}([= \pi'' : \tau], [= \pi' : \tau']) & \uparrow & \pi'' & \text{if } \pi' = \pi \\
\mathrm{lookup}_{\pi}([= \tau : \kappa], [= \tau' : \kappa]) & \uparrow & \tau & \text{if } \tau' = \pi \\
\mathrm{lookup}_{\pi}(\{\overline{l : \Sigma}\}, \{\overline{l' : \Sigma'}\}) & \uparrow & \tau & \text{if } \exists l \in \overline{l} \cap \overline{l'}.\ \mathrm{lookup}_{\pi}(\{\overline{l : \Sigma}\}.l, \{\overline{l' : \Sigma'}\}.l) \uparrow \tau \\
\mathrm{lookup}_{\pi}(\forall \overline{\alpha}.\Sigma_1 \to_{\mathrm{P}} \Sigma_2, \forall \overline{\alpha}'.\Sigma_1' \to_{\mathrm{P}} \Sigma_2') & \uparrow & \lambda \overline{\alpha}'.\tau & \text{if } \mathrm{lookup}_{\overline{\alpha}}(\Sigma_1', \Sigma_1) \uparrow \overline{\tau}' \wedge \ \mathrm{head}(\pi) \notin \mathrm{fv}(\overline{\tau}') \\
 & & & \quad \wedge \mathrm{lookup}_{\pi\,\overline{\alpha}'}(\Sigma_2[\overline{\tau}'/\overline{\alpha}], \Sigma_2') \uparrow \tau
\end{array}
$$

Fig. 42. (Colour online) Algorithmic type lookup with applicative functors.

2. If π rooted in Σ avoiding $\overline{\beta}_1$ and $\mathrm{fv}(\Sigma) \cap \overline{\beta}_2 = \emptyset$, then π rooted in Σ avoiding $\overline{\beta}_1, \overline{\beta}_2$.
3. If $\overline{\alpha}$ rooted in Σ, then $\overline{\alpha}$ rooted in $\Sigma[\overline{\tau}'/\overline{\alpha}']$, provided $\overline{\alpha} \cap (\mathrm{fv}(\overline{\tau}') \cup \overline{\alpha}') = \emptyset$.
4. If Ξ explicit, then Ξ valid.
5. If Ξ valid/explicit, then $\Xi[\overline{\tau}/\overline{\alpha}]$ valid/explicit.
6. If Ξ valid/explicit, then $\mathrm{norm}(\Xi)$ valid/explicit.

Lemma 9.4 (*Signature validity with applicative functors*)
Assume Γ valid.

1. If $\Gamma \vdash P : \Sigma \rightsquigarrow e$, then Σ valid.
2. If $\Gamma \vdash S/D \rightsquigarrow \Xi$, then Ξ explicit.
3. If $\Gamma \vdash M/B :_{\varphi} \Xi \rightsquigarrow e$, then Ξ valid.

Type Lookup. Of course, the more liberal definition of rootedness and signature validity now necessitates a more general type lookup algorithm. The upgrade is shown in Figure 42. Like rootedness, it now deals with semantic paths π instead of plain variables. That is, it no longer just looks for type variables but for paths. When lookup descends into the codomain of a functor type, it extends the current path with the functor's parameter variables. These parameters become parameters of the looked-up type, matching up with the raised kind that an abstract type from an applicative functor is given.

For example, consider

$$
\begin{array}{l}
\mathrm{lookup}_{\beta}(\forall \alpha.\{\mathsf{u}\colon [= \alpha : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}\colon [= \mathsf{int} : \Omega]\}, \\
\qquad \forall \alpha'.\{\mathsf{u}\colon [= \alpha' : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}\colon [= \beta\,\alpha' : \Omega]\})
\end{array}
$$

which looks for the type $\beta : \Omega \to \Omega$ (rooted in the second signature) in the first signature. It first takes the variables from the root's universal quantifier (in this case only a single α') to extend the path β to $\beta\,\alpha'$. It then performs lookup for this new path in the functors' codomains, yielding type int. Adding the parameters in the end, it returns $\lambda\alpha'.\mathsf{int}$ as the appropriate substitution for β itself.

But that is not enough. In general, a type looked up in the codomain may have occurrences of variables from the left hands universal quantifier, which would escape their scope if we left them alone. Consider:

$$
\begin{array}{l}
\mathrm{lookup}_{\beta}(\forall \alpha.\{\mathsf{u}\colon [= \alpha : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}\colon [= \mathsf{list}\ \alpha : \Omega]\}, \\
\qquad \forall \alpha'.\{\mathsf{u}\colon [= \alpha' : \Omega]\} \to_{\mathrm{P}} \{\mathsf{t}\colon [= \beta\,\alpha' : \Omega]\}).
\end{array}
$$

Here, just performing lookup in the codomain would give us $\mathsf{list}\ \alpha$ for $\beta\,\alpha'$, which is no good because the α it contains would be unbound. As with functor subtyping, we hence have to substitute α first, in a contravariant fashion. We do so with the corresponding types inversely looked up in the right-hand side's domain, *i.e.,* $\mathrm{lookup}_{\alpha}(\{\mathsf{u}\colon [= \alpha' : \Omega]\}, \{\mathsf{u}\colon [= \alpha : \Omega]\})$ for the example, and thereby mapping α to α'. As a result, the main lookup will return $\mathsf{list}\ \alpha'$ for $\beta\,\alpha'$ — but that is fine, because we have to lambda-abstract over α' anyway. We arrive at $\lambda\alpha'.\mathsf{list}\ \alpha'$ (or just list, by η-equivalence) as a proper substitute for β.

Unfortunately, as our earlier discussion of rootedness already suggested, contravariance complicates the lookup of multiple variables, because it can create dependencies between the results. Consider:

$$\begin{aligned}
\Xi &= \exists\beta_1\beta_2.\Sigma, \qquad \Sigma = \{\mathsf{F} : \forall\alpha.\{\mathsf{t} : [= \alpha : \Omega]\} \to_{\mathsf{P}} \{\mathsf{t} : [= \beta_1\,\alpha : \Omega]\},\\
&\qquad\qquad\qquad\quad \mathsf{G} : \forall\alpha.\{\mathsf{t} : [= \alpha : \Omega]\} \to_{\mathsf{P}} \{\mathsf{t} : [= \beta_2\,\alpha : \Omega]\}\}.\\
\Xi' &= \exists\beta_1'\beta_2'.\Sigma', \qquad \Sigma' = \{\mathsf{F} : \forall\alpha'.\{\mathsf{t} : [= \alpha' : \Omega]\} \to_{\mathsf{P}} \{\mathsf{t} : [= \beta_1'\,\alpha' : \Omega]\},\\
&\qquad\qquad\qquad\quad \mathsf{G} : \{\mathsf{t} : [= \beta_1'\ \mathsf{int} : \Omega]\} \to_{\mathsf{P}} \{\mathsf{t} : [= \beta_2' : \Omega]\}\}.
\end{aligned}$$

If we want to check $\Xi \leqslant \Xi'$, then looking up β_1', β_2' independently would deliver

$$\begin{aligned}
\mathrm{lookup}_{\beta_1'}(\Sigma, \Sigma') &\uparrow \lambda\alpha'.\beta_1\,\alpha'\\
\mathrm{lookup}_{\beta_2'}(\Sigma, \Sigma') &\uparrow \beta_2\,(\beta_1'\ \mathsf{int}).
\end{aligned}$$

The solution for β_2' still contains an occurrence of β_1', which we need to substitute away. Consequently, as in the definition of rootedness, we have to respect the quantification order of the existential variables (like those from Ξ' above) and perform their lookup in this order, substituting types as we go. As explained earlier, the definition of rootedness ensures that quantification order corresponds to dependency order.

In fact, the lookup rules, in the case of multiple variables and of functors, also contain explicit side conditions that check that the returned type(s) do not contain the looked-up variable(s) themselves. The main reason for these side conditions is technical: building them into the lookup judgment removes mutual interdependencies between various properties we prove below. In practice, they are implied by rootedness.

Because the new definition of lookup is more complicated, its "simple" properties are a little bit less simple than before (cf. Lemma 5.4):

Lemma 9.5 (*Simple properties of type lookup with applicative functors*)

1. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$ and $\overline{\alpha} \cap \mathrm{fv}(\Sigma) = \emptyset$, then $\mathrm{fv}(\overline{\tau}) \subseteq \mathrm{fv}(\Sigma) \cup \mathrm{fv}(\Sigma') - \overline{\alpha}$.
2. If $\mathrm{lookup}_{\pi}(\Sigma, \Sigma') \uparrow \tau$ and $\mathrm{head}(\pi) \notin \mathrm{fv}(\Sigma)$, then $\mathrm{fv}(\tau) \subseteq \mathrm{fv}(\Sigma) \cup \mathrm{fv}(\Sigma') - \mathrm{head}(\pi)$.
3. If $\mathrm{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$ and $\overline{\alpha} \cap (\overline{\alpha}' \cup \mathrm{fv}(\overline{\tau}')) = \emptyset$,
   then $\mathrm{lookup}_{\overline{\alpha}}(\Sigma[\overline{\tau}'/\overline{\alpha}'], \Sigma'[\overline{\tau}'/\overline{\alpha}']) \uparrow \overline{\tau}[\overline{\tau}'/\overline{\alpha}']$.
4. If $\mathrm{lookup}_{\pi}(\Sigma, \Sigma') \uparrow \tau$ and $\mathrm{fv}(\pi) \cap (\overline{\alpha}' \cup \mathrm{fv}(\overline{\tau}')) = \emptyset$,
   then $\mathrm{lookup}_{\pi}(\Sigma[\overline{\tau}'/\overline{\alpha}'], \Sigma'[\overline{\tau}'/\overline{\alpha}']) \uparrow \tau[\overline{\tau}'/\overline{\alpha}']$.

(Moreover, in parts 3 and 4, the length of the derivation stays the same.)

The soundness statement also requires a more verbose formulation than before, and because of the contravariant lookup in the functor case, both parts are mutually dependent:

Theorem 9.6 (*Soundness of type lookup with applicative functors*)

1. Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \alpha \vdash \Sigma' : \Omega$. If $\text{lookup}_\pi(\Sigma, \Sigma') \uparrow \tau_1$, then $\Gamma, \alpha \vdash \pi : \kappa$ and $\Gamma \vdash \tau_1 : \kappa$.
   Furthermore, if $\Gamma \vdash \Sigma \leqslant \Sigma'[\tau_2/\alpha]$ for $\Gamma \vdash \tau_2 : \kappa_\alpha$ and $\pi = \alpha\ \overline{\alpha}_0$ (with $\alpha \cap \overline{\alpha}_0 = \emptyset$), then $\tau_1 = \tau_2\ \overline{\alpha}_0$.
2. Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Sigma' : \Omega$. If $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_1$, then $\overline{\Gamma \vdash \tau_1 : \kappa_\alpha}$.
   Furthermore, if $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau}_2$, then $\overline{\tau}_1 = \overline{\tau}_2$.

Proof

By simultaneous induction on the size of the derivation of the lookup. Interestingly, proving well-kindedness of the looked-up types requires slightly different inductive steps than proving the type equivalence(s). Part 1:

- Case $\text{lookup}_\pi([= \tau_1 : \kappa], [= \tau' : \kappa])$: Then $\pi = \tau'$. By inversion of well-kindedness, $\Gamma \vdash \tau_1 : \kappa$ and $\Gamma, \alpha \vdash \tau' : \kappa$. Furthermore, by inversion of subtyping, $\tau_1 = \tau'[\tau_2/\alpha]$, for which we know via substitution that $\tau'[\tau_2/\alpha] = \pi[\tau_2/\alpha] = \tau_2\ \overline{\alpha}_0$.
- Case $\text{lookup}_\pi([= \pi'' : \tau_3], [= \pi' : \tau'_3])$: Analogous.
- Case $\text{lookup}_\pi(\{\overline{l : \Sigma}\}, \{\overline{l' : \Sigma'}\})$: Then $\text{lookup}_\pi(\Sigma, \Sigma') \uparrow \tau_1$ for some $\Sigma \in \overline{\Sigma}$ and $\Sigma' \in \overline{\Sigma}'$. By inverting well-kindedness, $\Gamma \vdash \Sigma : \Omega$ and $\Gamma, \alpha \vdash \Sigma' : \Omega$. The first claim then follows by induction. Furthermore, by inverting subtyping, $\Gamma \vdash \Sigma \leqslant \Sigma'[\tau_2/\alpha]$, and the second claim likewise follows by induction.
- Case $\text{lookup}_\pi(\forall \overline{\alpha}_1.\Sigma_1 \to_\text{P} \Sigma_2, \forall \overline{\alpha}'_1.\Sigma'_1 \to_\text{P} \Sigma'_2)$: Then $\tau_1 = \lambda \overline{\alpha}'_1.\tau_3$ such that both $\text{lookup}_{\overline{\alpha}_1}(\Sigma'_1, \Sigma_1) \uparrow \overline{\tau}'_1$ with $\alpha \notin \text{fv}(\overline{\tau}'_1)$, and $\text{lookup}_{\pi\,\overline{\alpha}'_1}(\Sigma_2[\overline{\tau}'_1/\overline{\alpha}_1], \Sigma'_2) \uparrow \tau_3$. Let $\Gamma'_1 = \Gamma, \overline{\alpha}, \overline{\alpha}'_1$. First, inverting the kinding rules, $\Gamma, \overline{\alpha}_1 \vdash \Sigma_1/\Sigma_2 : \Omega$ and $\Gamma'_1 \vdash \Sigma'_1/\Sigma'_2 : \Omega$. For Σ_1, we can weaken to $\Gamma'_1, \overline{\alpha}_1 \vdash \Sigma_1 : \Omega$, which allows us to invoke the induction hypothesis for part 2 and conclude $\overline{\Gamma'_1 \vdash \tau'_1 : \kappa_{\alpha_1}}$. Because $\alpha \notin \text{fv}(\overline{\tau}'_1)$, the result can be strengthened to $\overline{\Gamma, \overline{\alpha}'_1 \vdash \tau'_1 : \kappa_{\alpha_1}}$.
  Let $\Gamma'_2 = \Gamma, \overline{\alpha}'_1$. Obviously, $\Gamma'_2 \vdash [\overline{\tau}'_1/\overline{\alpha}_1] : \Gamma, \overline{\alpha}_1$, and applying the substitution lemma, $\Gamma'_2 \vdash \Sigma_2[\overline{\tau}'_1/\overline{\alpha}_1] : \Omega$. We can also use the substitution lemma to reorder Γ'_1 and derive $\Gamma'_2, \alpha \vdash \Sigma'_2 : \Omega$. We can now invoke the induction hypothesis on the codomains and get $\Gamma'_2, \alpha \vdash \pi\ \overline{\alpha}'_1 : \kappa'$ and $\Gamma'_2 \vdash \tau_3 : \kappa'$. With Lemma 9.1, we know both $\Gamma'_2, \alpha \vdash \pi : \overline{\alpha}'_1 \to \kappa'$ and $\Gamma \vdash \lambda \overline{\alpha}'_1.\tau_3 : \overline{\alpha}'_1 \to \kappa'$. Given that the $\overline{\alpha}'_1$ are locally fresh by the usual variable convention, and thus don't occur in π, the former can be strengthened to $\Gamma, \alpha \vdash \pi : \overline{\alpha}'_1 \to \kappa'$ as required.
  To furthermore prove the type equivalence, we can invert the subtyping assumption, revealing $\Gamma'_2 \vdash \Sigma'_1[\tau_2/\alpha] \leqslant \exists \overline{\alpha}_1.\Sigma_1 \uparrow \overline{\tau}'_2$ and $\Gamma'_2 \vdash \Sigma_2[\overline{\tau}'_2/\overline{\alpha}_1] \leqslant \Sigma'_2[\tau_2/\alpha]$. The substitution lemma implies $\Gamma'_2 \vdash \Sigma'_1[\tau_2/\overline{\alpha}] : \Omega$. And we can apply weakening to kinding of Σ_1, such that $\Gamma'_2, \overline{\alpha}_1 \vdash \Sigma_1 : \Omega$. Using Lemma 9.5, $\text{lookup}_{\overline{\alpha}_1}(\Sigma'_1[\tau_2/\alpha], \Sigma_1[\tau_2/\alpha]) \uparrow \overline{\tau}'_1[\tau_2/\alpha]$, but by variable containment we actually know that $\Sigma_1[\tau_2/\alpha] = \Sigma_1$ and $\overline{\tau}'_1[\tau_2/\alpha] = \overline{\tau}'_1$. Because that modified lookup derivation is still shorter than the current one, we can invoke the induction hypothesis (part 2) for the type equivalence claim, and get $\overline{\tau}'_1 = \overline{\tau}'_2$. As a consequence, $\Sigma_2[\overline{\tau}'_2/\overline{\alpha}_1] = \Sigma_2[\overline{\tau}'_1/\overline{\alpha}_1]$. So we know about the codomain that $\Gamma'_2 \vdash \Sigma_2[\overline{\tau}'_1/\overline{\alpha}_1] \leqslant \Sigma'_2[\tau_2/\alpha]$. Consequently, the induction hypothesis (part 1) also implies $\tau_3 = \alpha\ \overline{\alpha}_0\ \overline{\alpha}'_1$. or, via η-equivalence, $\lambda \overline{\alpha}'_1.\tau_3 = \alpha\ \overline{\alpha}_0$.

Part 2:

- Case $\text{lookup}_{\epsilon}(\Sigma, \Sigma')$: There is nothing to show.
- Case $\text{lookup}_{\alpha,\overline{\alpha}'}(\Sigma, \Sigma')$: Then $\overline{\tau}_1 = \tau_1, \overline{\tau}'_1$ and $\text{lookup}_{\alpha}(\Sigma, \Sigma') \uparrow \tau_1$ with $\text{fv}(\tau_1) \cap \overline{\alpha}' = \emptyset$, and $\text{lookup}_{\overline{\alpha}'}(\Sigma, \Sigma'[\tau_1/\alpha]) \uparrow \overline{\tau}'_1$. By inverting well-kindedness, $\Gamma, \alpha, \overline{\alpha}' \vdash \Sigma' : \Omega$, which, via the substitution lemma, can be tweaked to $\Gamma, \overline{\alpha}', \alpha \vdash \Sigma' : \Omega$. At the same time, weakening gives $\Gamma, \overline{\alpha}' \vdash \Sigma : \Omega$. Invoking the induction hypothesis (part 1) yields $\Gamma, \overline{\alpha}', \alpha \vdash \alpha : \kappa$ and $\Gamma, \overline{\alpha}' \vdash \tau_1 : \kappa$. Inverting the former tells $\kappa = \kappa_{\alpha}$. And because the side condition says $\overline{\alpha}' \cap \text{fv}(\tau_1) = \emptyset$, the latter can be strengthened to $\Gamma \vdash \tau_1 : \kappa_{\alpha}$. We can invoke the substitution lemma to derive $\Gamma, \overline{\alpha}' \vdash \Sigma'[\tau_1/\alpha] : \Omega$, which is enough to invoke the induction hypothesis again and conclude $\overline{\Gamma \vdash \tau'_1 : \kappa_{\alpha'}}$ as well.
  Furthermore, for proving the type equivalence, inverting matching reveals $\Gamma \vdash \Sigma \leqslant \Sigma'[\tau_2, \overline{\tau}'_2/\alpha, \overline{\alpha}']$ such that $\Gamma \vdash \tau_2 : \kappa_{\alpha}$ and $\overline{\Gamma \vdash \tau'_2 : \kappa_{\alpha'}}$. And because $\tau_2, \overline{\tau}'_2$ are all well-formed in plain Γ, the variables $\alpha, \overline{\alpha}'$ don't appear free in them, so $\Sigma'[\tau_2, \overline{\tau}'_2/\overline{\alpha}, \overline{\alpha}'] = \Sigma'[\overline{\tau}'_2/\overline{\alpha}'][\tau_2/\alpha] = \Sigma'[\tau_2/\alpha][\overline{\tau}'_2/\overline{\alpha}']$. Substitution on Σ' gives $\Gamma, \alpha \vdash \Sigma'[\overline{\tau}'_2/\overline{\alpha}'] : \Omega$. By application of Lemma 9.5, we have $\text{lookup}_{\alpha}(\Sigma[\overline{\tau}'_2/\overline{\alpha}'], \Sigma'[\overline{\tau}'_2/\overline{\alpha}']) \uparrow \tau_1[\overline{\tau}'_2/\overline{\alpha}']$. By the variable convention, $\text{fv}(\Sigma) \cap \overline{\alpha}' = \emptyset$. With the side condition on τ_1, thus, $\text{lookup}_{\alpha}(\Sigma, \Sigma'[\overline{\tau}'_2/\overline{\alpha}']) \uparrow \tau_1$. Because that still has a derivation shorter than the current one, we can invoke the induction hypothesis (part 1) again on the first lookup, to obtain that $\tau_1 = \tau_2$.
  Consequently, $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma'[\tau_2/\alpha]) \uparrow \overline{\tau}'_1$ also holds (and still has a derivation smaller than the current one), and so does $\Gamma, \overline{\alpha}' \vdash \Sigma'[\tau_2/\alpha] : \Omega$. Now, because $\Sigma'[\tau_2, \overline{\tau}'_2/\overline{\alpha}, \overline{\alpha}'] = \Sigma'[\tau_2/\alpha][\overline{\tau}'_2/\overline{\alpha}']$, we can apply U-MATCH to construct a derivation for $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}'.\Sigma'[\tau_2/\alpha] \uparrow \overline{\tau}'_2$. We can once more apply the induction hypothesis to that derivation, which produces $\overline{\tau}'_1 = \overline{\tau}'_2$. □

Corollary 9.7 (*Uniqueness of type lookup with applicative functors*)
Let $\Gamma \vdash \Sigma : \Omega$ and $\Gamma \vdash \exists \overline{\alpha}.\Sigma' : \Omega$ and $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau}$. If $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_1$ and $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}_2$, then $\overline{\tau}_1 = \overline{\tau}_2 = \overline{\tau}$.

Thanks to uniqueness, we can still read the lookup judgment as a quasi-deterministic algorithm.

Let us now turn to completeness, which becomes significantly more involved as well:

Theorem 9.8 (*Completeness of type lookup with applicative functors*)
Let $\Gamma \vdash \Sigma : \Omega$ valid and $\Gamma \vdash \exists \overline{\alpha}.\Sigma' : \Omega$ explicit.

1. If $\Gamma \vdash \Sigma \leqslant \Sigma'[\overline{\tau}/\overline{\alpha}]$ and $\overline{\Gamma \vdash \tau : \kappa_{\alpha}}$, and π rooted in Σ' avoiding $\overline{\alpha}$, with $\pi = \alpha\ \overline{\alpha}_1$ and $\alpha \in \overline{\alpha}$ and $\overline{\alpha} \cap \overline{\alpha}_1 = \emptyset$, then $\text{lookup}_{\pi}(\Sigma, \Sigma') \uparrow \tau\ \overline{\alpha}_1$ with $\tau = \alpha[\overline{\tau}/\overline{\alpha}]$.
2. If $\Gamma \vdash \Sigma \leqslant \exists \overline{\alpha}.\Sigma' \uparrow \overline{\tau}$, then $\text{lookup}_{\overline{\alpha}}(\Sigma, \Sigma') \uparrow \overline{\tau}$.

Proof
By simultaneous induction on the derivation of rootedness (implied by explicitness in part 2). Part 1:

- Case π rooted in $[= \tau' : \kappa]$: Then $\pi = \tau'$. Inverting subtyping, we know $\Sigma = [= \tau'' : \kappa]$ with $\tau'' = \tau'[\overline{\tau}/\overline{\alpha}]$. By substitution, $\pi[\overline{\tau}/\overline{\alpha}] = \tau'[\overline{\tau}/\overline{\alpha}]$, and hence

transitively, $\tau'' = \pi[\overline{\tau}/\overline{\alpha}] = (\alpha\ \overline{\alpha}_1)[\overline{\tau}/\overline{\alpha}] = \tau\ \overline{\alpha}_1$. So $\mathrm{lookup}_\pi([= \tau'' : \kappa], [= \tau' : \kappa]) \uparrow \tau\ \overline{\alpha}_1$.

- Case π rooted in $[= \pi' : \tau]$: Analogous.
- Case π rooted in $\{\overline{l' : \Sigma'}\}$: Then π rooted in $\{\overline{l' : \Sigma'}\}.l$ avoiding $\overline{\alpha}$. Inverting subtyping, we know $\Sigma = \{\overline{l : \Sigma}\}$ and $\Gamma \vdash \{\overline{l : \Sigma}\}.l \leqslant \{\overline{l' : \Sigma'}\}.l[\overline{\tau}/\overline{\alpha}]$. Inverting well-typedness and validity/explicitness, $\Gamma \vdash \{\overline{l : \Sigma}\}.l : \Omega$ valid and $\Gamma, \overline{\alpha} \vdash \{\overline{l' : \Sigma'}\}.l : \Omega$ explicit. Then by invoking the induction hypothesis, $\mathrm{lookup}_\pi(\{\overline{l : \Sigma}\}.l, \{\overline{l' : \Sigma'}\}.l) \uparrow \tau\ \overline{\alpha}_1$.
- Case π rooted in $\forall\overline{\alpha}'_1.\Sigma'_1 \to_\mathrm{P} \Sigma'_2$: Then $\pi\ \overline{\alpha}'_1$ rooted in Σ'_2 avoiding $\overline{\alpha}$ and $\mathrm{fv}(\Sigma'_1) \cap \overline{\alpha} = \emptyset$. Let $\Gamma' = \Gamma, \overline{\alpha}'_1$. Inverting subtyping, we know $\Sigma = \forall\overline{\alpha}_1.\Sigma_1 \to_\mathrm{P} \Sigma_2$ and $\Gamma' \vdash \Sigma'_1[\overline{\tau}/\overline{\alpha}] \leqslant \exists\overline{\alpha}_1.\Sigma_1 \uparrow \overline{\tau}_1$ and $\Gamma' \vdash \Sigma_2[\overline{\tau}_1/\overline{\alpha}_1] \leqslant \Sigma'_2[\overline{\tau}/\overline{\alpha}]$. Moreover, inverting well-typedness and validity/explicitness gives $\Gamma, \overline{\alpha}, \overline{\alpha}'_1 \vdash \Sigma'_1/\Sigma'_2 : \Omega$ explicit and, after weakening, $\Gamma', \overline{\alpha}_1 \vdash \Sigma_1/\Sigma_2 : \Omega$ explicit/valid, where $\overline{\alpha}_1$ rooted in Σ_1.
  By substitution and Lemma 9.3, $\Gamma' \vdash \Sigma'_1[\overline{\tau}/\overline{\alpha}] : \Omega$ valid. By typing rules and definition of explicitness, $\Gamma' \vdash \exists\overline{\alpha}_1.\Sigma_1 : \Omega$ explicit. Consequently, we can invoke the induction hypothesis (part 2), and have $\mathrm{lookup}_{\overline{\alpha}_1}(\Sigma'_1[\overline{\tau}/\overline{\alpha}], \Sigma_1) \uparrow \overline{\tau}_1$. Because of the variable side condition on functor rootedness, $\Sigma'_1[\overline{\tau}/\overline{\alpha}] = \Sigma'_1$. Moreover, because $\alpha \notin \mathrm{fv}(\Sigma_1) \cup \Sigma'_1[\overline{\tau}/\overline{\alpha}]$ by variable containment, Lemma 9.5 implies $\alpha \notin \mathrm{fv}(\overline{\tau}_1)$. That gives the first half of the definition of lookup in functors.
  Now, by soundness of type lookup, $\overline{\Gamma' \vdash \tau_1 : \kappa_{\alpha_1}}$. By substitution and Lemma 9.3, $\Gamma' \vdash \Sigma_2[\overline{\tau}_1/\overline{\alpha}_1] : \Omega$ valid. We invoke the induction hypothesis a second time (this time on part 1) and get $\mathrm{lookup}_{\pi\,\overline{\alpha}'_1}(\Sigma_2[\overline{\tau}_1/\overline{\alpha}_1], \Sigma'_2) \uparrow \tau\ \overline{\alpha}_1\ \overline{\alpha}'_1$. Consequently, we can derive $\mathrm{lookup}_\pi(\Sigma, \Sigma') \uparrow \lambda\overline{\alpha}'_1.\tau\ \overline{\alpha}_1\ \overline{\alpha}'_1$, and by η-equivalence, $\lambda\overline{\alpha}'.\tau_1\ \overline{\alpha}_1\ \overline{\alpha}' = \tau_1\ \overline{\alpha}_1$.

Part 2: inverting $\exists\overline{\alpha}.\Sigma'$ explicit implies $\overline{\alpha}$ rooted in Σ'.

- Case ϵ rooted in Σ': Then there is nothing to show.
- Case $\alpha, \overline{\alpha}'$ rooted in Σ': Then α rooted in Σ' avoiding $\alpha, \overline{\alpha}'$, and $\overline{\alpha}'$ rooted in Σ'. Inverting matching implies $\Gamma \vdash \Sigma \leqslant \Sigma'[\tau, \overline{\tau}'/\alpha, \overline{\alpha}']$ with $\Gamma \vdash \tau : \kappa_\alpha$ and $\overline{\Gamma \vdash \tau' : \kappa_{\alpha'}}$.
  From inverting well-typedness and explicitness we get $\Gamma, \alpha, \overline{\alpha}' \vdash \Sigma' : \Omega$ explicit. Let $\pi = \alpha$. Then we can invoke part 1 of the induction hypothesis to get $\mathrm{lookup}_\alpha(\Sigma, \Sigma') \uparrow \tau$. By variable containment, $\mathrm{fv}(\tau) \cap \overline{\alpha}' = \emptyset$.
  By substitution and Lemma 9.3, $\Gamma, \overline{\alpha}' \vdash \Sigma'[\tau/\alpha] : \Omega$ explicit and $\overline{\alpha}'$ rooted in $\Sigma'[\tau/\alpha]$, and so, $\Gamma \vdash \exists\overline{\alpha}'.\Sigma'[\tau/\alpha] : \Omega$ explicit. Because $\Gamma \vdash \tau : \kappa_\alpha$ and $\overline{\Gamma \vdash \tau' : \kappa_{\alpha'}}$, we know via variable containment that $\Sigma'[\tau, \overline{\tau}'/\alpha, \overline{\alpha}'] = \Sigma'[\tau/\alpha][\overline{\tau}'/\overline{\alpha}']$. With rule U-MATCH we can then construct the derivation $\Gamma \vdash \Sigma \leqslant \exists\overline{\alpha}'.\Sigma'[\tau/\alpha] \uparrow \overline{\tau}'$. With that, we can invoke part 2 of the induction hypothesis, to also get $\mathrm{lookup}_{\overline{\alpha}'}(\Sigma, \Sigma') \uparrow \overline{\tau}'$. □

As before, this property is sufficient to imply decidability of matching. (In addition, when we apply the matching rule U-MATCH algorithmically, we do not actually need to check the rule's side condition on the well-formedness of the types we have looked up, because it is already implied by soundness of lookup.)

Corollary 9.9 (*Decidability of matching with applicative functors*)
Assume that Γ is well-formed and valid, and also that $\Gamma \vdash \tau \leqslant \tau' \rightsquigarrow f$ is decidable for types well-formed under Γ. If Σ valid and Ξ explicit, and both are well-formed under Γ, then $\Gamma \vdash \Sigma \leqslant \Xi \uparrow \overline{\tau} \rightsquigarrow f$ is still decidable in the presence of applicative functors and the relaxed definition of rootedness from Figure 41.

Decidability of elaboration then follows as well, even though the elaboration rules under the applicative functor extensions are no longer purely syntactic: rules M-FUNCT-I and M-FUNCT-P overlap. However, they have disjoint premises, and thus the overlap does not induce any non-determinism. In the case of the multiple rules for value bindings, we have ensured the absence of overlap via syntactic side conditions.

Corollary 9.10 (*Decidability of elaboration with applicative functors*)
Under valid and well-formed Γ, provided we can (simultaneously) show that core elaboration is decidable, then all judgments of module elaboration with applicative functors are decidable, too.

Remark. At this point, the alert reader may ask: Where did the alleged simplicity go? It is true that the above decidability proof is not as simple anymore. However, we like to make a couple of observations.

First, the complexity witnessed above is only concerned with (signature matching for) applicative functors. The basic system from Sections 2–6, with generative functors only, is not affected. It is not completely surprising that applicative functors are more complex, considering the difficulties they have caused historically.

Second, the *declarative* semantics of the system *with* applicative functors is only mildly more involved than that of the basic system. From our perspective, the rules are still fewer and smaller than in any of the previous accounts of applicative functors — especially considering that they also do more. Moreover, the soundness proof from Section 9.1 is not substantially harder than the one for the basic system (Section 5.1) — and that arguably is all that is needed to understand the type system.

What gets more complicated is the *algorithm* to implement type lookup (Section 9.2) — or rather, the proof that this algorithm (which, by itself, is only a few lines of code) is complete. However, this algorithm arguably is only relevant to implementors, and its correctness proof only interesting to experts.

It is also worth noting that a fair amount of the encompassing complexity may actually be incidental. It is mainly due to the fact that our rules, unlike in most other systems, separate type lookup from subtyping. We chose this design because it makes the declarative subtyping rules pleasantly minimalist. For the basic system it also makes for an almost trivial lookup algorithm (Section 5.2). However, with the generalization to applicative functors, this factoring leads to a more complicated algorithm: in that system, lookup and subtyping become intertwined, which means that to separate them, lookup has to duplicate some of the work of subtyping, and its correctness proof needs to make sure that both algorithms operate in sync. The issue of rootedness could be avoided by decorating semantic signatures with "locators" (compare with Rossberg & Dreyer (2013)). A more traditional, interleaved, and

algorithmic definition of matching would eliminate the need for a correctness proof altogether (while slightly complicating the declarative semantics and its soundness proof). We leave further exploration of this option to future work.

Finally, it is also worth pointing out that our novel tracking of dynamic purity and dynamic module equivalence (Section 8) turns out to be only a minor extension to the system. In particular, it does not affect most of the definitions or proofs in a significant way — since value paths are modeled as phantom types, they are handled by the exact same mechanisms as ordinary abstract types.

10 Mechanization in Coq

One of our original motivations for the F-ing approach was that a simpler semantics for modules would be an easier starting point for language mechanization. As a proof of concept, we embarked on mechanizing the elaboration semantics of Sections 4 and 6 (but omitting normalization), and proved the soundness result of Theorem 5.1, but including module packages.

We did so using Coq (Coq Development Team, 2007) and the locally nameless approach (LN) of Aydemir *et al.* (2008). (There is no reason we could not have used other proof assistants such as Twelf or Isabelle; but we were interested in learning Coq and testing the effectiveness of the locally nameless approach.) This effort required roughly 13,000 lines of Coq code. As inexpert users of Coq, we made little use of automation, so most likely, the proofs could easily be shortened significantly.

As with any mechanization, there are some minor differences compared with the informal system. Our mechanized F_ω 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_ω 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_ω variables, and instead use a translation environment that twins source identifiers (which may be shadowed) with locally fresh F_ω 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.* (2007) 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 9), and separately defines the interpretation of semantic signatures as System F_ω types by an inductive definition (again much like the syntactic sugar definitions in Figure 9). 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_ω

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. Although we implemented the *n*-ary binders as derived forms over the unary ones provided by basic F_ω, we still needed derived lemmas for *n*-ary substitution (substituting locally closed terms for free names) and *n*-ary open (for opening binders with locally closed terms). Then we needed lemmas relating the commutation of all the combinations of *n*-ary and unary operations. The final straw was dealing with rules (notably for sequencing of binding and declarations) that required us to extend the scope of bindings over terms from subderivations. Doing this the recommended way requires the introduction of a third family of *closing* operations (the inverse of open), for turning named variables back into bound indices, together with a plethora of lemmas needed to actually reason about them (again with unary and *n*-versions of close and all possible commutations). We managed to work around these two cases by expressing the desired properties indirectly using additional (and thus unsatisfactory) premises stipulating equations between opened terms.

In the end, out of a total of around 550 lemmas, approximately 400 were tedious "infrastructure" lemmas; only the remainder had direct relevance to the meta-theory 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 classes (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. It would, however, be interesting to see whether more recent proposals to streamline the LN approach (Aydemir *et al.*, 2009) could significantly shorten larger developments like ours, without obscuring the presentation.

Despite the tedium, the mechanization still turned out to be relatively straightforward overall, and did not require any technical ingenuity. We believe that a Coq user with more experience than us (or somebody with respective experience using another proof assistant) but without specialist background in modules, could easily have carried it out without much effort.

11 Related work and discussion

The literature on ML module semantics is voluminous and varied. We will therefore focus on the most closely related work. A more detailed history of various accounts of ML-style modules can be found in Chapter 2 of Russo's thesis (1998; 2003).

Existential types for ADTs. Mitchell & Plotkin (1988) 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 elaboration of sealed structures, and more directly, in our support for modules as first-class values (Section 6), both of which are simply existential packages.

Cardelli & Leroy (1990) explained how to interpret the dot notation, which arises naturally when defining ADTs as modules, via a program transformation into uses of existentials. The idea is to unpack every existential immediately, such that the scope of the unpack matches the scope of the module definition. Our elaboration's use of unpacking and repacking can be viewed as a more compositional extension of this basic idea.

Dependent type systems for modules. In a very influential position paper, MacQueen (1986) 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 (Harper *et al*., 1990; Harper & Mitchell, 1993; Harper & Lillibridge, 1994; Leroy, 1994; Leroy, 1995; Leroy, 1996; Shao, 1999; Dreyer *et al*., 2003; Dreyer, 2005).

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 (1994) and Leroy (1994). 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 *et al.* (1990) 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 *et al.* (2003) 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*. Russo (1998) describes and formalizes a similar notion of "generalized path", with an analogous type-based restriction, as part of his system of higher-order functors. But the motivation is solely the ability to express paths like (F M).t, whereas for F-ing modules, we harvest their expressive power as a way of simplifying the language and its rules.

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 $F_{\leqslant}$ by Ghelli & Pierce (1998), 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.,* Lillibridge (1997) and Leroy (2000)) or else require (at least in some cases) extra signature annotations when leaving the scope of an abstract type (*e.g.,* Shao (1999), Dreyer *et al.* (2003)). 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*; at that point, we 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 (*i.e.,* it is *valid*, but not *explicit*, as defined in Section 5.2). Harper & Stone (2000) exploit elaboration in a similar fashion and to similar ends. One downside of this approach, some (*e.g.,* Shao (1999)) would argue, is that one loses "fully syntactic" signatures — *i.e.,* the ability to express the full static information about any module using a syntactic signature, and thus typecheck the module independently from the context in which it is used. But it is not clear that in practice this is really such a big deal, because a programmer can always avoid "non-syntactic" signatures by either adding a binding or an explicit signature annotation. In fact, Shao's approach to ruling out non-syntactic signatures would simply amount to restricting the projection rule M-DOT (Figure 14) in the same way as the path rule P-MOD (Figure 17) in our system, thereby *forcing* the programmer to take these measures.

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.,* Leroy (1996)) or were lacking some key component, such as a dynamic semantics (*e.g.,* Russo (1998)). 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 in Section 4 is essentially a conservative extension of SML's, as well as the generative fragment of Moscow ML (Russo, 2003).

Higher-order modules and applicative functors. The main way in which the language defined in Section 4 diverges from Standard ML is its support for higher-order modules, which constitute a relatively simple extension if one sticks to the generative semantics for functors. (Our semantics for higher-order modules in that section is similar to that of Leroy (1994; 1996) and Harper & Lillibridge (1994).) However, as a number of researchers noted in the early years of ML modules, the generative semantics is also fairly restrictive, because it assumes conservatively that any types specified abstractly in the result signature of an unknown functor will be generated anew every time the functor is applied. For example, if a higher-order functor H has a functor argument F of type $S \to S$, then H must account for the possibility that F is instantiated with an impure/generative functor and treat it as such during the typechecking of H's body, even though H may in fact be instantiated with a transparent F like the identity functor. Thus, under a generative semantics, abstraction over functor arguments can result in the rejection of seemingly reasonable programs due to insufficient propagation of type information.

Harper *et al.* (1990) were the first to propose the use of an applicative semantics (although they did not call it that) for achieving more flexible typechecking of higher-order functors. Leroy (1995) later popularized the idea of applicative functor semantics in the setting of a more fully realized module language, and it is his semantics that serves as the basis of OCaml's module system. In addition to better supporting higher-order modules, Leroy also motivated applicative semantics by the desire to treat semantically equivalent types (*e.g.,* integer sets) as equivalent, even if they were created by separate (but equal) instantiations of the same functor. Indeed, this latter motivation has in practice turned out to be arguably more compelling than the one concerning higher-order modules.

As we pointed out at the beginning of Section 8, the applicative functor semantics does not obviate generative semantics — both are appropriate in different instances — but constructing a language that supports and reconciles both forms has proven very difficult. Several proposals have been made (Shao, 1999; Dreyer *et al.*, 2003; Russo, 2003), but all of them suffer from breaking abstraction safety (cf. Section 8 for examples).

Our semantics of applicative functors in Sections 7 and 8 is novel and does not correspond directly to any existing account. As we explained in those sections, our motivation has been to provide an account of applicative functors that is (a) simple, (b) abstraction-safe, and (c) not overly conservative. To achieve simplicity, we adopt the adage that "applicative = pure" and "generative = impure". To achieve abstraction safety, we employ "stamps" (modeled as hidden abstract types) to statically track the identity of values, so that, for instance, the identity of the type of sets can depend (as it should) on the identity of the comparison function by which its elements are ordered. While this approach is necessarily conservative (in order to ensure decidability of typechecking), it is no more conservative than other abstraction-safe designs, and we have tried to be as liberal as possible by tracking identity at the level of individual value components.

Technically, our semantics for applicative functors is based closely on the formulation in Russo's thesis (Russo, 1998). Although we believe the applicative higher-order modules of Russo (1998) to be sound, their subsequent integration with Standard ML modules in Moscow ML turned out not to be (Dreyer *et al.*, 2002). In an attempt at backward compatibility, Moscow ML's early releases supported both applicative and generative higher-order functors. The typing relation was a seductively straightforward integration of both the generative and applicative rules. Dreyer's counterexample to type soundness is recounted by Russo (2003), together with a relatively simple, if unproven, fix. Even if a revised Moscow ML can be proven type sound, we claim that the marriage of applicative and generative functors presented in this article remains superior, by offering abstraction safety over and above simple type safety. In our refined system, only those abstract types whose invariants are guaranteed not to be tied to mutable state are rendered applicative. Moscow ML provides no such guarantee and freely allows the coercion of a generative into an applicative functor (by simple η-expansion).

We credit Biswas (1995) with discovering the skolemization technique for typing applicative higher-order functors: he used it to introduce higher-kinded *universal* quantifiers, parameterizing a higher-order functor on its argument's type dependencies in order to propagate actual dependencies at application of the functor (by implicit type application). The contribution of Russo (1998) was to additionally use higher-kinded *existential* quantifiers to abstract (and thus hide) concrete type dependencies at module sealing (by an implicit `pack`). Shao (1999) uses a similar skolemization technique, with the difference being that he collects all abstract types of a given module into a single variable of higher-order product kind (the module's "flexroot"), instead of quantifying them separately in a sequence of individual variables. Unfortunately, employing this "uncurried" formulation would necessitate jumping through extra hoops to handle the avoidance problem or constructs like **where** (besides relying on a mild extension to F_ω's type language).

We point out that the addition of applicative signatures alone (*i.e.,* the basic system from Section 4, extended with only the rules from Figure 26, but without the refined module elaboration from Figure 29) subsumes the more limited applicative functors of Shao (1999). Shao's system, like ours, distinguishes between opaque and transparent functor signatures, with the latter using higher-order type constructors

to abstract over static type dependencies. The difference is that in Shao's system, the *only* way to introduce an applicative functor is to seal a fully transparent functor by an applicative functor signature. This simple design choice has as an unfortunate side-effect: in Shao's system, unlike ours, a user cannot use sealing *within the body* of an applicative functor. The ability to use sealing inside an applicative functor is a desirable feature, since in principle one may wish to impose abstraction boundaries at any point inside a module, and indeed it is supported by most other designs, including our own. Furthermore, we depend crucially on this feature in our semantics of value sharing (via phantom types), which we depend on in turn to ensure abstraction safety. Specifically, we treat every value binding in a module as if it were a little sealed submodule, introducing an abstract phantom type to statically represent the identity of the value. In a system like Shao's, such an approach would automatically cause any functor (with a value component in it) to be treated as generative. Consequently, we do not know how to effectively enforce abstraction safety in a system like Shao's.

The module calculus of Dreyer *et al.* (2003) provides support for *both* the "strong" Shao-style sealing construct, which demands generativity of (immediately) enclosing functors, *and* a "weak" variant of sealing, which does not demand generativity and may thus be used inside applicative functors. Dreyer *et al.* account for these two variants in terms of a dichotomy between "dynamic effects" and "static effects". In our system, we have only retained the weak variant of sealing (adjusted to properly ensure abstraction safety), because our point of view is that the need for generativity has solely to do with the computational effects in the module being sealed, and that sealing *per se* is not a computational effect. Of course, if one really wished to "strongly seal" a pure module in our language, one could easily do so by inserting an impure no-op expression into the body of the module, thus inducing a *pro forma* effect. But we see no compelling reason for wanting to strongly seal a pure module.

An alternative semantics for higher-order functors was proposed by MacQueen & Tofte (1994), but it relied fundamentally on the idea of re-elaborating a functor's body at each application. In recent work, Kuan & MacQueen (2010) have investigated how to account for such a semantics in a more satisfactory way by tracking the "static effects" of higher-order functors in an "entity calculus". However, it remains unclear how to reconcile their approach, which underlies the module system of modern-day SML/NJ, with the tradition of type-theoretic accounts of ML modules to which "F-ing modules" belongs.

Interpreting ML modules into F_ω. We are certainly not the first to explain ML modules by translation into F_ω. Harper *et al.* (1990) give a "phase-splitting" translation of an early ML module calculus into an F_ω-like language, but do not yet deal with the crucial aspect of type generativity. As mentioned above, Cardelli & Leroy (1990) show how a calculus with dot notation — *i.e.,* with a mildly dependently-typed variant of System F existentials whose witness type is projectible on the type level — can be translated down to plain System F existential. Shao (1999) gives a multi-stage translation of his more advanced module calculus into a language called FTC, which is a variant of F_ω enriched with Cardelli/Leroy-style dot notation

and a restricted form of dependent products for expressing functors. However, he does not provide any translation of this language into F_ω itself, and it is not obvious how to extend the Cardelli/Leroy translation to FTC.

Shan (2004) presents a type-directed translation of the Dreyer–Crary–Harper module calculus (Dreyer *et al.*, 2003) into F_ω. His translation naturally uses some techniques similar to ours. In particular, his translation of signatures closely mirrors that of Russo (1998; 1999; 2003), and to translate module terms, he opens and repacks existentials in the same way we do. Our elaboration also borrows from Shan the technique of abstracting over the whole environment for the translation of applicative functors.

The biggest 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 translate it down to F_ω. This translation is directed by (and impossible without) the types and contexts from the source language. 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 (Russo, 1998; Russo, 1999). Russo started with the Definition of Standard ML (Milner *et al.*, 1997), 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 (1999), was that the state of generated type variables, threaded 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 (Russo, 1998; Russo, 2000).

We point the interested reader to Chapter 9 of Russo's thesis (1998; 2003) for an in-depth comparison with the non-dependent approach to modules that he pioneered (and that the F-ing approach is derived from), giving targeted examples to pinpoint the problems with dependently typed accounts and how they are avoided by this approach.

It is worth noting that 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 (Dreyer, 2007a), which he and coauthors later used as the target of an elaboration semantics for recursive modules (Dreyer, 2007b), mixin modules (Rossberg & Dreyer, 2013), and modules in the presence of type inference (Dreyer & Blume, 2007). 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 a non-recursive ML-like module system and we are only using plain F_ω as the target language.

Mechanization of module semantics. Lee *et al.* (2007) mechanized the meta-theory of full Standard ML, based on a variant of Harper–Stone elaboration given by Dreyer in his thesis (Dreyer, 2005). 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 meta-theoretic properties of their target language, which employs singleton kinds (Stone & Harper, 2006). 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 (Jones, 1996; Shields & Peyton Jones, 2002; Shan, 2004), using universal types for client-side data abstraction and existential types for implementor-side data abstraction. Several other authors (MacQueen, 1986; Harper & Pierce, 2005) 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 & Rémy (2009) have proposed directly programming in a variant of Dreyer's RTG (Dreyer, 2007a) (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 the present article 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.

12 Conclusion

In this article, we have shown that it is possible to give a direct, type-theoretic semantics for a comprehensive ML-style module system by elaboration into standard System F_ω. In so doing, we have also offered a novel account of applicative versus generative functor semantics (via a simple "pure/impure" distinction), which avoids the problems with abstraction safety that have plagued previous accounts.

Our main focus has been on semantics — a concern that we have not addressed in this article is implementation. As already alluded to in several places (such as Sections 4 and 7.3), we do not expect a real-world compiler to implement the F-ing rules verbatim. Obvious optimizations include: eliminating redundant administrative redexes at compile time, introducing type tuples to group semantic type parameters into single variables (effectively reconstructing structure stamps), lazily expanding type abbreviations, and minimizing the environments abstracted over by applicative functors. It also seems preferable for compilers to reconstruct user-friendly syntactic type expressions where possible when presenting semantic types to users. Most of

these techniques are well known, and we do not envision any particular difficulties in applying them to our system. But such concerns are outside the scope of this article.

Finally, while our semantics of ML modules accounts for almost all of the major features that can be found either in the literature or in the various implemented dialects of ML, there is one key feature we have left out: *recursive modules*. As Dreyer (2007a) has observed, the combination of recursion and ML-style abstract data types seems to demand an underlying type theory that goes beyond plain System F_ω, and moreover, in our opinion, doing recursive modules "right" requires abandoning some of the fundamental design decisions of traditional ML modules. Nevertheless, the basic ideas of the "F-ing" approach still apply: a semantics for recursive modules can be given using a variation of our elaboration, and targeting a language that is a conservative extension of F_ω. The first and last authors' work on MixML, a module system with recursive mixin composition, explores precisely that path (Rossberg & Dreyer, 2013).

References

Ahmed, A., Dreyer, D. & Rossberg, A. (2009) State-dependent representation independence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 350–353.

Atkey, R. (2012) Relational parametricity for higher kinds. *EACSL Annual Conference on Computer Science Logic*, pp. 46–61.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 3–15.

Aydemir, B., Weirich, S. & Zdancewic, S. (2009) *Abstracting syntax*. Technical report.

Biswas, S. K. (1995) Higher-order functors with transparent signatures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 154–163.

Cardelli, L. & Leroy, X. (1990) Abstract types and the dot notation. In *Programming Concepts and Methods*, IFIP State of the Art Reports. North Holland, pp. 479–504.

Coq Development Team. (2007) *The Coq proof assistant reference manual*. INRIA. Available at: `http://coq.inria.fr/`.

Dreyer, D. (2005) *Understanding and Evolving the ML Module System*. Ph.D. thesis, Carnegie Mellon University.

Dreyer, D. (2007a) Recursive type generativity. *J. Funct. Program.* **17**(4-5), 433–471.

Dreyer, D. (2007b) A type system for recursive modules. In *ACM SIGPLAN International Conference on Functional Programming*, 289–302.

Dreyer, D. & Blume, M. (2007) Principal type schemes for modular programs. In *European Symposium on Programming*, pp. 441–457.

Dreyer, D., Crary, K. & Harper, R. (2002) *Moscow ML's higher-order modules are unsound*. Posting to Types forum, 17 September.

Dreyer, D., Crary, K. & Harper, R. (2003) A type system for higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 236–249.

Elsman, M. (1999) *Program Modules, Separate Compilation, and Intermodule Optimisation*. Ph.D. thesis, University of Copenhagen.

Geuvers, H. (1992) The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *IEEE Symposium on Logic in Computer Science*, 453–460.

Ghelli, G. & Pierce, B. (1998) Bounded existentials and minimal typing. *Theor. Comput. Sci.* **193**(1-2), 75–96.

Goldfarb, W. D. (1981) The undecidability of the second-order unification problem. *Theor. Comput. Sci.* **13**, 225–230.

Harper, R. (2012) *Programming in Standard ML*. Available at: `http://www.cs.cmu.edu/~rwh/smlbook/`.

Harper, R. & Lillibridge, M. (1994) A type-theoretic approach to higher-order modules with sharing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 123–137.

Harper, R. & Mitchell, J. C. (1993) On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.* **15**(2), 211–252.

Harper, R., Mitchell, J. C. & Moggi, E. (1990) Higher-order modules and the phase distinction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 341–354.

Harper, R. & Pierce, B. C. (2005) Design considerations for ML-style module systems. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed), chapter 8, MIT.

Harper, R. & Stone, C. (2000) A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, MIT, 341–388.

Jones, M. P. (1996) Using parameterized signatures to express modular structure. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 68–78.

Kuan, G. & MacQueen, D. (2010) Engineering higher-order modules in SML/NJ. In *International Symposium on the Implementation and Application of Functional Languages*. Lecture Notes in Computer Science, Volume 6041, 2010, pp 218–235.

Launchbury, J. & Peyton Jones, S. L. (1995) State in Haskell. *LISP Symbol. Comput.* **8**(4), 293–341.

Lee, D. K., Crary, K. & Harper, R. (2007) Towards a mechanized metatheory of Standard ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 173–184.

Leifer, J., Peskine, G., Sewell, P. & Wansbrough, K. (2003) Global abstraction-safe marshalling with hash types. In *ACM SIGPLAN International Conference on Functional Programming*, 87–98.

Leroy, X. (1994) Manifest types, modules, and separate compilation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 109–122.

Leroy, X. (1995) Applicative functors and fully transparent higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 142–153.

Leroy, X. (1996) A syntactic theory of type generativity and sharing. *J. Funct. Program.* **6**(5), 667–698.

Leroy, X. (2000) A modular module system. *J. Funct. Program.* **10**(3), 269–303.

Lillibridge, M. (1997) *Translucent Sums: A Foundation for Higher-Order Module Systems*. Ph.D. thesis, Carnegie Mellon University.

MacQueen, D. B. (1986) Using dependent types to express modular structure. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 277–286.

MacQueen, D. B. & Tofte, M. (1994) A semantics for higher-order functors. In In *European Symposium on Programming*, 409–423.

Milner, R., Tofte, M. & Harper, R. (1990) *The Definition of Standard ML*, MIT.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (revised)*, MIT.

Mitchell, J. C. & Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* **10**(3), 470–502.

Montagu, B. & Rémy, D. (2009) Modeling abstract types in modules with open existential types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 354–365.

Paulson, L. C. (1996) *ML for the Working Programmer,* 2nd ed. Cambridge University Press.

Peyton Jones, S. (2003) *Wearing the hair shirt: a retrospective on Haskell.* Invited talk, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Available at: `http://research.microsoft.com/~simonpj`.

Romanenko, S., Russo, C. V. & Sestoft, P. (2000) *Moscow ML Version 2.0.* Available at: `http://www.dina.kvl.dk/~sestoft/mosml`.

Rossberg, A. (1999) *Undecidability of OCaml type checking.* Posting to Caml mailing list, 13 July.

Rossberg, A. & Dreyer, D. (2013) Mixin' up the ML module system. *ACM Trans. Lang. Syst.* **35**(1), Article 2, 2:1–2:84.

Rossberg, A., Le Botlan, D., Tack, G. & Smolka, G. (2004) Alice through the looking glass. In *Trends in Functional Programming*, 79–85.

Rossberg, A., Russo, C. V. & Dreyer, D. (2010) F-ing modules. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 89–102.

Russo, C. V. (1998) *Types for Modules.* Ph.D. thesis, LFCS, University of Edinburgh.

Russo, C. V. (1999) Non-dependent types for Standard ML modules. In *International Conference on Principles and Practice of Declarative Programming*, 80–97.

Russo, C. V. (2000) First-class structures for Standard ML. *Nord. J. Comput.* **7**(4), 348–374.

Russo, C. V. (2003) Types for modules. In *Electronic Notes in Theoretical Computer Science*, **60**, 3–421.

Sewell, P., Leifer, J. J., Wansbrough, K., Zappa Nardelli, F., Allen.-Williams, M., Habouzit, P. & Vafeiadis, V. (2007) Acute: High-level programming language design for distributed computation. *J. Funct. Program.* **17**(4–5), 547–612.

Shan, C.-C. (2004) *Higher-order modules in System F_ω and Haskell.* Technical Report. Available at: `http://www.cs.rutgers.edu/~ccshan/xlate/xlate.pdf`.

Shao, Z. (1999) Transparent modules with fully syntactic signatures. In *ACM SIGPLAN International Conference on Functional Programming*, 220–232.

Shields, M. & Peyton Jones, S. (2002) First-class modules for Haskell. In *International Workshop on Foundations of Object-Oriented Language*, pp. 28–40.

SML/NJ Development Team (1993) *Standard ML of New Jersey user's guide.* 0.93 ed., AT&T Bell Laboratories.

Stone, C. A. & Harper, R. (2006) Extensional equivalence and singleton types. *ACM Trans. Comput. Log.* **7**(4), 676–722.

Sulzmann, M., Chakravarty, M. M. T., Peyton Jones, S. & Donnelly, K. (2007) System F with type equality coercions. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 53–66.

Torgersen, M., Ernst, E. & Hanser, C. P. (2005) Wild FJ. In *International Workshop on Foundations of Object-Oriented Languages*, 1–15.

Wright, A. (1995) Simple imperative polymorphism. In *LISP and Symbolic Computation*, pp. 343–356.