Fully Abstract Module Compilation

Karl Crary

Carnegie Mellon University

Abstract

We give a translation suitable for compilation of modern module calculi supporting sealing, generativity, translucent signatures, applicative functors, higher-order functors and/or first-class modules. Ours is the first module-compilation translation with a dynamic correctness theorem. The theorem states that the translation produces target terms that are contextually equivalent to the source, in an appropriate sense. A corollary of the theorem is that the translation is fully abstract. Consequently, the translation preserves all abstraction present in the source. In passing, we also show that modules are a definitional extension of the underlying core language. All of our proofs are formalized in Coq.

1 Introduction

ML-style module calculi [10] provide powerful abstraction tools for various programming-in-the-large settings. Structures collect related type and term components, as well as other structures, together in a single expression. More than just a namespace management mechanism, modules can be made opaque, so that clients can use them only through their specified interfaces. Functors map modules to other modules, which allows the programmer to give an implementation of an abstract data type that depends parametrically on an unknown implementation of another.

In an untyped setting, compiling modules is not too difficult. One can simply erase the types and compile modular code as ordinary code, rendering modules as tuples or records, and functors and functions. Nevertheless, there are compelling reasons to prefer type-preserving compilation. Chief among them is the assistance that type structure offers in achieving compiler correctness. Simply as a matter of engineering, experience shows that it is very difficult to implement (accidentally) a type-preserving global transformation[1] that is incorrect and yet type correct.

Types can take on an even more significant role in compiler verification. The best correctness result for a compiler phase is *contextual equivalence,* which says that the source and translated terms are indistinguishable by any legal observation (in an appropriate sense). This is the most robust form of translation correctness because it applies regardless of how the translated code is used, be it translated further (vertical composition), linked with other code (horizontal composition), or used in other way. Types are essential to contextual equivalence: without type structure to circumscribe how the code may be used, too many observations are possible, making non-trivial contextual equivalence nearly impossible.

In this paper we give the first compilation strategy for ML-style modules that (1) supports the full array of abstraction mechanisms offered by the Standard ML module system (*i.e.,* structures, sealing, generative functors, and translucent signatures) [23] as well as some other important features (applicative functors, higher-order functors, and first-class modules), and (2) admits a contextual equivalence result between modules and their translation.

As a corollary of contextual equivalence, the module translation is *fully abstract:* Modules are equivalent exactly when their translations are equivalent. The left-to-right portion of this is also called *abstraction preservation.* It says that no legal target code can break any abstractions present in the source code.

This is particularly valuable today, as functional languages are increasingly becoming part of heterogeneous (multi-language) development environments. Full abstraction means that a programmer can be confident that his or her abstractions will be respected even by code linked in from another language, provided types are preserved and enforced [25].

Moreover, our compilation strategy admits *separate compilation,* meaning that units can be linked entirely in the target language, without reference to the source code that generated them. In practice, this means that every module expression—including functors—is independently translated to a target-language expression.

The first mathematical examination of the compilation of modules was undertaken by Harper, *et al.* [12]. Their interest was not in compilation *per se,* but in the equational theory of higher-order modules. Specifically, they wished to show that code that employs functors respects a compile-time/run-time *phase distinction*, meaning that code can always be type-checked without executing any code.

That the phase distinction is respected is not obvious *a priori.* A module contains both type and term components, and a functor is a function that maps modules to modules, so at first glance it appears as though the type components of the result can depend on the term components of the argument. If so, it would be necessary to execute the argument terms in order to determine the result types.

This does not happen because the module language is

[1] We refer here to such transformations as CPS or closure conversion, in contrast to *local* transformations such as arithmetic optimization, where types help much less.

structured in such a way that types can never depend on terms. Harper, *et al.* show that module expressions can be separated into two parts: a static (or "compile-time") phase and a dynamic (or "run-time") phase. Dynamic-phase expressions can depend on static-phase expressions, but not vice versa. This relies crucially on modules being second-class.

Harper, *et al.* go on to show that the static-phase equational theory is unaffected by the deletion of all dynamic-phase equations. But for our purposes, we are interested in their phase-separation algorithm itself. In their algorithm, every module M is split into two pieces $[c, e]$, in which c is a type constructor representing M's static portion, and e is a term representing its dynamic portion. Both pieces can be rendered without reference to the module language, so modules are compiled away.

Harper, *et al.'s* module language supported some important features, but it omitted other ones, most notably sealing and generativity. The only form of abstraction available was lambda abstraction, which meant that their language supported only client-side abstraction, not provider-side abstraction, which is more robust [2].

Subsequently, Shao [32] employed a similar algorithm in a compiler for Standard ML. Unlike Harper, *et al.,* Shao's source language supports sealing and generativity. However, the translation deals with them simply by removing them, so it does not preserve abstraction. (This was consistent with his purpose of facilitating cross-module inlining.)

Next, Shan [31] and Rossberg, *et al.* [27] proposed a different approach for compiling modules with sealing and generativity, in which sealing was treated as introducing an existential type. This provides a very natural treatment of abstraction [24].

Neither Shan nor Rossberg, *et al.* included any dynamic correctness result, much less full abstraction. In fact, neither one gave an evaluation semantics for the source language.[2] We resolve the question here in the negative; their translation is *probably not* fully abstract. More precisely, their translation is not fully abstract if the target's dynamic semantics provides any strict mechanism (*e.g.,* call-by-value or `seq`) capable of observing the termination of a function expression. However, the problem is easily corrected by a change to the target's static semantics.

This paper's main contribution is the first module-compilation algorithm with a dynamic correctness proof. The algorithm is given as a phase-separation translation: as in Harper, *et al.*'s original translation, we split each pure module into a type constructor and a term. The correctness result is as strong as possible: we show that modules are translated to contextually equivalent terms, and we conclude, as a corollary, that the translation is fully abstract.

The most obvious obstacle to the contextual equivalence result is that the source expression and its translation belong to different types, different syntactic classes, and indeed different languages. The difference in languages is easily dealt with: our target language is a strict subset of our source language, obtained by omitting all module-oriented forms. Thus, we can compare a source expression and its translation simply by taking the translation as another source expression.

To mediate the difference in types and syntactic classes, we show that we can define a pair of functions Snd and Join that convert back-and-forth between modules and terms of appropriate types. The result then states that if a module M is phase-separated into static portion c and dynamic portion e, then $\mathsf{Join}[c, e]$ is contextually equivalent to M. Since Snd and Join are inverses, one may equivalently say that $\mathsf{Snd}\,M$ is contextually equivalent to e.

If Join merely traversed e to reverse the phase-separation translation, this result would be trivial. It does not. The important fact about Snd and Join is they are defined *within the grammar of the language.* They are not able to traverse the syntax of an expression, and thus they can only interact with an expression through the interface given by its type. Their purpose is to coerce between interfaces.

One observation that we can draw from this result is that modules are a definitional extension of the core language. Put less formally, the module language can be viewed as merely a very convenient syntax for writing modular programs. A practical consequence of this observation is that some useful module features (notably first-class modules [29]) need not be included as distinct features to be compiled, because they are already definitional extensions of the language.

Our development proceeds as follows: First, we summarize our module calculus and state some preliminary results regarding it (Section 2). Our module calculus is nearly identical to Crary [2], except we add a value restriction for polymorphic functions. Second, we give the phase separation algorithm (Section 3). Third, we define contextual equivalence (Section 4). This is adapted from Crary, with small changes to deal with the value restriction. We also state some useful lemmas about contextual equivalence, corollaries of the development of logical equivalence in Crary. Finally we prove our correctness results (Section 5): We define Snd and Join, show that they are inverses, show that modules translate to contextually equivalent terms (modulo Join), and conclude full abstraction as a corollary.

All of our proofs are formalized in Coq, and may be found at:

`www.cs.cmu.edu/~crary/papers/2018/famc-formal.tgz`

2 The Module Calculus

Our module calculus is nearly identical to Crary [2], which is adapted from that of Dreyer [5], which in turn builds on a long line of prior work on module calculi [21, 12, 22, 9, 17, 18, 15, 34, 28, 33, 6]. The sole change from Crary is the addition of a value restriction to polymorphic functions. (As we will show, the value restriction is necessary for our translation to be fully abstract.) The module calculus itself is not a research contribution of this paper; our purpose here is just to lay the foundation for our phase-separation algorithm.

The module calculus is intended to account for the fundamental elements of a module system that are vital for data abstraction [2]. However, it does not account for all of the convenience features of modern module systems: Fields of modules are identified by position, not by name. Coercion of a module to a specified signature (*i.e.,* by drop-

[2] This was understandable in both cases. Shan was working from a pre-existing type theory [6] that did not commit to a specific dynamic semantics. Rossberg, *et al.* were *defining* a module system by giving an elaboration of ML modules into a module-free core language, so the source language necessarily had no independent semantics at all. (Nevertheless, the question remains whether the elaboration realizes the programmer's expectations.)

k	::=	1	unit kind
	|	T	types
	|	$\mathsf{S}(c)$	singleton kind
	|	$\Pi\alpha{:}k.k$	dependent functions
	|	$\Sigma\alpha{:}k.k$	dependent pairs
$c,$	::=	α	
τ	|	$\star$	unit constructor
	|	$\lambda\alpha{:}k.c \mid c\,c$	lambda, application
	|	$\langle c, c\rangle$	pair
	|	$\pi_1 c \mid \pi_2 c$	projection
	|	unit	unit type
	|	$\tau_1 \to \tau_2$	functions
	|	$\tau_1 \times \tau_2$	products
	|	$\forall\alpha{:}k.\tau$	universals
	|	$\exists\alpha{:}k.\tau$	existentials
e	::=	x	
	|	$\star$	unit term
	|	$\lambda x{:}\tau.e \mid e\,e$	lambda, application
	|	$\langle e, e\rangle$	pair
	|	$\pi_1 e \mid \pi_2 e$	projection
	|	$\Lambda\alpha{:}k.e$	polymorphic fun.
	|	$e[c]$	polymorphic app.
	|	$\mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau$	existential package
	|	$\mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,e$	unpack
	|	$\mathsf{fix}_\tau\, e$	recursion
	|	$\mathsf{let}\, x = e\,\mathsf{in}\,e$	term binding
	|	$\mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e$	module binding
	|	$\mathsf{Ext}\,M$	extraction
M	::=	m	
	|	$\star$	unit module
	|	$(\!\lvert c \rvert\!)$	atomic module
	|	$(\!\lvert e \rvert\!)$	atomic module
	|	$\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M$	generative functor
	|	$M\,M$	generative app.
	|	$\lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M$	applicative functor
	|	$M \cdot M$	applicative app.
	|	$\langle M, M\rangle$	pair
	|	$\pi_1 M \mid \pi_2 M$	projection
	|	$\mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(M : \sigma)$	unpack
	|	$\mathsf{let}\, x = e\,\mathsf{in}\,M$	term binding
	|	$\mathsf{let}\,\alpha/m = M\,\mathsf{in}\,(M : \sigma)$	module binding
	|	$M :> \sigma$	sealing
σ	::=	1	unit signature
	|	$(\!\lvert k \rvert\!)$	atomic signature
	|	$(\!\lvert \tau \rvert\!)$	atomic signature
	|	$\Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma$	generative functors
	|	$\Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma$	applicative functors
	|	$\Sigma\alpha{:}\sigma.\sigma$	pairs
Γ	::=	ϵ	empty context
	|	$\Gamma, \alpha{:}k$	constr. hypothesis
	|	$\Gamma, x{:}\tau$	term hypothesis
	|	$\Gamma, \alpha/m{:}\sigma$	module hypothesis

Figure 1: Syntax

ping fields, reordering fields, or monomorphizing polymorphic functions) is not done automatically; to do so creates complications that are orthogonal to the abstraction concerns we focus on here. Like nearly all module calculi (but not Harper and Mitchell [11]), sharing specifications are not primitive, but are taken as syntactic sugar over translucent signatures. The `open` declaration, which is useful in practice but makes little sense in a type theory with a proper notion of binding, is not supported. First-class modules (supported in some dialects of ML [29]) are not explicitly supported (but see Section 6). We also do not support recursive modules [3, 30, 5] (supported in some dialects of ML [30, 19]); the abstraction implications of recursive modules are not yet well examined. All of these features can be implemented by elaboration [15, 6, 5], but those elaborated aspects of the language fall outside the results we prove here.

Most significantly, we follow earlier work [15, 6, 27] by using elaboration to address the avoidance problem [6], in which the structure of a program mandates that type variables leave scope, but there is no best way for a module's type to avoid those variables. This arises most importantly in sealed arguments to functors. No satisfactory type-theoretic treatment of the avoidance problem has yet been found. The elaboration involves retaining in "hidden" modules the type variables that are supposed to leave scope. Unfortunately, the hidden-ness of those modules is not enforceable after elaboration, which may lead to tangible compromises in data abstraction. To avoid this, the programmer must manually ascribe signatures whenever type variables leave scope. This preserves abstraction, but is inconvenient.

Syntax A key design goal of the module calculus is to respect the *phase distinction* [12], which means that the meaning of static phrases, and specifically their equivalence, can be determined without referring to dynamic phrases. The static syntactic classes are kinds (k), type constructors (c), and signatures (σ). The dynamic syntactic classes are terms (e) and modules (M). Signatures serve as the types of modules, and kinds serve (as usual) as the types of type constructors. The full syntax is given in Figure 1. (Figures 1, 2, and 4 are borrowed from Crary [2].)

Static phrases The constructor and kind expressions are the singleton kind calculus of Stone and Harper [35], with some minor extensions. Actual types belong to the kind T. (We often use the metavariable τ instead of c for actual types.) The unit constructor $\star$ belongs to the kind 1.

If τ is a type, we may form the singleton kind $\mathsf{S}(\tau)$, which contains precisely τ (and other types equivalent to it). This is used to model type definitions and sharing specifications in signatures. Thus, if $\tau : \mathsf{S}(\tau')$, we can conclude that τ and τ' are equivalent types. As a consequence, equivalence is context sensitive: α and τ are equivalent under the assumption $\alpha : \mathsf{S}(\tau)$, but not under $\alpha : \mathsf{T}$. As a further consequence, equivalence depends on the kind at which constructors are compared: $\lambda\alpha{:}\mathsf{T}.\alpha$ and $\lambda\alpha{:}\mathsf{T}.\tau$ are equivalent at $\mathsf{S}(\tau) \to \mathsf{T}$ but not at $\mathsf{T} \to \mathsf{T}$. Although the singleton kind primitive is restricted to types, one can use it to define singletons at any kind.

Since singletons allow kinds to depend on constructors, it is useful to support dependent kinds. The dependent form of function kinds is written $\Pi\alpha{:}k.k'$ and the dependent form of products is written $\Sigma\alpha{:}k.k'$. When α does not appear free in k', these are written using $k \to k'$ and $k \times k'$ respectively,

as usual.

Dynamic phrases The syntax for terms is largely standard. The recursion form $\mathsf{fix}_\tau\, e$ has the type τ, provided e has the type $(\mathsf{unit}{\to}\tau){\to}\tau$, and we evaluate $\mathsf{fix}_\tau\, e$ to $e(\lambda\_{:}\mathsf{unit}.\mathsf{fix}_\tau\, e)$.[3]

The module language contains static and dynamic atomic modules, generative and applicative functors, and pairs. An atomic module models a single field of a module: a static atom ($(\!|\, c \,|\!)$) contains a single constructor, and a dynamic atom ($(\!|\, e \,|\!)$) contains a single term. For example, the ML module:

```
struct
  type t = bool
  val x = true
end
```

could be elaborated $\langle (\!|\, \mathsf{bool} \,|\!), (\!|\, \mathsf{true} \,|\!) \rangle$. (Observe that, as noted above, the field names disappear when a module is elaborated.)

Abstraction is introduced using the form $M :> \sigma$, which *seals* M so that its components can be accessed only through the interface given by σ. In particular, if a signature gives the kind of a type field as T, that type is unknown and therefore abstract. Proper enforcement of type abstraction requires that sealing be viewed as a computational effect [6], and that types cannot be extracted from impure modules. Thus, to use the types from a sealed module, one must bind that module to a variable; variables are always pure.

We syntactically distinguish between generative and applicative functors. A generative functor can have sealing in its body, and calling it induces an effect. In contrast, an applicative functor cannot, and does not.

For technical reasons related to the avoidance problem [5, section 4.2.6], a module that either let-binds a module or unpacks an existential type must include an explicit signature annotation, and such modules are deemed to induce an effect.

Static portions, twinned variables, and phase separation To extract a term from a dynamic atom, one uses the form $\mathsf{Ext}\, M$. However, to extract a type constructor from a static atom, we employ a judgement rather than a syntactic form. The judgement $\mathsf{Fst}(M) \gg c$ says that c is the *static portion* of M.[4] In particular, $\mathsf{Fst}((\!|\, c \,|\!)) \gg c$.

The reason for this design, invented by Dreyer [5], is it removes any syntactic dependency of static phrases (such as types) on terms or modules. By disentangling the singleton kind calculus from all dynamic phrases, we are able to use Stone and Harper's singleton-kind metatheory [35] off the shelf.

For static extraction to work compositionally, we must be able to compute the static portion of any pure module, not only static atom literals. (Impure modules do not have static portions.) The rules for doing so are given in Figure 2. Two subtle points arise: First, by its nature, a generative functor returns nothing that is statically predictable. Therefore the static component of a generative functor is trivial. Second,

[3] Observe that the argument to e is a value, so this evaluation behaves acceptably in a call-by-value setting. That would not be the case with the evaluation to $e(\mathsf{fix}_\tau\, e)$ that would be implied by a more conventional typing.

[4] The name Fst is motivated by the connection to phase separation, which explicit renders modules as a pair of a static and dynamic component.

$$
\begin{array}{lcl}
\mathsf{Fst}(\sigma) & : & \mathit{kind} \\
\mathsf{Fst}(1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\{\!|\, k \,|\!\}) & \stackrel{\text{def}}{=} & k \\
\mathsf{Fst}(\{\!|\, \tau \,|\!\}) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\,\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2) \\
\mathsf{Fst}(\Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Sigma\alpha{:}\,\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2)
\end{array}
$$

$$\boxed{\Gamma \vdash \mathsf{Fst}(M) \gg c}$$

$$\frac{\alpha/m \in \mathrm{Dom}(\Gamma)}{\Gamma \vdash \mathsf{Fst}(m) \gg \alpha} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\star) \gg \star} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}((\!|\, c \,|\!)) \gg c}$$

$$\frac{}{\Gamma \vdash \mathsf{Fst}((\!|\, e \,|\!)) \gg \star} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M) \gg \star}$$

$$\frac{\Gamma, \alpha/m{:}\sigma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M) \gg \lambda\alpha{:}\,\mathsf{Fst}(\sigma).c}$$

$$\frac{\Gamma \vdash \mathsf{Fst}(M_1) \gg c_1 \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash \mathsf{Fst}(M_1 \cdot M_2) \gg c_1 c_2 \qquad \Gamma \vdash \mathsf{Fst}(\langle M_1, M_2 \rangle) \gg \langle c_1, c_2 \rangle}$$

$$\frac{\Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\pi_i M) \gg \pi_i c} \qquad \frac{\Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\mathsf{let}\, x = e\, \mathsf{in}\, M) \gg c}$$

Figure 2: Static portions

for any module variable m, we must be able to compute m's static portion, but of course that cannot be known, so we must have a type-constructor variable prepared to stand in for m's static portion. This leads to the concept of *twinned variables* [16]: every module variable m is twinned with a constructor variable α, written α/m, where α is the static portion of m.[5] However, when α is not used, we will often leave it out.

Twinned variables are recorded in the context and static portions depend on them, so the static portion judgement is written in its most general form as $\Gamma \vdash M \gg c$. However, when Γ is empty or can be determined from context, we often write $\mathsf{Fst}(M) \gg c$, or even just $\mathsf{Fst}(M)$ for the unique c such that $\mathsf{Fst}(M) \gg c$.

Our static-portion rules are very similar in spirit to the *non-standard equational rules* of Harper, *et al.* [12]. As in this work, Harper *et al.* gave rules for determining the static portions of modules as part of their static semantics, and also—*separately*—gave an algorithm for compiling modules into static and dynamic components. The main difference (apart from the extra features we support here), is that their "non-standard" rules determined not only a static portion, but also a dynamic portion, even though only the static portion ended up being relevant to typing judgements.

Signatures Signatures forms include signatures for atomic modules, generative and application functors, and pairs.

[5] An alternative [12, 5] is to use a naming convention to associate module and constructor variables, but this complicates binding and substitution. Sorting out those complications carefully leaves you with something very close to twinned variables.

$$
\begin{array}{lcl}
\mathsf{S}(c : k) & : & \mathit{kind} \\
\mathsf{S}(c : 1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{S}(c : \mathsf{T}) & \stackrel{\text{def}}{=} & \mathsf{S}(c) \\
\mathsf{S}(c : \mathsf{S}(c')) & \stackrel{\text{def}}{=} & \mathsf{S}(c) \\
\mathsf{S}(c : \Pi\alpha{:}k_1.k_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}k_1.\mathsf{S}(c\,\alpha : k_2) \\
\mathsf{S}(c : \Sigma\alpha{:}k_1.k_2) & \stackrel{\text{def}}{=} & \mathsf{S}(\pi_1 c : k_1) \times \mathsf{S}(\pi_2 c : [\pi_1 c/\alpha]k_2) \\
\\
\mathsf{S}(c : \sigma) & : & \mathit{signature} \\
\mathsf{S}(c : 1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{S}(c : (\!|\, k \,|\!)) & \stackrel{\text{def}}{=} & (\!|\, \mathsf{S}(c : k) \,|\!) \\
\mathsf{S}(c : \langle\!|\, \tau \,|\!\rangle) & \stackrel{\text{def}}{=} & \langle\!|\, \tau \,|\!\rangle \\
\mathsf{S}(c : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \\
\mathsf{S}(c : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\mathsf{S}(c\,\alpha : \sigma_2) \\
\mathsf{S}(c : \Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \mathsf{S}(\pi_1 c : \sigma_1) \times \mathsf{S}(\pi_2 c : [\pi_1 c/\alpha]\sigma_2)
\end{array}
$$

Figure 3: Higher-order singletons

The signatures for functors and for pairs are dependent, so one might expect to write them with a twinned left-hand-side, like $\Sigma\alpha/m{:}\sigma_1.\sigma_2$. However, since a module can never appear within a signature (signatures being static phrases), the m variable will never be used, so we don't bother to write it. (As usual, when α does not appear free in σ_2, we write $\Sigma\alpha{:}\sigma_1.\sigma_2$ as $\sigma_1 \times \sigma_2$.)

In a twinned binding $\alpha/m{:}\sigma$, or in the binding $\alpha{:}\sigma$ within a dependent signature, α stands for the static portion of some module belonging to σ. Thus, α will have kind $\mathsf{Fst}(\sigma)$, which stands for the static portion of σ. The definition of Fst appears in Figure 2. Whenever $m : \sigma$ and $\mathsf{Fst}(m) \gg c$, we have $c : \mathsf{Fst}(\sigma)$.

Higher-order singletons The primitive singleton kind $\mathsf{S}(c)$ is well-formed only when c is a type. However, singletons at higher kinds (written $\mathsf{S}(c : k)$) are definable using dependent kinds. For example, $\mathsf{S}(c : k_1 \to k_2) = \Pi\alpha{:}k_1.\mathsf{S}(c\,\alpha : k_2)$, since a constructor is equivalent to c in kind $k_1 \to k_2$ precisely when it takes an argument of kind k_1 and does with it whatever c does with it.

We can use the same technique to define singleton signatures. Suppose $c : \mathsf{Fst}(\sigma)$. Then the singleton signature $\mathsf{S}(c : \sigma)$ contains all pure modules in σ whose static component is equivalent to c. For instance $\mathsf{S}(c : (\!|\, k \,|\!)) = (\!|\, \mathsf{S}(c : k) \,|\!)$. The definitions of higher-order singletons are given in Figure 3. Note that when the static component of σ is trivial (*i.e.*, when $\mathsf{Fst}(\sigma) = 1$), $\mathsf{S}(c : \sigma) = \sigma$, since all static components are equivalent at kind 1.

Semantics The static semantics is given by several judgements summarized in Figure 4. The rules are exactly those from Crary [2], with the sole exception that the typing rule for polymorphic functions is replaced by the one given below, which includes a value restriction.

The top-level judgement is the typing judgement for modules, written $\Gamma \vdash_\kappa M : \sigma$, where the *purity class* κ is either P, indicating that the module is pure (unsealed), or I, indicating that it is impure (sealed). A "forget" rule allows pure modules to be viewed as impure.

$\vdash \Gamma\ \mathsf{ok}$	context formation
$\Gamma \vdash k : \mathsf{kind}$	kind formation
$\Gamma \vdash k \equiv k'$	kind equivalence
$\Gamma \vdash k \leq k'$	subkind
$\Gamma \vdash c : k$	constructor formation
$\Gamma \vdash c \equiv c' : k$	constructor equivalence
$\Gamma \vdash \sigma : \mathsf{sig}$	signature formation
$\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}$	signature equivalence
$\Gamma \vdash \sigma \leq \sigma'$	subsignature
$\Gamma \vdash e : \tau$	term formation
$\Gamma \vdash_\kappa M : \sigma$	module formation
$\Gamma \vdash \mathsf{Fst}(M) \gg c$	static portion

Figure 4: Static semantic judgements

The dynamic semantics is given by a standard, call-by-value, structured operational semantics, written $\Gamma \vdash e \mapsto e'$ and $\Gamma \vdash M \mapsto M'$. (It is convenient to be able to evaluate open terms, but in the typical case in which Γ is empty, we will omit the turnstile.) The value forms are:

$$
\begin{array}{rcl}
v & ::= & x \mid \star \mid \lambda x{:}\tau.e \mid \langle v, v\rangle \mid \Lambda\alpha{:}k.e \\
 & \mid & \mathsf{pack}\,[c, v]\,\mathsf{as}\,\exists\alpha{:}k.\tau \\
V & ::= & m \mid \star \mid (\!|\, c \,|\!) \mid \langle\!|\, v \,|\!\rangle \mid \langle V, V\rangle \\
 & \mid & \lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M \mid \lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M
\end{array}
$$

We write $e{\downarrow}$ or $M{\downarrow}$ to mean that e or M evaluates to a value.

Two rules illustrate issues that arise in the dynamic semantics. The evaluation rule for sealed modules immediately removes the seal:

$$\frac{}{\Gamma \vdash (M :> \sigma) \mapsto M}$$

In essence, this step is the computational effect that the type system tracks. Rules that substitute for module variables, such as the beta rules for functors, also must substitute for the twinned constructor variable, which is obtained using the Fst judgement:

$$\frac{\Gamma \vdash \mathsf{Fst}(V_2) \gg c_2}{\Gamma \vdash (\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M_1)V_2 \mapsto [c_2, V_2/\alpha, m]M_1}$$

This rule works because well-formed module values are always pure, so their static portion can always be obtained.

The value restriction In the typing rule for polymorphic functions, we require that the body be a static value:

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash sv : \tau}{\Gamma \vdash \Lambda\alpha{:}k.sv : \forall\alpha{:}k.\tau}$$

where a static value (sv) is like an ordinary value (v) except variables are excluded:

$$
\begin{array}{rcl}
sv & ::= & \star \mid \lambda x{:}\tau.e \mid \langle sv, sv\rangle \mid \Lambda\alpha{:}k.sv \\
 & \mid & \mathsf{pack}\,[c, sv]\,\mathsf{as}\,\exists\alpha{:}k.\tau
\end{array}
$$

We distinguish between static and ordinary values and use the former because we wish for well-formedness to be preserved by the substitution of any appropriately typed term—even a non-value—for a variable, and that is not the case

with ordinary values. This is necessary to have a useful contextual-equivalence functionality lemma.

The inclusion of the value restriction has no significant effect on the development of logical equivalence given in Crary [2], but it is essential for the phase separation translation to be fully abstract.

Metatheory We can now state some metatheoretic results from Crary [2]:

Lemma 2.1

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *then* $\Gamma \vdash \mathsf{Fst}(M) : \mathsf{Fst}(\sigma)$.
- *If* $\vdash \Gamma\ \mathsf{ok}$ *and* $\Gamma \vdash c : k$ *then* $\Gamma \vdash c : \mathsf{S}(c : k)$ *and* $\Gamma, \alpha{:}\mathsf{S}(c : k) \vdash \alpha \equiv c : k$.
- *If* $\vdash \Gamma\ \mathsf{ok}$ *and* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *then* $\Gamma \vdash_{\mathsf{P}} M : \mathsf{S}(\mathsf{Fst}(M) : \sigma)$.
- *If* $\Gamma \vdash_{\mathsf{I}} V : \sigma$ *then* $\Gamma \vdash_{\mathsf{P}} V : \sigma$.

Theorem 2.2 (Type preservation) *If* $\vdash \Gamma\ \mathsf{ok}$ *then:*

- *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e \mapsto e'$ *then* $\Gamma \vdash e' : \tau$.
- *If* $\Gamma \vdash_\kappa M : \sigma$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash_\kappa M' : \sigma$.
- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\sigma)$.

Theorem 2.3 (Progress) *If* $\vdash e : \tau$ *then either* e *is a value or takes a step. If* $\vdash_\kappa M : \sigma$ *then either* M *is a value or takes a step.*

Theorem 2.4 (Determinism)

- *If* $\Gamma \vdash e \mapsto e_1$ *and* $\Gamma \vdash e \mapsto e_2$ *then* $e_1 = e_2$.
- *If* $\Gamma \vdash M \mapsto M_1$ *and* $\Gamma \vdash M \mapsto M_2$ *then* $M_1 = M_2$.
- *If* $\Gamma \vdash \mathsf{Fst}(M) \gg c_1$ *and* $\Gamma \vdash \mathsf{Fst}(M) \gg c_2$ *then* $c_1 = c_2$.

3 Phase Separation

The target language for our translation is the subset of the source language obtained by deleting all the module-oriented forms. Note that the syntactic classes of kinds and constructors contain no module-oriented forms, and are thus unchanged. (This is an advantage of Dreyer's device [5] wherein Fst is a judgement, rather than a syntactic form.)

Using a fragment of the source as the target dramatically simplifies several aspects of the development: (1) We need not develop the metatheory of the target language; the source's metatheory applies directly. (2) The type translation is the identity. (3) Full abstraction is a direct consequence of the dynamic correctness theorem, since every target observation is a source observation, and every source observation can be translated to a target observation.

We express phase separation as a type-directed translation [13] using three judgements, one each for terms, pure modules, and impure modules. The term translation judgement is $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}$, meaning that $\Gamma \vdash e : \tau$ and $\bar{e}$ is a translation of e. Since the type translation is the identity, $\bar{e}$ will have the same type as e.

Different translation derivations can ostensibly result in different translations, but we can show that the translation is coherent [1], meaning that all of a term's translations are equivalent. Coherence is another simple corollary of dynamic correctness.

The term translation does very little, since most term forms do not pertain to modules. Its rules are given in Appendix A. The main judgements are the phase-separation judgements for modules. For pure modules, the judgement is written:

$$\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]$$

Here, M is separated into its static portion c and its dynamic portion e. Recall that the module calculus has its own internal concept of static components ($\mathsf{Fst}(M)$). These notions agree: if also $\Gamma \vdash \mathsf{Fst}(M) \gg c'$ then $c = c'$.

The types of c and e are governed by the phase separation judgement for signatures:

$$\sigma \rightsquigarrow [\alpha{:}k.\, \tau]$$

Here σ is separated into its static portion k and its dynamic portion τ (in which τ can refer to α, which stands for the actual static component). Thus, in the above, c belongs to kind k, and e belongs to type $[c/\alpha]\tau$.

The best way to understand the translation is through the phase-separation rules for signatures. A static atom is all static, so $(\!|\, k \,|\!) \rightsquigarrow [\_ : k.\, \mathsf{unit}]$. Conversely, a dynamic atom is all dynamic, so $\langle\!|\, \tau \,|\!\rangle \rightsquigarrow [\_{:}1.\, \tau]$. A trivial module is trivial in both components, so $1 \rightsquigarrow [\_{:}1.\, \mathsf{unit}]$.

In an applicative functor, the static portion maps the static portion of the argument to the static portion of the result, while the dynamic portion maps both portions of the argument to the dynamic portion of the result:

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\, \tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\, \tau_2]}{\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\beta : \Pi\alpha{:}k_1.k_2.\ \forall\alpha{:}k_1.\, [\alpha/\alpha_1]\tau_1 \to [\beta\alpha/\alpha_2]\tau_2]}$$

In a pair, the static (dynamic) portion is the pair of the static (dynamic) portions of the constituents:

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\, \tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\, \tau_2]}{\Sigma\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\beta : \Sigma\alpha{:}k_1.k_2.\ [\pi_1\beta/\alpha_1]\tau_1 \times [\pi_1\beta, \pi_2\beta/\alpha, \alpha_2]\tau_2]}$$

These rules are essentially the rules from Harper, *et al.* [12] (adapted for dependent kinds), but they did not account for generative functors. For generative functors, we adapt the translation of Shan [31] and Rossberg, *et al.* [27].

The tricky point is how to phase-separate impure (*i.e.,* sealed) modules. Our translation wishes to separate modules into their static and dynamic components, but sealed modules are not supposed to have a static component, since their abstract type components are notionally determined at run-time.[6] Any translation that makes them available statically will not preserve abstraction.

But once a sealed module has executed, its result value *will* be pure (recall part 4 of Lemma 2.1), and therefore it will have static and dynamic components. Thus, the appropriate translation of an impure module is a term that dynamically computes the static and dynamic portions of that eventual value. Thus we write:

$$\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e$$

where e belongs to $\exists\alpha{:}k.\tau$ (assuming $\sigma \rightsquigarrow [\alpha{:}k.\, \tau]$).

Now we can give the rule for a generative functor, which returns an impure module. Its static portion is trivial, while

[6] Indeed, when first-class modules are added, they are *actually* determined at run-time.

$\boxed{\sigma \rightsquigarrow [\alpha{:}k.\,\tau]}$

$$\overline{1 \rightsquigarrow [\_{:}1.\,\mathsf{unit}]} \qquad \overline{\langle\!|\, k \,|\!\rangle \rightsquigarrow [\_ : k.\,\mathsf{unit}]} \qquad \overline{\langle\!|\, \tau \,|\!\rangle \rightsquigarrow [\_{:}1.\,\tau]}$$

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]}{\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\_{:}1.\,\forall\alpha{:}k_1.\,[\alpha/\alpha_1]\tau_1 \to \exists\alpha_2{:}k_2.\tau_2]}$$

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]}{\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\beta : \Pi\alpha{:}k_1.k_2.\,\forall\alpha{:}k_1.\,[\alpha/\alpha_1]\tau_1 \to [\beta\alpha/\alpha_2]\tau_2]}$$

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]}{\Sigma\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\beta : \Sigma\alpha{:}k_1.k_2.\,[\pi_1\beta/\alpha_1]\tau_1 \times [\pi_1\beta, \pi_2\beta/\alpha, \alpha_2]\tau_2]}$$

Figure 5: The signature translation

its dynamic portion maps both portions of the argument to an existential package containing the static and dynamic portions of the module value that the functor eventually returns:

$$\frac{\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1] \quad \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]}{\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\_{:}1.\,\forall\alpha{:}k_1.\,[\alpha/\alpha_1]\tau_1 \to \exists\alpha_2{:}k_2.\tau_2]}$$

For convenience, the signature translation rules are collected in Figure 5. The module translation is given in Figures 6, and 7. Most of its rules are dictated by the signature phase-separation rules given above. We touch on a few points that are interesting or subtle:

The translation's calling convention is given by the context translation $\overline{\Gamma}$, which is formalized in Figure 8. Constructor and term bindings are translated directly, as one would expect. In a module binding α/m, we introduce two variables—a constructor variable α and a term variable $\hat{m}$—to represent the module's static and dynamic portions. Here we assume that there are infinitely many term variables, so for every module variable m we can associate a term variable $\hat{m}$ that is not otherwise used.

With the calling convention established, we can consider the translation rules for functors. For applicative functors, the rule states:

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\,\tau_1] \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, e]}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\lambda\alpha{:}k_1.c,\ \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e]}$$

The functor's static portion $\lambda\alpha{:}k_1.c$ takes the static portion of the functor argument ($\alpha{:}k_1$) to the static portion of the body (c). In accordance with the phase distinction, there is no dependence on the dynamic portion of the argument. The functor's dynamic portion takes the static and dynamic portions of the argument ($\alpha{:}k_1$ and $\hat{m}{:}\tau_1$)—thereby following the calling convention for $\alpha/m{:}\sigma_1$—to the dynamic portion of the body (e).

The applicative functor application rule states (in the pure case):

$$\frac{\Gamma \vdash_{\mathsf{P}} M_1 : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [c_1, e_1] \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_1 \rightsquigarrow [c_2, e_2]}{\Gamma \vdash_{\mathsf{P}} M_1 \cdot M_2 : [c_2/\alpha]\sigma_2 \rightsquigarrow [c_1 c_2, e_1\,[c_2]\,e_2]}$$

$\boxed{\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]}$

$$\frac{(\alpha/m : \sigma) \in \Gamma}{\Gamma \vdash_{\mathsf{P}} m : \mathsf{S}(\alpha : \sigma) \rightsquigarrow [\alpha, \hat{m}]}$$

$$\overline{\Gamma \vdash \star : 1 \rightsquigarrow [\star, \star]}$$

$$\frac{\Gamma \vdash c : k}{\Gamma \vdash \langle\!|\, c \,|\!\rangle : \langle\!|\, k \,|\!\rangle \rightsquigarrow [c, \star]} \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow \bar{e}}{\Gamma \vdash \langle\!|\, e \,|\!\rangle : \langle\!|\, \tau \,|\!\rangle \rightsquigarrow [\star, \bar{e}]}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\,\tau_1] \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{I}} M : \sigma_2 \rightsquigarrow e}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\star, \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e]}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\,\tau_1] \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, e]}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\lambda\alpha{:}k_1.c,\ \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M_1 : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [c_1, e_1] \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_1 \rightsquigarrow [c_2, e_2]}{\Gamma \vdash_{\mathsf{P}} M_1 \cdot M_2 : [c_2/\alpha]\sigma_2 \rightsquigarrow [c_1 c_2, e_1\,[c_2]\,e_2]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M_1 : \sigma_1 \rightsquigarrow [c_1, e_1] \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_2 \rightsquigarrow [c_2, e_2]}{\Gamma \vdash_{\mathsf{P}} \langle M_1, M_2\rangle : \sigma_1 \times \sigma_2 \rightsquigarrow [\langle c_1, c_2\rangle, \langle e_1, e_2\rangle]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Sigma\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [c, e]}{\Gamma \vdash_{\mathsf{P}} \pi_1 M : \sigma_1 \rightsquigarrow [\pi_1 c, \pi_1 e]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Sigma\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [c, e]}{\Gamma \vdash_{\mathsf{P}} \pi_2 M : [\pi_1 c/\alpha]\sigma_2 \rightsquigarrow [\pi_2 c, \pi_2 e]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} e : \tau \rightsquigarrow e_1 \quad \Gamma, x{:}\tau \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e_2]}{\Gamma \vdash_{\mathsf{P}} \mathsf{let}\, x = e\, \mathsf{in}\, M : \sigma \rightsquigarrow [c, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma_1 \rightsquigarrow [c, e] \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f}{\Gamma \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, f\,[c]\,e]}$$

Figure 6: The pure module translation

$\boxed{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e}$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1 : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow e_1 \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_1 \rightsquigarrow [c_2, e_2]}{\Gamma \vdash_{\mathsf{I}} M_1 M_2 : [c_2/\alpha]\sigma_2 \rightsquigarrow \mathsf{unpack}\,[\_, f] = e_1 \,\mathsf{in}\, f\,[c_2]\,e_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{I}} M_1 : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow e_1 \\ \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_1 \rightsquigarrow [c_2, e_2] \qquad \sigma_2 \rightsquigarrow [\beta{:}k_2.\,\tau_2]\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathsf{I}} M_1 \cdot M_2 : [c_2/\alpha]\sigma_2 \rightsquigarrow \mathsf{unpack}\,[\gamma, f] = e_1 \\ \qquad\qquad \mathsf{in}\,\mathsf{pack}\,[\gamma\, c_2, f\,[c_2]\,e_2] \\ \qquad\qquad\quad \mathsf{as}\,[c_2/\alpha](\exists\beta{:}k_2.\tau_2)\end{array}}$$

$$\frac{\begin{array}{cc}\Gamma \vdash M_1 : \sigma_1 \rightsquigarrow e_1 & \Gamma \vdash M_2 : \sigma_2 \rightsquigarrow e_2 \\ \sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1] & \sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]\end{array}}{\begin{array}{l}\Gamma \vdash \langle M_1, M_2\rangle : \sigma_1 \times \sigma_2 \rightsquigarrow \mathsf{unpack}\,[\alpha_1, x_1] = e_1 \\ \qquad\qquad \mathsf{in}\,\mathsf{unpack}\,[\alpha_2, x_2] = e_2 \\ \qquad\qquad \mathsf{in}\,\mathsf{pack}\,[\langle\alpha_1, \alpha_2\rangle, \langle x_1, x_2\rangle] \\ \qquad\qquad\quad \mathsf{as}\,\exists\beta{:}(k_1 \times k_2).\,[\pi_1\beta/\alpha_1]\tau_1 \\ \qquad\qquad\qquad\qquad \times [\pi_2\beta/\alpha_2]\tau_2\end{array}}$$

$$\frac{\Gamma \vdash e : \exists\alpha{:}k.\tau \rightsquigarrow e_1 \quad \Gamma, \alpha{:}k, x{:}\tau \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e_2 \quad \Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash_{\mathsf{I}} \mathsf{unpack}\,[\alpha, x] = e \,\mathsf{in}\, (M : \sigma) : \sigma \rightsquigarrow \mathsf{unpack}\,[\alpha, x] = e_1 \,\mathsf{in}\, e_2}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e_1 \quad \Gamma, x{:}\tau \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e_2}{\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, M : \sigma \rightsquigarrow \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1 \rightsquigarrow e_1 \quad \Gamma, \alpha/m{:}\sigma_2 \vdash M_2 : \sigma_2 \rightsquigarrow e_2 \quad \Gamma \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M_1 \,\mathsf{in}\, (M_2 : \sigma_2) : \sigma_2 \rightsquigarrow \mathsf{unpack}\,[\alpha, \hat{m}] = e_1 \,\mathsf{in}\, e_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e}{\Gamma \vdash_{\mathsf{I}} (M :> \sigma) : \sigma \rightsquigarrow e}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e] \quad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma_1 \rightsquigarrow e \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f \quad \sigma_2 \rightsquigarrow [\alpha{:}k_2.\,\tau_2]}{\Gamma \vdash_{\mathsf{I}} M : \sigma_2 \rightsquigarrow \mathsf{unpack}\,[\alpha, x] = e \,\mathsf{in}\, \mathsf{pack}\,[\alpha, f[\alpha]x]\,\mathsf{as}\,\exists\alpha{:}k_2.\tau_2}$$

Figure 7: The impure module translation

$$\begin{array}{lcll}\overline{\epsilon} & \stackrel{\text{def}}{=} & \epsilon & \\ \overline{\Gamma, \alpha{:}k} & \stackrel{\text{def}}{=} & \overline{\Gamma}, \alpha{:}k & \\ \overline{\Gamma, x{:}\tau} & \stackrel{\text{def}}{=} & \overline{\Gamma}, x{:}\tau & \\ \overline{\Gamma, \alpha/m{:}\sigma} & \stackrel{\text{def}}{=} & \overline{\Gamma}, \alpha{:}k, \hat{m}{:}\tau & \text{where } \sigma \rightsquigarrow [\alpha{:}k.\,\tau]\end{array}$$

Figure 8: The context translation

This simply applies the functor's static portion to the argument's, and applies the functor's dynamic portion to the argument's static and dynamic portion. The impure case does the same thing, but first unpacks the functor from an existential package, and afterwards repacks the result.

Observe that, unless one were to do something obviously foolish, both of these rules are completely determined by the signature translation. The same is true for the translation of generative functors:

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\,\tau_1] \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{I}} M : \sigma_2 \rightsquigarrow e}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\star, \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e]}$$

The static portion of a generative functor is trivial. The dynamic portion takes in the static and dynamic portions of the argument (again $\alpha{:}k_1$ and $\hat{m} : \tau_1$)—again following the calling convention—and then computes an existential package (e) containing the static and dynamic portions of the result.

The generative functor application rule:

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1 : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow e_1 \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma_1 \rightsquigarrow [c_2, e_2]}{\Gamma \vdash_{\mathsf{I}} M_1 M_2 : [c_2/\alpha]\sigma_2 \rightsquigarrow \mathsf{unpack}\,[\_, f] = e_1 \,\mathsf{in}\, f\,[c_2]\,e_2}$$

after extracting the functor's dynamic portion from a trivial existential package simply applies it to the argument's static and dynamic portion.

Again, these rules are determined by the signature translation. The same is true of the rules governing atomic modules and pairs. The translation of let binding is determined simply by the calling convention:

$$\frac{\Gamma \vdash M_1 : \sigma_1 \rightsquigarrow e_1 \quad \Gamma, \alpha/m{:}\sigma_2 \vdash M_2 : \sigma_2 \rightsquigarrow e_2 \quad \Gamma \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M_1 \,\mathsf{in}\, (M_2 : \sigma_2) : \sigma_2 \rightsquigarrow \mathsf{unpack}\,[\alpha, \hat{m}] = e_1 \,\mathsf{in}\, e_2}$$

Since M_1 is impure, its translation is an existential package that must be unpacked to follow the calling convention. Once this is done, the right-hand-side is just passed through. Since σ_2 does not mention α, the type of e_2 also will not mention α.

The translation rule corresponding to the "forgetting" rule takes the statically determined static and dynamic components and rolls them into an existential package:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e] \quad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau}$$

Interestingly, this leaves nothing for the sealing rule to do:

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e}{\Gamma \vdash_{\mathsf{I}} (M :> \sigma) : \sigma \rightsquigarrow e}$$

All the sealing rule does is require that M be translated impurely. This induces the forgetting rule to do the work of sealing M up as an existential package.

Retyping principles Two technically tricky points pertain to the module calculus's retyping principles. The first such principle is the subsumption rule:

$$\frac{\Gamma \vdash_{\kappa} M : \sigma_1 \quad \Gamma \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash_{\kappa} M : \sigma_2}$$

If $\sigma_1 \leq \sigma_2$ and $\sigma_i \rightsquigarrow [\alpha{:}k_i.\,\tau_i]$, we can show that $k_1 \leq k_2$. If we had subtyping (and the full $F_{\leq}$ subtyping rule [4]) we

could also show that $\tau_1 \leq \tau_2$. But we do not have subtyping, and it is not the case that $\tau_1 \equiv \tau_2$, since $k' \leq k$ does not imply $\forall\alpha{:}k.\tau \equiv \forall\alpha{:}k'.\tau$. Consequently, the subsumption rule must apply a $\tau_1 \to \tau_2$ coercion to the dynamic portion:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma_1 \rightsquigarrow [c, e] \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f}{\Gamma \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, f\,[c]\,e]}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma_1 \rightsquigarrow e \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f \quad \sigma_2 \rightsquigarrow [\alpha{:}k_2.\,\tau_2]}{\Gamma \vdash_{\mathsf{I}} M : \sigma_2 \rightsquigarrow \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,\mathsf{pack}\,[\alpha, f[\alpha]x]\,\mathsf{as}\,\exists\alpha{:}k_2.\tau_2}$$

The coercion is computed by a judgement $\Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f$. It is essentially a large eta-expansion; it unrolls a term representing the dynamic portion of a module belonging to σ_1, and then re-rolls it at the type corresponding to σ_2. Details appear in Appendix A. In a practical implementation, this coercion can very frequently be inlined. Moreover, for fragments without higher-order functors (such as Standard ML), the coercion is always the identity.

The second retyping principle is the calculus's extensional typing rules, such as this rule for module pairs:

$$\frac{\Gamma \vdash_{\mathsf{P}} \pi_1 M : \sigma_1 \quad \Gamma \vdash_{\mathsf{P}} \pi_2 M : \sigma_2}{\Gamma \vdash_{\mathsf{P}} M : \sigma_1 \times \sigma_2}$$

The extensional typing rules say that a module belongs to a signature if uses of M produce appropriate results. For instance, M belongs to $\sigma_1 \times \sigma_2$ if its left projection belongs to σ_1 and its right to σ_2. These rules are never needed when M is an introduction form, but they allow one to ascribe stronger signatures to paths (*i.e.,* series of eliminations starting from a variable) than the elimination rules permit [9]. For example, they allow one to give a path a translucent signature that refers to itself, a procedure whimsically called "selfification." Extensional typing is necessary for higher-order singleton kinds and signatures to work properly (recall parts 2 and 3 of Lemma 2.1).

However, the extensional typing rules cause a problem for the translation. Suppose M translated to $[c, e]$ before using extensionality. The translation rule corresponding to the above rule would produce $[\langle\pi_1 c, \pi_2 c\rangle, \langle\pi_1 e, \pi_2 e\rangle]$. A similar thing would happen with the extensional typing rule for functors.

Unfortunately, this eta-expansion is not only inefficient, it is incorrect. In an eager operational semantics such as ours, e and $\langle\pi_1 e, \pi_2 e\rangle$ have observably different behavior whenever e has effects.

The solution to this problem is to exploit the fact that the extensional typing rules are needed only for paths. Our translation omits the extensional typing rules, and replaces them with a single "selfifying" translation rule for variables:

$$\frac{(\alpha/m : \sigma) \in \Gamma}{\Gamma \vdash_{\mathsf{P}} m : \mathsf{S}(\alpha : \sigma) \rightsquigarrow [\alpha, \hat{m}]}$$

Note the typing rule for variables gives m the signature σ, but the translation here gives it $\mathsf{S}(\alpha : \sigma)$. Because of this discrepancy, the translation rules are not in perfect correspondence with the typing rules, but we can nevertheless show that all well-typed expressions translate and vice versa:

Theorem 3.1 *If* $\vdash \Gamma\,\mathsf{ok}$ *then:*

- $\Gamma \vdash \sigma_1 \leq \sigma_2$ *iff* $\Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f$ *(for some* f*).*
- $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *iff* $\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]$ *(for some* c*,* e*).*
- $\Gamma \vdash_{\mathsf{I}} M : \sigma$ *iff* $\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e$ *(for some* e*).*
- $\Gamma \vdash e : \tau$ *iff* $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}$ *(for some* $\bar{e}$*).*

We will need more machinery before we can show the translation's full correctness, but at this point we can show its static correctness:

Lemma 3.2 (Static correctness) *If* $\vdash \Gamma\,\mathsf{ok}$ *then:*

- *If* $\Gamma \vdash \sigma : \mathsf{sig}$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *then* $\overline{\Gamma} \vdash k : \mathsf{kind}$ *and* $\overline{\Gamma}, \alpha{:}k \vdash \tau : \mathsf{T}$.
- *If* $\Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f$ *and* $\sigma_i \rightsquigarrow [\alpha{:}k_i.\,\tau_i]$ *then* $\overline{\Gamma} \vdash f : \forall\alpha{:}k_1.\,\tau_1 \to \tau_2$.
- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *then* $\overline{\Gamma} \vdash c : k$ *and* $\overline{\Gamma} \vdash e : [c/\alpha]\tau$.
- *If* $\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *then* $\overline{\Gamma} \vdash e : \exists\alpha{:}k.\tau$.
- *If* $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}$ *then* $\overline{\Gamma} \vdash \bar{e} : \tau$.

4 Contextual Equivalence

We follow Crary [2] by defining contextual equivalence as the coarsest congruence that is consistent with execution,[7] and our definitions in this section are adapted from there. However, in contrast to Crary, our indexed relations include a relation on static values. This comes into play in the compatibility rule for polymorphic functions because of the value restriction. Fortunately, this affects the development almost not at all, and ultimately we will prove that the value relation can be ignored (Lemma 4.5).

Definition 4.1 An *indexed dynamic relation* R is a triple of relations: relations (R_t and R_v) on terms and static values indexed by a context and type; and a relation (R_m) on modules indexed by a context, signature, and purity class; such that if $\vdash \Gamma\,\mathsf{ok}$ then:

- if $\Gamma \vdash e\ R_t\ e' : \tau$ then $\Gamma \vdash e, e' : \tau$, and
  if $\Gamma \vdash e\ R_v\ e' : \tau$ then $\Gamma \vdash e, e' : \tau$ and e, e' are static values, and
- if $\Gamma \vdash_\kappa M\ R_m\ M' : \sigma$ then $\Gamma \vdash_\kappa M, M' : \sigma$, and
- if $\Gamma \vdash_{\mathsf{P}} M\ R_m\ M' : \sigma$ then $\Gamma \vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\sigma)$.

Definition 4.2 Suppose R is an indexed dynamic relation.

- R is *compatible* if it respects the rules in Appendix B. (Informally, R is compatible if R-related expressions can be built from R-related subexpressions.)
- R is *substitutive* if it respects substitution (of values where appropriate) and weakening for constructor, term, and module hypotheses. (For example, if $\Gamma, \alpha/m{:}\sigma, \Gamma' \vdash e\ R_t\ e' : \tau$ and $\Gamma \vdash_{\mathsf{P}} V : \sigma$ then $\Gamma, [c/\alpha]\Gamma' \vdash [\mathsf{Fst}(V), V/\alpha, m]e\ R_t\ [\mathsf{Fst}(V), V/\alpha, m]e' : [c/\alpha]\tau$.)

[7] A more common definition would say that two expressions are contextually equivalent if they produce the same behavior when used to fill a hole in a program. Such a definition would be equivalent to ours, but in the setting of the module calculus, it is too unwieldy to use [2].

- R is a *congruence* if it is compatible, substitutive, reflexive, transitive, and symmetric.
- R is *consistent* if it preserves module termination. (That is, if $\vdash_{\mathsf{I}} M \ R_m \ M' : \sigma$ and M halts then M' halts.)

Although we define consistency in terms of module termination, it is easy to show that a consistent, compatible relation also preserves term termination.

Definition 4.3 *Contextually equivalence* (written $\Gamma \vdash e \approx e' : \tau$ and $\Gamma \vdash e \approx_v e' : \tau$ and $\Gamma \vdash_\kappa M \approx M' : \sigma$) is defined as the union of all consistent congruences.[8]

4.1 Properties of Contextual Equivalence

In proving dynamic correctness of phase separation, we will use several properties of contextual equivalence. One-and-a-half of these can be proven directly (the first and the left-to-right direction of the second). But contextual equivalence, which effectively quantifies over all observing contexts that might use the comparees, is awkward to work with. For the rest of the properties, we require a tool to obtain leverage on contextual equivalence. The tool is logical equivalence, which defines the equivalence of comparees intrinsically, rather than extrinsically through their use in surrounding code.

Crary [2] defines logical equivalence for the module calculus and shows that it coincides with contextual equivalence. We use that development off-the-shelf, except that we need a notion of logical equivalence for values. We define it as simply the restriction of logical equivalence to static values.

The desired properties are all corollaries of the coincidence of logical and contextual equivalence. Conveniently, we can state them all without reference to logical equivalence, so we do so in the interest of brevity. Proofs are given in the companion Coq formalization.

First, contextual equivalence is adequate for the operational semantics:

Lemma 4.4 (Adequacy) *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e \mapsto e'$ then $\Gamma \vdash e \approx e' : \tau$, and similarly for modules.*

Proof Sketch

The compatible, reflexive, and transitive closure of evaluation is a consistent congruence, and therefore is contained in contextual equivalence.

Second, contextual equivalence for values is precisely the restriction of ordinary contextual equivalence to static values. Thus, henceforth we can ignore contextual equivalence for values.

Lemma 4.5 (Value equivalence) $\Gamma \vdash sv \approx_v sv' : \tau$ *if and only if* $\Gamma \vdash sv \approx sv' : \tau$.

Proof Sketch

Immediate from the definition of logical equivalence and its coincidence with contextual equivalence. Alternatively, for the left-to-right direction, observe that $sv \approx (\Lambda\_{:}1.sv)[\star] \approx (\Lambda\_{:}1.sv')[\star] \approx sv'$.

[8] In the Coq formalization, contextual equivalence is given a different definition (the symmetrization of contextual approximation) which is then proved equivalent to this one.

Third, we can show two expressions are contextually equivalent by considering their closed instances:

Lemma 4.6 (Closed instances)

- *If $\Gamma \vdash e, e' : \tau$ and for any substitution γ such that $\vdash \gamma : \Gamma$ we have $\vdash \gamma(e) \approx \gamma(e') : \gamma(\tau)$, then $\Gamma \vdash e \approx e' : \tau$.*
- *If $\Gamma \vdash_{\mathsf{I}} M, M' : \sigma$ and for any substitution γ such that $\vdash \gamma : \Gamma$ we have $\vdash_{\mathsf{I}} \gamma(M) \approx \gamma(M') : \gamma(\sigma)$, then $\Gamma \vdash_{\mathsf{I}} M \approx M' : \sigma$.*
- *If $\Gamma \vdash_{\mathsf{P}} M, M' : \sigma$ and $\Gamma \vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\sigma)$ and for any substitution γ such that $\vdash \gamma : \Gamma$ we have $\vdash_{\mathsf{P}} \gamma(M) \approx \gamma(M') : \gamma(\sigma)$, then $\Gamma \vdash_{\mathsf{I}} M \approx M' : \sigma$.*

Fourth, when two closed expressions have the same termination behavior, we can assume they terminate when showing they are contextually equivalent:

Lemma 4.7 (Conditional equivalence)

- *If $\vdash e, e' : \tau$ and $e\!\downarrow \Leftrightarrow e'\!\downarrow$, then if $e\!\downarrow$ implies $\vdash e \approx e' : \tau$, then $\vdash e \approx e' : \tau$.*
- *If $\vdash_{\mathsf{I}} M, M' : \tau$ and $M\!\downarrow \Leftrightarrow M'\!\downarrow$, then if $M\!\downarrow$ implies $\vdash_{\mathsf{I}} M \approx M' : \sigma$, then $\vdash_{\mathsf{I}} M \approx M' : \sigma$.*
- *If $\vdash_{\mathsf{P}} M, M' : \tau$ and $\vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\sigma)$ and $M\!\downarrow \Leftrightarrow M'\!\downarrow$, then if $M\!\downarrow$ implies $\vdash_{\mathsf{P}} M \approx M' : \sigma$, then $\vdash_{\mathsf{P}} M \approx M' : \sigma$.*

Fifth, for closed terms and closed, pure modules, we have a variety of extensionality properties:

Lemma 4.8 (Extensionality) *Suppose $\vdash e, e' : \tau$ and $e\!\downarrow \Leftrightarrow e'\!\downarrow$, then $\vdash e \approx e' : \tau$ provided:*

- *$\tau = \mathsf{unit}$, or*
- *$\tau = \tau_1 \to \tau_2$ and for every $\vdash v : \tau_1$ we have $\vdash e\,v \approx e'\,v : \tau_2$, or*
- *$\tau = \forall \alpha{:}k.\tau'$ and for every $\vdash c : k$ we have $\vdash e[c] \approx e'[c] : [c/\alpha]\tau'$, or*
- *$\tau = \tau_1 \times \tau_2$ and $\vdash \pi_1 e \approx \pi_1 e' : \tau_1$ and $\vdash \pi_2 e \approx \pi_2 e' : \tau_2$.*

Suppose $\vdash_{\mathsf{P}} M, M' : \sigma$ and $M\!\downarrow \Leftrightarrow M'\!\downarrow$, then $\vdash_{\mathsf{P}} M \approx M' : \sigma$ provided:

- *$\sigma = 1$, or*
- *$\sigma = (\!|\, k \,|\!)$ and $\vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : k$, or*
- *$\sigma = \langle\!|\, \tau \,|\!\rangle$ and $\vdash \mathsf{Ext}\, M \approx \mathsf{Ext}\, M' : \tau$, or*
- *$\sigma = \Pi^{\mathsf{gn}} \alpha{:}\sigma_1.\sigma_2$ and for every $\vdash_{\mathsf{P}} V : \sigma_1$ we have $\vdash_{\mathsf{I}} MV \approx M'V : [\mathsf{Fst}(V)/\alpha]\sigma_2$, or*
- *$\sigma = \Pi^{\mathsf{ap}} \alpha{:}\sigma_1.\sigma_2$ and $\vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\Pi^{\mathsf{ap}} \alpha{:}\sigma_1.\sigma_2)$ and for every $\vdash_{\mathsf{P}} V : \sigma_1$ we have $\vdash_{\mathsf{P}} M \cdot V \approx M' \cdot V : [\mathsf{Fst}(V)/\alpha]\sigma_2$, or*
- *$\sigma = \Sigma \alpha{:}\sigma_1.\sigma_2$ and $\vdash_{\mathsf{P}} \pi_1 M \approx \pi_1 M' : \sigma_1$ and $\vdash_{\mathsf{P}} \pi_2 M \approx \pi_2 M' : [\pi_1\, \mathsf{Fst}(M)/\alpha]\sigma_2$.*

$$
\begin{array}{lcl}
\mathsf{Snd}_1 M & \stackrel{\text{def}}{=} & \mathsf{let}\ \_ = M\ \mathsf{in}\ \star \\
\mathsf{Snd}_{\langle\!| k |\!\rangle} M & \stackrel{\text{def}}{=} & \mathsf{let}\ \_ = M\ \mathsf{in}\ \star \\
\mathsf{Snd}_{\langle\!| \tau |\!\rangle} M & \stackrel{\text{def}}{=} & \mathsf{Ext}\, M \\
\mathsf{Snd}_{\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2} M & \stackrel{\text{def}}{=} & \mathsf{let}\, m = M \\
 & & \mathsf{in}\ \Lambda\alpha{:}k_1.\lambda x{:}\tau_1. \\
 & & \quad \mathsf{let}\, \beta/n = m(\mathsf{Join}_{\sigma_1}[\alpha, x]) \\
 & & \quad \mathsf{in\ pack}\, [\beta, \mathsf{Snd}_{\sigma_2} n]\ \mathsf{as}\ \exists\beta{:}k_2.\tau_2 \\
 & & (\text{where } \sigma_1 \rightsquigarrow [\alpha{:}k_1.\, \tau_1] \\
 & & \ \text{and } \sigma_2 \rightsquigarrow [\beta{:}k_2.\, \tau_2]) \\
\mathsf{Snd}_{\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2} M & \stackrel{\text{def}}{=} & \mathsf{let}\, m = M \\
 & & \mathsf{in}\ \Lambda\alpha{:}k_1.\lambda x{:}\tau_1. \\
 & & \quad \mathsf{Snd}_{\sigma_2}(m \cdot (\mathsf{Join}_{\sigma_1}[\alpha, x])) \\
 & & (\text{where } \sigma_1 \rightsquigarrow [\alpha{:}k_1.\, \tau_1]) \\
\mathsf{Snd}_{\Sigma\alpha:\sigma_1.\sigma_2} M & \stackrel{\text{def}}{=} & \mathsf{let}\, \gamma/m = M \\
 & & \mathsf{in}\ \langle \mathsf{Snd}_{\sigma_1}(\pi_1 m), \mathsf{Snd}_{[\pi_1\gamma/\alpha]\sigma_2}(\pi_2 m)\rangle \\
 & & \\
\mathsf{Join}_1[c, e] & \stackrel{\text{def}}{=} & \mathsf{let}\ \_ = e\ \mathsf{in}\ \star \\
\mathsf{Join}_{\langle\!| k |\!\rangle}[c, e] & \stackrel{\text{def}}{=} & \mathsf{let}\ \_ = e\ \mathsf{in}\ \langle\!| c |\!\rangle \\
\mathsf{Join}_{\langle\!| \tau |\!\rangle}[c, e] & \stackrel{\text{def}}{=} & \langle\!| e |\!\rangle \\
\mathsf{Join}_{\Pi^{\mathsf{gn}}\alpha:\sigma_1.\sigma_2}[c, e] & \stackrel{\text{def}}{=} & \mathsf{let}\, x = e \\
 & & \mathsf{in}\ \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1. \\
 & & \quad \mathsf{unpack}\, [\beta, y] = x[\alpha](\mathsf{Snd}_{\sigma_1} m) \\
 & & \quad \mathsf{in}\, (\mathsf{Join}_{\sigma_2}[\beta, y] : \sigma_2) \\
\mathsf{Join}_{\Pi^{\mathsf{ap}}\alpha:\sigma_1.\sigma_2}[c, e] & \stackrel{\text{def}}{=} & \mathsf{let}\, x = e \\
 & & \mathsf{in}\ \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & & \quad \mathsf{Join}_{\sigma_2}[c\,\alpha, x[\alpha](\mathsf{Snd}_{\sigma_1} m)] \\
\mathsf{Join}_{\Sigma\alpha:\sigma_1.\sigma_2}[c, e] & \stackrel{\text{def}}{=} & \mathsf{let}\, x = e \\
 & & \mathsf{in}\ \langle \mathsf{Join}_{\sigma_1}[\pi_1 c, \pi_1 x], \\
 & & \quad \mathsf{Join}_{[\pi_1 c/\alpha]\sigma_2}[\pi_2 c, \pi_2 x]\rangle
\end{array}
$$

Figure 9: Dynamic Phase Separation

5 Correctness

Our main aim is to show that every module is contextually equivalent to its translation, in an appropriate sense. But what is the appropriate sense? Since a module and its translation belong to different types, and indeed different syntactic classes, we require some way to relate them.

We do this using Snd_σ and Join_σ, two (families of) functions that convert back-and-forth between modules and terms. Given a module, Snd computes its dynamic component, and given a module's static and dynamic component, Join recovers the original module.

The important fact about Snd_σ and Join_σ is that they are expressed *within the language.* They do not traverse the syntax of their arguments the way the translation does. Rather, they operate through the interface given by their arguments' signature/kind/type. We refer to Snd as *dynamic phase separation* since it does phase separation at run time instead of compile time.

The definitions of Snd and Join are given in Figure 9. The fact that they are definable within the language is interesting in its own right. It shows that modules are a definitional extension of the core language. In other words, modules confer no fundamental expressive power absent in the core. On the other hand, the fact that the definitions are rather baroque shows that modules are nevertheless a useful feature for a language to support.

Lemma 5.1 *Suppose* $\vdash \Gamma\ \mathsf{ok}$ *and* $\Gamma \vdash \sigma : \mathsf{sig}$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\, \tau]$. *Then:*

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *then* $\Gamma \vdash \mathsf{Snd}_\sigma M : [\mathsf{Fst}(M)/\alpha]\tau$.
- *If* $\Gamma \vdash c : k$ *and* $\Gamma \vdash e : [c/\alpha]\tau$ *then* $\Gamma \vdash_{\mathsf{P}} \mathsf{Join}_\sigma[c, e] : \sigma$ *and* $\Gamma \vdash c \equiv \mathsf{Fst}(\mathsf{Join}_\sigma[c, e]) : k$.

The main lemmas state that Snd and Join are inverses:

Lemma 5.2 (Join of Snd) *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *then* $\Gamma \vdash_{\mathsf{P}} M \approx \mathsf{Join}_\sigma[\mathsf{Fst}(M), \mathsf{Snd}_\sigma(M)] : \sigma$.

Proof Sketch

The proof is by induction on the gross structure of σ (that is, considering all atoms ($\langle\!| k |\!\rangle$ or $\langle\!| \tau |\!\rangle$) to be identical for the purposes of the induction order).

We give one case, by way of example. Suppose $\sigma = \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2$. By the closed instances lemma, we may assume without loss of generality that Γ is empty. (Note that σ's instances have the same gross structure as σ.) It is not hard to show that $M{\downarrow} \Leftrightarrow \mathsf{Join}_\sigma[\mathsf{Fst}(M), \mathsf{Snd}_\sigma(M)]{\downarrow}$. Therefore, by the conditional equivalence lemma, we may assume without loss of generality that M is some value V. Let $\sigma_1 \rightsquigarrow [\alpha{:}k_1.\, \tau_1]$. Then:

$$
\begin{array}{ll}
 & \mathsf{Join}_\sigma[\mathsf{Fst}(V), \mathsf{Snd}_\sigma(V)] \\
= & \mathsf{let}\, f = \mathsf{let}\, m = V \\
 & \qquad \mathsf{in}\ \Lambda\alpha{:}k_1.\lambda x{:}\tau_1. \\
 & \qquad\quad \mathsf{Snd}_{\sigma_2}(m \cdot \mathsf{Join}_{\sigma_1}[\alpha, x]) \\
 & \ \mathsf{in}\ \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & \quad \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V)\,\alpha, f[\alpha](\mathsf{Snd}_{\sigma_1} m)] \\
\approx & \mathsf{let}\, f = \Lambda\alpha{:}k_1.\lambda x{:}\tau_1. \\
 & \qquad \mathsf{Snd}_{\sigma_2}(V \cdot \mathsf{Join}_{\sigma_1}[\alpha, x]) \\
 & \ \mathsf{in}\ \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & \quad \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V)\,\alpha, f[\alpha](\mathsf{Snd}_{\sigma_1} m)] \\
\approx & \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & \quad \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V)\,\alpha, \\
 & \qquad (\Lambda\alpha{:}k_1.\lambda x{:}\tau_1. \\
 & \qquad\quad \mathsf{Snd}_{\sigma_2}(V \cdot \mathsf{Join}_{\sigma_1}[\alpha, x])) \\
 & \qquad [\alpha]\,(\mathsf{Snd}_{\sigma_1} m)] \\
\approx & \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & \quad \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V)\,\alpha, \\
 & \qquad \mathsf{Snd}_{\sigma_2}(V \cdot \mathsf{Join}_{\sigma_1}[\alpha, \mathsf{Snd}_{\sigma_1} m])) \\
= & \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1. \\
 & \quad \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V \cdot m), \\
 & \qquad \mathsf{Snd}_{\sigma_2}(V \cdot \mathsf{Join}_{\sigma_1}[\mathsf{Fst}(m), \mathsf{Snd}_{\sigma_1} m])) \\
\approx & \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.\, \mathsf{Join}_{\sigma_2}[\mathsf{Fst}(V \cdot m), \mathsf{Snd}_{\sigma_2}(V \cdot m)] \\
\approx & \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.\, V \cdot m \\
\approx & V
\end{array}
$$

The last step is by eta equivalence (which follows from extensionality, since V halts), and the previous two steps use the induction hypothesis.

Lemma 5.3 (Snd of Join) *If* $\Gamma \vdash \sigma : \mathsf{sig}$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\, \tau]$ *and* $\Gamma \vdash c : k$ *and* $\Gamma \vdash e : [c/\alpha]\tau$ *then* $\Gamma \vdash e \approx \mathsf{Snd}_\sigma(\mathsf{Join}_\sigma[c, e]) : [c/\alpha]\tau$.

Proof Sketch

By induction on the gross structure of σ. (But see Section 5.3.)

Next, we establish the correctness of the subsignature coercion judgement. It is equivalent to the function that converts a term to a module at the subsignature, and back to a term at the supersignature:

Lemma 5.4 *If* $\Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f$ *and* $\sigma_i \rightsquigarrow [\alpha{:}k_i.\,\tau_i]$ *then* $\Gamma \vdash f \approx \Lambda\alpha{:}k_1.\lambda x{:}\tau_1.\mathsf{Snd}_{\sigma_2}(\mathsf{Join}_{\sigma_1}[\alpha, x]) : \forall\alpha{:}k_1.\,\tau_1 \to \tau_2$.

Proof Sketch

By induction on the derivation.

Recall that the variable translation case selfified its signature. To deal with this, we require one more lemma:

Lemma 5.5 *Suppose* $\Gamma \vdash \sigma : \mathsf{sig}$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *and* $\Gamma \vdash c : k$ *and* $\Gamma \vdash e : [c/\alpha]\tau$. *Then* $\Gamma \vdash_{\mathsf{P}} \mathsf{Join}_\sigma[c, e] \approx \mathsf{Join}_{\mathsf{S}(c:\sigma)}[c, e] : \sigma$.

Proof Sketch

We can show that if $\Gamma \vdash_{\mathsf{P}} M, M' : \sigma$ and $\Gamma \vdash \mathsf{Fst}(M) \equiv \mathsf{Fst}(M') : \mathsf{Fst}(\sigma)$ and if M and M' differ only in type annotations, then $\Gamma \vdash_{\mathsf{P}} M \approx M' : \sigma$.

Let $\mathsf{S}(c : \sigma) \rightsquigarrow [\alpha{:}k'.\,\tau']$. Then $k' = \mathsf{S}(c : k)$ and we can show that $\Gamma, \alpha{:}k' \vdash \tau \equiv \tau' : \mathsf{T}$, so $\Gamma \vdash e : [c/\alpha]\tau'$. The first two conditions hold using Lemma 5.1 and $\mathsf{S}(c : \sigma) \leq \sigma$. The third is by inspection.

To mediate the difference between the original context Γ and its translation $\overline{\Gamma}$, we define the substitution Snd_Γ to map $\hat{m}$ to $\mathsf{Snd}_\sigma m$ for every $\alpha/m{:}\sigma \in \Gamma$. Note that Snd_σ does not touch constructor variables, so we need not apply it to any kinds, constructors, or signatures.

Lemma 5.6 *If* $\vdash \Gamma\ \mathsf{ok}$ *then* $\Gamma \vdash \mathsf{Snd}_\Gamma : \overline{\Gamma}$.

With this, we can finally establish our main theorem:

Theorem 5.7 (Dynamic correctness)

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]$ *then* $\Gamma \vdash M \approx \mathsf{Join}_\sigma[c, \mathsf{Snd}_\Gamma(e)] : \sigma$.
- *If* $\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e$ *then* $\Gamma \vdash_{\mathsf{I}} M \approx \mathsf{unpack}\,[\alpha, x] = \mathsf{Snd}_\Gamma(e)\ \mathsf{in}\ (\mathsf{Join}_\sigma[\alpha, x] : \sigma) : \sigma$.
- *If* $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}$ *then* $\Gamma \vdash e \approx \mathsf{Snd}_\Gamma(\bar{e}) : \tau$.

Proof Sketch

The proof is by induction on the translation derivation. We give a few cases by way of example.

Case:

$$\frac{(\alpha/m : \sigma) \in \Gamma}{\Gamma \vdash_{\mathsf{P}} m : \mathsf{S}(\alpha : \sigma) \rightsquigarrow [\alpha, \hat{m}]}$$

Then $\mathsf{Join}_{\mathsf{S}(\alpha:\sigma)}[\alpha, \mathsf{Snd}_\Gamma(\hat{m})] = \mathsf{Join}_{\mathsf{S}(\alpha:\sigma)}[\mathsf{Fst}(m), \mathsf{Snd}_\sigma(m)] \approx \mathsf{Join}_\sigma[\mathsf{Fst}(m), \mathsf{Snd}_\sigma(m)] \approx m$, the second and third steps using Lemmas 5.5 and 5.2.

Case:

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, e] \quad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\tau_1]}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\lambda\alpha{:}k_1.c, \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e]}$$

Then:

$$\begin{aligned}
&\mathsf{Join}_{\Pi^{\mathsf{ap}}\alpha:\sigma_1.\sigma_2}[\lambda\alpha{:}k_1.c, \mathsf{Snd}_\Gamma(\Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.e)] \\
&= \mathsf{Join}_{\Pi^{\mathsf{ap}}\alpha:\sigma_1.\sigma_2}[\lambda\alpha{:}k_1.c, \Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.\mathsf{Snd}_\Gamma(e)] \\
&\approx \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.\mathsf{Join}_{\sigma_2}[(\lambda\alpha{:}k_1.c)\alpha, \\
&\qquad\qquad (\Lambda\alpha{:}k_1.\lambda\hat{m}{:}\tau_1.\mathsf{Snd}_\Gamma(e))[\alpha](\mathsf{Snd}_{\sigma_1}m)] \\
&\approx \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.\mathsf{Join}_{\sigma_2}[c, [\mathsf{Snd}_{\sigma_1}m/\hat{m}]\mathsf{Snd}_\Gamma(e)] \\
&= \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.\mathsf{Join}_{\sigma_2}[c, \mathsf{Snd}_{\Gamma,\alpha/m:\sigma_1}(e)] \\
&\approx \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M
\end{aligned}$$

Case:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e] \quad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow \mathsf{pack}\,[c, e]\ \mathsf{as}\ \exists\alpha{:}k.\tau}$$

Then $(\mathsf{unpack}\,[\beta, x] = \mathsf{Snd}_\Gamma(\mathsf{pack}\,[c, e]\ \mathsf{as}\ \exists\alpha{:}k.\tau)\ \mathsf{in}\ \mathsf{Join}_\sigma[\beta, x]) = (\mathsf{unpack}\,[\beta, x] = \mathsf{pack}\,[c, \mathsf{Snd}_\Gamma(e)]\ \mathsf{as}\ \exists\alpha{:}k.\tau\ \mathsf{in}\ \mathsf{Join}_\sigma[\beta, x]) \approx \mathsf{Join}_\sigma[c, \mathsf{Snd}_\Gamma(e)] \approx M$.

Case:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma_1 \rightsquigarrow [c, e] \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow f}{\Gamma \vdash_{\mathsf{P}} M : \sigma_2 \rightsquigarrow [c, f\,[c]\,e]}$$

Then $\mathsf{Join}_{\sigma_2}[c, \mathsf{Snd}_\Gamma(f\,[c]\,e)] = \mathsf{Join}_{\sigma_2}[c, f\,[c]\,(\mathsf{Snd}_\Gamma(e))] \approx \mathsf{Join}_{\sigma_2}[c, \mathsf{Snd}_{\sigma_2}(\mathsf{Join}_{\sigma_1}[c, \mathsf{Snd}_\Gamma(e)])] \approx \mathsf{Join}_{\sigma_1}[c, \mathsf{Snd}_\Gamma(e)] \approx M$. The first step is because f is closed with respect to term variables; the second and third are by Lemmas 5.4 and 5.2.

Observe that for closed terms, the theorem degenerates to if $\vdash e : \tau \rightsquigarrow \bar{e}$ then $\vdash e \approx \bar{e} : \tau$.

5.1 Abstraction Preservation

It remains to collect some corollaries. We can prove a converse of dynamic correctness:

Corollary 5.8 *Let* Join_σ *be the substitution that maps* m *to* $\mathsf{Join}_\sigma[\alpha, \hat{m}]$ *for every* $\alpha/m{:}\sigma \in \Gamma$. *(So* $\overline{\Gamma} \vdash \mathsf{Join}_\Gamma : \Gamma$ *whenever* $\vdash \Gamma\ \mathsf{ok}$*.) Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c, e]$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *then* $\overline{\Gamma} \vdash e \approx \mathsf{Snd}_\sigma(\mathsf{Join}_\Gamma(M)) : [c/\alpha]\tau$.
- *If* $\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *then* $\overline{\Gamma} \vdash e \approx \mathsf{let}\,\alpha/m = \mathsf{Join}_\Gamma(M)\ \mathsf{in}\ \mathsf{pack}\,[\alpha, \mathsf{Snd}_\sigma m]\ \mathsf{as}\ \exists\alpha{:}k.\tau : \exists\alpha{:}k.\tau$.
- *If* $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}$ *then* $\overline{\Gamma} \vdash \bar{e} \approx \mathsf{Join}_\Gamma(e) : \tau$.

Proof Sketch

Consider the first clause. Using compatibility, we can show that contextual equivalence respects functionality, so $\mathsf{Join}_\Gamma(\mathsf{Snd}_\Gamma(e)) \approx e$ by Lemma 5.3. By dynamic correctness, $M \approx \mathsf{Join}_\sigma[c, \mathsf{Snd}_\Gamma(e)]$. By substitutivity, $\mathsf{Join}_\Gamma(M) \approx \mathsf{Join}_\Gamma(\mathsf{Join}_\sigma[c, \mathsf{Snd}_\Gamma(e)]) = \mathsf{Join}_\sigma[c, \mathsf{Join}_\Gamma(\mathsf{Snd}_\Gamma(e))] \approx \mathsf{Join}_\sigma[c, e]$. Then, again by Lemma 5.3, $\mathsf{Snd}_\sigma(\mathsf{Join}_\Gamma(M)) \approx \mathsf{Snd}_\sigma(\mathsf{Join}_\sigma[c, e]) \approx e$.

The second clause is similar but messier. The third clause is similar but simpler.

Abstraction preservation follows directly from this. If one neglects (for the moment) the difference between the source and target language, this is the hard direction of full abstraction:

Corollary 5.9 (Abstraction preservation) *Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

- *If* $\Gamma \vdash_{\mathsf{P}} M_1 \approx M_2 : \sigma$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *and* $\Gamma \vdash_{\mathsf{P}} M_i : \sigma \rightsquigarrow [c_i, e_i]$ *then* $\overline{\Gamma} \vdash e_1 \approx e_2 : [c_1/\alpha]\tau$.
- *If* $\Gamma \vdash_{\mathsf{I}} M_1 \approx M_2 : \sigma$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ *and* $\Gamma \vdash_{\mathsf{I}} M_i : \sigma \rightsquigarrow e_i$ *then* $\overline{\Gamma} \vdash e_1 \approx e_2 : \exists\alpha{:}k.\tau$.
- *If* $\Gamma \vdash e_1 \approx e_2 : \tau$ *and* $\Gamma \vdash e_i : \tau \rightsquigarrow \bar{e}_i$ *then* $\overline{\Gamma} \vdash \bar{e}_1 \approx \bar{e}_2 : \tau$.

Proof Sketch

We give the first clause; the others are similar. Suppose $\Gamma \vdash_{\mathsf{P}} M_1 \approx M_2 : \sigma$. By substitutivity of contextual equivalence, $\overline{\Gamma} \vdash \mathsf{Join}_\Gamma(M_1) \approx \mathsf{Join}_\Gamma(M_2) : \sigma$. Therefore $e_1 \approx \mathsf{Snd}_\sigma(\mathsf{Join}_\Gamma(M_1)) \approx \mathsf{Snd}_\sigma(\mathsf{Join}_\Gamma(M_2)) \approx e_2$ by Corollary 5.8 and compatibility.

The coherence of the translation is an immediate consequence of this, using reflexivity for the antecedent:

Corollary 5.10 (Coherence) *Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\tau]$ *and* $\Gamma \vdash_{\mathsf{P}} M : \sigma \rightsquigarrow [c_i, e_i]$ *(for* $i = 1, 2$*) then* $\overline{\Gamma} \vdash e_1 \approx e_2 : [c_1/\alpha]\tau$.
- *If* $\Gamma \vdash_{\mathsf{I}} M : \sigma$ *and* $\sigma \rightsquigarrow [\alpha{:}k.\tau]$ *and* $\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e_i$ *(for* $i = 1, 2$*) then* $\overline{\Gamma} \vdash e_1 \approx e_2 : \exists\alpha{:}k.\tau$.
- *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau \rightsquigarrow \bar{e}_i$ *(for* $i = 1, 2$*) then* $\overline{\Gamma} \vdash \bar{e}_1 \approx \bar{e}_2 : \tau$.

5.2 Full Abstraction

Full abstraction states that expressions are source-equivalent (*i.e.,* equivalent in the source language) exactly when their translations are target-equivalent. Abstraction preservation gives us most of this: It says that that translations of source-equivalent expressions are source-equivalent. Since the target language is a subset of the source, every observing context in the target is also one in the source, and consequently source-equivalence implies target-equivalence. Thus, abstraction preservation gives us half of full abstraction.

It remains to show that expressions are source-equivalent when their translations are target-equivalent. Dynamic correctness says that source expressions are source-equivalent to their translations, so the result follows by transitivity, provided target-equivalent expressions are source-equivalent:

Lemma 5.11 *Let* $\Gamma \vdash_{tgt} e \approx e' : \tau$ *be contextual equivalence for the target, defined analogously to Section 4, but using only the features of the target language. Then* $\Gamma \vdash_{tgt} e \approx e' : \tau$ *implies* $\Gamma \vdash e \approx e' : \tau$.

Proof Sketch

As developed in Crary [2], contextual equivalence coincides with logical equivalence, which is biorthogonally closed [26]. Dynamic correctness tells us that every closed source term has an equivalent target term. Thus, in biorthogonal closure we may assume without loss of generality that the continuations are target terms. It follows that biorthogonally closed sets are closed under target-equivalence.

Corollary 5.12 (Full abstraction) *Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

- *If* $\sigma \rightsquigarrow [\alpha{:}k.\tau]$ *and* $\Gamma \vdash_{\mathsf{P}} M_i : \sigma \rightsquigarrow [c_i, e_i]$ *then* $\Gamma \vdash_{\mathsf{P}} M_1 \approx M_2 : \sigma$ *exactly when both* $\overline{\Gamma} \vdash c_1 \equiv c_2 : k$ *and* $\overline{\Gamma} \vdash_{tgt} e_1 \approx e_2 : [c_1/\alpha]\tau$.
- *If* $\sigma \rightsquigarrow [\alpha{:}k.\tau]$ *and* $\Gamma \vdash_{\mathsf{I}} M_i : \sigma \rightsquigarrow e_i$ *then* $\Gamma \vdash_{\mathsf{I}} M_1 \approx M_2 : \sigma$ *exactly when* $\overline{\Gamma} \vdash_{tgt} e_1 \approx e_2 : \exists\alpha{:}k.\tau$.
- *If* $\Gamma \vdash e_i : \tau \rightsquigarrow e_i'$ *then* $\Gamma \vdash e_1 \approx e_2 : \tau$ *exactly when* $\overline{\Gamma} \vdash_{tgt} e_1' \approx e_2' : \tau$.

5.3 The Role of the Value Restriction

Observe that $\mathsf{Join} \circ \mathsf{Snd}$ and $\mathsf{Snd} \circ \mathsf{Join}$ have the effect of eta-expanding their arguments (after evaluating them to values). For example, suppose F is an applicative functor value. As we saw in the proof sketch of Lemma 5.2, $\mathsf{Join}[\mathsf{Fst}(F), \mathsf{Snd}(F)]$ is equivalent, after some simplification, to $\lambda m.\, F \cdot m$. The latter is equivalent to F because F is a value.

On the other hand, suppose f is a value, and suppose c and f have an appropriate kind/type to be the static and dynamic components of an applicative functor. Note that f's type must have the form $\forall\alpha{:}k.\,\tau{\to}\tau'$. Then $\mathsf{Snd}(\mathsf{Join}[c, f])$ is equivalent, after some simplification, to $\Lambda\alpha.\lambda x.f[\alpha]x$. This is equivalent to f, but only because of the value restriction. Since f is a value, it is certainly equivalent to $\Lambda\alpha.f[\alpha]$, but $f[\alpha]$ is *not* a value, so without a value restriction, it would not in general be equivalent to $\lambda x.f[\alpha]x$. The value restriction ensures that $f[\alpha]$ will halt, which is good enough to conclude $f[\alpha] \approx \lambda x.f[\alpha]x$. Thus Lemma 5.3 relies crucially on the value restriction.

We can work backward from this observation to obtain a counterexample of full abstraction if the value restriction were omitted. Suppose m is a variable with an applicative-functor signature. Since m is a value, $m \approx \lambda^{\mathsf{ap}}\alpha/n.\, m{\cdot}n$. But m translates to dynamic component $\hat{m}$, whereas $\lambda^{\mathsf{ap}}\alpha/n.\, m \cdot n$ translates to dynamic component $\Lambda\alpha.\lambda\hat{n}.\,\hat{m}[\alpha]\hat{n}$. These translations can be distinguished by the observing context $\mathsf{let}\,\hat{m} = \Lambda\alpha.\bot\ \mathsf{in}\ \mathsf{let}\ \_ = [\,]\,c\ \mathsf{in}\ \star$ (where c is any constructor of appropriate kind).

This issue with full abstraction does not rely on applicative functors. A similar example can be constructed using a generative functor.

(Both Shan [31] and Rossberg, *et al.* [27] produce a generally similar translation to ours, but neither of their target languages has a value restriction on polymorphic functions.[9] Consequently neither of them preserves abstraction, so long as their target language is capable of expressing the eager let used in the observing context above.)

Note that this issue does not rule out having a second form of polymorphic function without a value restriction, were one desired. The important thing is that the target language have a form of function able to take in both a constructor and a value at once, without any intervening effects.

One could even have a value restriction in the target and not the source, although having different source and target languages would complicate the development. We also conjecture that the translation would be fully abstract even without a value restriction if it were to translate variables to their full eta-long form. But such a translation is undesirable for efficiency reasons, so we have not explored it carefully.

6 Definitional Extensions

The alignment of abstraction with effects in our module calculus suggests a natural treatment of first-class modules [29, 5] as a definitional extension. Suppose we have a new term form $\mathsf{md}_\sigma(M)$, which stands for an encapsulated module and has type $\mathsf{md}(\sigma)$. And suppose such an

[9] Note that it is the *target* language at issue here, not any value restriction that might exist in the source language. Both Shan and Rossberg *et al.* target ordinary F_ω (extended with extra types), which has no value restriction.

encapsulated module can be made back into a module using $\mathsf{tm}_\sigma(e)$.

Although this formulation offers less expressive power than systems that allow the propagation of type information through first-class code [9], it supports most important applications of first-class modules such as run-time selection of implementations of abstract data types.

The critical point in the static semantics of first-class modules is that the tm form must be impure. Since the module depends on arbitrary code, it is impossible to determine its type components statically:

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma}{\Gamma \vdash \mathsf{md}_\sigma(M) : \mathsf{md}(\sigma)} \qquad \frac{\Gamma \vdash e : \mathsf{md}(\sigma)}{\Gamma \vdash_{\mathsf{I}} \mathsf{tm}_\sigma(e) : \sigma}$$

Two questions present themselves regarding these first-class modules: Does the language still respect the phase distinction, and does the extension leave intact the language's existing abstraction properties? At first glance, neither seems certain. The phase distinction depends on the fact that types cannot depend on terms, but first-class modules appear, at least superficially, to provide a way for that to happen. And, first-class modules provide a wealth of new observing contexts that might be used to distinguish between previously indistinguishable modules.

But the machinery developed here allows us easily to answer both questions in the affirmative. We show that we can faithfully define first-class modules as a definitional extension in the original language. This shows that the extension respects the phase distinction, since the original did. It also shows that the extension preserves abstraction, because any observing context in the extended language can be turned into an equivalent observing context in the original language. Consequently, any modules that are indistinguishable in the original language remain so with the extension.

Suppose $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$. We define:

$$\begin{array}{lcl} \mathsf{md}(\sigma) & \stackrel{\text{def}}{=} & \exists \alpha{:}k.\tau \\ \mathsf{md}_\sigma(M) & \stackrel{\text{def}}{=} & \mathsf{let}\,\alpha/m = M\,\mathsf{in}\,\mathsf{pack}\,[\alpha, \mathsf{Snd}_\sigma m]\,\mathsf{as}\,\exists \alpha{:}k.\tau \\ \mathsf{tm}_\sigma(e) & \stackrel{\text{def}}{=} & \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(\mathsf{Join}_\sigma[\alpha, x] : \sigma) \end{array}$$

We can easily show that $\mathsf{tm}_\sigma(\mathsf{md}_\sigma(N)) \approx N$, thereby simulating the extension's operational semantics.

Another useful extension is an operation $\mathsf{purify}_{\mathsf{S}(c:\sigma)}(M)$ that takes an impure but transparent module and makes it pure [33, 5]. This expresses the fact that in a module whose exported type components are fully specified, any typing effects are benign. This is useful for using generative functors within the implementation of an applicative functor.

$$\frac{\Gamma \vdash c : \mathsf{Fst}(\sigma) \quad \Gamma \vdash_{\mathsf{I}} M : \mathsf{S}(c : \sigma)}{\Gamma \vdash_{\mathsf{P}} \mathsf{purify}_{\mathsf{S}(c:\sigma)}(M) : \mathsf{S}(c : \sigma)}$$

We show that purify is a definitional extension by defining $\mathsf{purify}_{\mathsf{S}(c:\sigma)}(M)$ as:

$$\mathsf{Join}_{\mathsf{S}(c:\sigma)}[c, \mathsf{let}\, m = M\,\mathsf{in}\,\mathsf{Snd}_{\mathsf{S}(c:\sigma)} m]$$

This trick has two parts: First, by converting M to a term, we disappear any typing effects it might have. Then, since M is transparent, its static component (c) is known without referring to M, so we are able to convert back to a pure module using Join.

When M is a value (and hence pure), we can easily show that $\mathsf{purify}_{\mathsf{S}(c:\sigma)}(M) \approx M$, thereby simulating the extension's operational semantics.

Although these definitions are satisfactory from a theoretical standpoint, in practice it may be preferable to translate the extensions directly:

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e}{\Gamma \vdash \mathsf{md}_\sigma(M) : \mathsf{md}(\sigma) \rightsquigarrow e} \qquad \frac{\Gamma \vdash e : \mathsf{md}(\sigma) \rightsquigarrow \bar{e}}{\Gamma \vdash_{\mathsf{I}} \mathsf{tm}_\sigma(e) : \sigma \rightsquigarrow \bar{e}}$$

Observe that first-class modules are actually trivial when viewed through the lens of phase separation. On the other hand, the type translation is no longer the identity ($\mathsf{md}(\sigma)$ becomes $\exists \alpha{:}k.\tau$), which adds a bit of overhead to the compiler.

$$\frac{\Gamma \vdash c : \mathsf{Fst}(\sigma) \quad \Gamma \vdash_{\mathsf{I}} M : \mathsf{S}(c : \sigma) \rightsquigarrow e}{\Gamma \vdash_{\mathsf{P}} \mathsf{purify}_{\mathsf{S}(c:\sigma)}(M) : \mathsf{S}(c : \sigma) \rightsquigarrow [c, \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,x]}$$

This translation works because in the unpack, α has the kind $\mathsf{S}(c : \mathsf{Fst}(\sigma))$, so $\alpha \equiv c : \mathsf{Fst}(\sigma)$, so x's type can be expressed without reference to α.

7 Formalization

All the results in this paper are formalized in Coq (version 8.4). We implemented binding using deBruijn indices and explicit substitutions. To make reasoning about relations cleaner, we used the axioms of functional and propositional extensionality.

The full development is 89k lines. (These counts include comments and whitespace.) Of those, 70k are preliminaries or the development of logical equivalence, which are borrowed from Crary [2], with some modest editing, mostly to deal with the value restriction. The all-new material—the translation and its correctness proof—account for 19k lines.

We remarked above that the addition of value restriction had almost no effect on the development of logical equivalence. As a coarse measure of that, updating the logical-equivalence Coq proofs to account for the value restriction took less than one day.

The numerous typing premises involved in showing contextual equivalences rapidly become unmanageable. To manage this, we defined *reductive contextual equivalence*. We say that $\Gamma \vdash e \Rightarrow e'$ if for any τ, $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e \approx e' : \tau$ (and similarly for modules). One example of how this simplifies proofs is we can say that $\Gamma \vdash e \mapsto e'$ implies $\Gamma \vdash e \Rightarrow e'$, eliminating the typing premise from Lemma 4.4.

8 Additional Discussion of Related Work

As discussed above, the closest work to ours is Shan [31] and Rossberg, *et al.* [27]. The most significant difference from ours is they do not establish dynamic correctness, but there is a stylistic difference as well. Although they are not fundamentally different from the pattern of the Harper, *et al.* [12], their translations do not explicitly separate modules into type and term components. Instead, they realize the phase distinction by separating all the opaque type components into a prefix on which the module depends. (Shao [33] calls this prefix the module's *flexroot*.) In ordinary modules, the prefix is existentially quantified, and in functors it is lambda abstracted. In essence, the module is automatically

rewritten to replace sharing-by-fibration with sharing-by-parameterization [14]. An advantage of their presentation is it allows one to dispense with any machinery to control the propagation of type information (such as the singleton kinds we use here).

These two differences may be connected. Our proof technique relies crucially on separating modules into type constructors and terms. It is not clear how to define something like Snd and Join without doing so.

Shan also addresses weak sealing [6], which we do not. He does so by generating a two-stage program. The first stage resolves static effects (in the sense of Dreyer, *et al.* [6]) that arise from weak sealing, resulting in a program without weak sealing. The second stage then executes that program, resolving the remaining (dynamic) effects as it goes. Rossberg, *et al.* do something similar. This approach is very plausible, but we do not follow it here, because the equational theory of weak sealing is still not well understood. Specifically, we do not yet know how to reconcile logical equivalence with weak sealing, so we cannot even begin to prove full abstraction.

Elsman, *et al.* [7, 8] suggest a radically different approach called static interpretation. It resolves all module-level computation (*e.g.,* functor application) at compile time. This sacrifices separate compilation (functors cannot be compiled separately from their arguments), but it makes it easy to optimize across module boundaries. In essence, static interpretation weakens the phase distinction by assigning module-level computation to the compile-time phase rather than the run-time phase. Their languages do not include sealing, but do include generative datatypes, which offer a means of abstraction. As with other prior work, they do not show dynamic correctness.

9 Conclusion

Our correctness and full abstraction results show that the phase-separation algorithm is canonical, from a theoretical standpoint anyway. Any other algorithm that is dynamically correct (in our sense) is necessarily equivalent to ours.

Nevertheless, there is room for improvement in the language itself. Our module calculus supports only a fragment of the features needed for Standard ML [16]. Many of the missing features are easily supported, but some—particularly those involving references, exceptions, or other forms of effects—are more difficult. Another important feature for languages that emphasize applicative functors (such as OCaml [20]) is weak sealing [6]. Still another direction is more powerful module features such as recursive modules [3, 30, 5].

In each of these cases, we anticipate that the general proof strategy employed here should work. However, a precondition for progress in any of these domains is a working account of logical equivalence. These are important avenues for future work.

A The Term and Subsignature Translations

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow \bar{e}}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow x} \qquad \frac{}{\Gamma \vdash \star : \mathsf{unit} \rightsquigarrow \star}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2 \rightsquigarrow \bar{e}}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x{:}\tau_1.\bar{e}}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \rightsquigarrow \bar{e}_1 \quad \Gamma \vdash e_2 : \tau \rightsquigarrow \bar{e}_2}{\Gamma \vdash e_1 e_2 : \tau' \rightsquigarrow \bar{e}_1 \bar{e}_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \bar{e}_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \bar{e}_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \rightsquigarrow \langle \bar{e}_1, \bar{e}_2 \rangle}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \bar{e}}{\Gamma \vdash \pi_1 e : \tau_1 \rightsquigarrow \pi_1 \bar{e}} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \bar{e}}{\Gamma \vdash \pi_2 e : \tau_2 \rightsquigarrow \pi_2 \bar{e}}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash sv : \tau \rightsquigarrow \bar{e}}{\Gamma \vdash \Lambda\alpha{:}k.sv : \forall\alpha{:}k.\tau \rightsquigarrow \Lambda\alpha{:}k.\bar{e}}$$

$$\frac{\Gamma \vdash e : \forall\alpha{:}k.\tau \rightsquigarrow \bar{e} \quad \Gamma \vdash c : k}{\Gamma \vdash e[c] : [c/\alpha]\tau \rightsquigarrow \bar{e}\,[c]}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash e : [c/\alpha]\tau \rightsquigarrow \bar{e} \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau : \exists\alpha{:}k.\tau \rightsquigarrow \mathsf{pack}\,[c, \bar{e}]\,\mathsf{as}\,\exists\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e_1 : \exists\alpha{:}k.\tau \rightsquigarrow \bar{e}_1 \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2 : \tau' \rightsquigarrow \bar{e}_2 \quad \Gamma \vdash \tau' : \mathsf{T}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2 : \tau' \rightsquigarrow \mathsf{unpack}\,[\alpha, x] = \bar{e}_1\,\mathsf{in}\,\bar{e}_2}$$

$$\frac{\Gamma \vdash e : (\mathsf{unit} \to \tau) \to \tau \rightsquigarrow \bar{e}}{\Gamma \vdash \mathsf{fix}_\tau\, e : \tau \rightsquigarrow \mathsf{fix}_\tau\, \bar{e}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \bar{e}_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \bar{e}_2}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 : \tau_2 \rightsquigarrow \mathsf{let}\,x = \bar{e}_1\,\mathsf{in}\,\bar{e}_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma \rightsquigarrow e_1 \quad \Gamma, \alpha/m{:}\sigma \vdash e : \tau \rightsquigarrow e_2 \quad \Gamma \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e : \tau \rightsquigarrow \mathsf{unpack}\,[\alpha, \hat{m}] = e_1\,\mathsf{in}\,e_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \langle\!|\, \tau \,|\!\rangle \rightsquigarrow e}{\Gamma \vdash \mathsf{Ext}\,M : \tau \rightsquigarrow \mathsf{unpack}\,[\_, x] = e\,\mathsf{in}\,x}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow \bar{e} \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash e : \tau' \rightsquigarrow \bar{e}}$$

$\boxed{\Gamma \vdash \sigma \leq \sigma' \rightsquigarrow f}$

$$\frac{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig} \quad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash \sigma \leq \sigma' \rightsquigarrow \Lambda\alpha{:}k.\lambda x{:}\tau.x}$$

$$\frac{\Gamma \vdash \sigma \leq \sigma' \rightsquigarrow f \quad \Gamma \vdash \sigma' \leq \sigma'' \rightsquigarrow f' \quad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash \sigma \leq \sigma'' \rightsquigarrow \Lambda\alpha{:}k.\lambda x{:}\tau.\, f'\,[\alpha]\,(f\,[\alpha]\,x)}$$

$$\frac{\Gamma \vdash k \leq k'}{\Gamma \vdash \langle\!|\, k \,|\!\rangle \leq \langle\!|\, k \,|\!\rangle \rightsquigarrow \Lambda\alpha{:}k.\lambda x{:}\mathsf{unit}.x}$$

$$\frac{\Gamma \vdash \sigma_1' \le \sigma_1 \rightsquigarrow f_1 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \le \sigma_2' \rightsquigarrow f_2 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig} \qquad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\tau_1] \qquad \sigma_1' \rightsquigarrow [\alpha{:}k_1'.\tau_1'] \qquad \sigma_2 \rightsquigarrow [\beta{:}k_2.\tau_2] \qquad \sigma_2' \rightsquigarrow [\beta{:}k_2'.\tau_2']}{\begin{array}{l}\Gamma \vdash \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2 \le \Pi^{\mathsf{gn}}\alpha{:}\sigma_1'.\sigma_2' \\ \quad \rightsquigarrow \Lambda\_{:}1.\lambda f{:}(\forall\alpha{:}k_1.\tau_1 \to \exists\beta{:}k_2.\tau_2). \\ \quad\quad \Lambda\alpha{:}k_1'.\lambda x{:}\tau_1'. \\ \quad\quad\quad \mathsf{unpack}\,[\beta, y] = f\,[\alpha]\,(f_1\,[\alpha]\,x) \\ \quad\quad\quad \mathsf{in\ pack}\,[\beta, f_2\,[\beta]\,y]\,\mathsf{as}\,\exists\beta{:}k_2'.\tau_2'\end{array}}$$

$$\frac{\Gamma \vdash \sigma_1' \le \sigma_1 \rightsquigarrow f_1 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \le \sigma_2' \rightsquigarrow f_2 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig} \qquad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\tau_1] \qquad \sigma_1' \rightsquigarrow [\alpha{:}k_1'.\tau_1'] \qquad \sigma_2 \rightsquigarrow [\beta{:}k_2.\tau_2]}{\begin{array}{l}\Gamma \vdash \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2 \le \Pi^{\mathsf{ap}}\alpha{:}\sigma_1'.\sigma_2' \\ \quad \rightsquigarrow \Lambda\gamma{:}(\Pi\alpha{:}k_1.k_2).\lambda f{:}(\forall\alpha{:}k_1.\tau_1 \to [\gamma\alpha/\beta]\tau_2). \\ \quad\quad \Lambda\alpha{:}k_1'.\lambda x{:}\tau_1'. \\ \quad\quad\quad f_2\,[\gamma\alpha]\,(f\,[\alpha]\,(f_1\,[\alpha]x))\end{array}}$$

$$\frac{\Gamma \vdash \sigma_1 \le \sigma_1' \rightsquigarrow f_1 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \le \sigma_2' \rightsquigarrow f_2 \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2' : \mathsf{sig} \qquad \sigma_1 \rightsquigarrow [\alpha{:}k_1.\,\tau_1] \qquad \sigma_2 \rightsquigarrow [\beta{:}k_2.\,\tau_2]}{\begin{array}{l}\Gamma \vdash \Sigma\alpha{:}\sigma_1.\sigma_2 \le \Sigma\alpha{:}\sigma_1'.\sigma_2' \\ \quad \rightsquigarrow \Lambda\gamma{:}(\Sigma\alpha{:}k_1.k_2).\lambda x{:}([\pi_1\gamma/\alpha]\tau_1 \times [\pi_1\gamma, \pi_2\gamma/\alpha, \beta]\tau_2). \\ \quad\quad \langle f_1\,[\pi_1\gamma]\,(\pi_1 x), ([\pi_1\gamma/\alpha]f_2)\,[\pi_2\gamma]\,(\pi_2 x)\rangle\end{array}}$$

B Compatibility

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma, x{:}\tau_1 \vdash e\ R_t\ e' : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e\ R_t\ \lambda x{:}\tau_1.e' : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1\ R_t\ e_1' : \tau \to \tau' \quad \Gamma \vdash e_2\ R_t\ e_2' : \tau}{\Gamma \vdash e_1 e_2\ R_t\ e_1' e_2' : \tau'}$$

$$\frac{\Gamma \vdash e_1\ R_t\ e_1' : \tau_1 \quad \Gamma \vdash e_2\ R_t\ e_2' : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle\ R_t\ \langle e_1', e_2'\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e\ R_t\ e' : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e\ R_t\ \pi_1 e' : \tau_1} \qquad \frac{\Gamma \vdash e\ R_t\ e' : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 e\ R_t\ \pi_2 e' : \tau_2}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k; \Gamma \vdash e\ R_v\ e' : \tau}{\Gamma \vdash \Lambda\alpha{:}k.e\ R_t\ \Lambda\alpha{:}k.e' : \forall\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e\ R_t\ e' : \forall\alpha{:}k.\tau \quad \Gamma \vdash c : k}{\Gamma \vdash e[c]\ R_t\ e'[c] : [c/\alpha]\tau}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash e\ R_t\ e' : [c/\alpha]\tau \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau\ R_t\ \mathsf{pack}\,[c, e']\,\mathsf{as}\,\exists\alpha{:}k.\tau : \exists\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e_1\ R_t\ e_1' : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2\ R_t\ e_2' : \tau' \quad \Gamma \vdash \tau' : \mathsf{T}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2\ R_t\ \mathsf{unpack}\,[\alpha, x] = e_1'\,\mathsf{in}\,e_2' : \tau'}$$

$$\frac{\Gamma \vdash \tau : \mathsf{T} \quad \Gamma \vdash e\ R_t\ e' : (\mathsf{unit} \to \tau) \to \tau}{\Gamma \vdash \mathsf{fix}_\tau\ e\ R_t\ \mathsf{fix}_\tau\ e' : \tau}$$

$$\frac{\Gamma \vdash e_1\ R_t\ e_1' : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2\ R_t\ e_2' : \tau_2}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2\ R_t\ \mathsf{let}\,x = e_1'\,\mathsf{in}\,e_2' : \tau_2}$$

$$\frac{\Gamma \vdash M\ R_m\ M' : \sigma \quad \Gamma, \alpha/m{:}\sigma \vdash e\ R_t\ e' : \tau \quad \Gamma \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e\ R_t\ \mathsf{let}\,\alpha/m = M'\,\mathsf{in}\,e' : \tau}$$

$$\frac{\Gamma \vdash M\ R_m\ M' : \langle\!|\,\tau\,|\!\rangle}{\Gamma \vdash \mathsf{Ext}\,M\ R_t\ \mathsf{Ext}\,M' : \tau} \qquad \frac{\Gamma \vdash e\ R_t\ e' : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash e\ R_t\ e' : \tau'}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma, x{:}\tau_1 \vdash e\ R_t\ e' : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e\ R_v\ \lambda x{:}\tau_1.e' : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1\ R_v\ e_1' : \tau_1 \quad \Gamma \vdash e_2\ R_v\ e_2' : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle\ R_v\ \langle e_1', e_2'\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k; \Gamma \vdash e\ R_v\ e' : \tau}{\Gamma \vdash \Lambda\alpha{:}k.e\ R_v\ \Lambda\alpha{:}k.e' : \forall\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash e\ R_v\ e' : [c/\alpha]\tau \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau\ R_v\ \mathsf{pack}\,[c, e']\,\mathsf{as}\,\exists\alpha{:}k.\tau : \exists\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e\ R_v\ e' : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash e\ R_v\ e' : \tau'}$$

$$\frac{\Gamma \vdash c : k}{\Gamma_{\mathsf{P}} \vdash \langle\!|\,c\,|\!\rangle\ R_m\ \langle\!|\,c\,|\!\rangle : \langle\!|\,k\,|\!\rangle} \qquad \frac{\Gamma \vdash e\ R_t\ e' : \tau}{\Gamma \vdash_{\mathsf{P}} \langle\!|\,e\,|\!\rangle\ R_m\ \langle\!|\,e'\,|\!\rangle : \langle\!|\,\tau\,|\!\rangle}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{I}} M\ R_m\ M' : \sigma_2}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M\ R_m\ \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M' : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1\ R_m\ M_1' : \Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma' \qquad \Gamma \vdash_{\mathsf{P}} M_2\ R_m\ M_2' : \sigma \qquad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_{\mathsf{I}} M_1 M_2\ R_m\ M_1' M_2' : [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M\ R_m\ M' : \sigma_2}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M\ R_m\ \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M' : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_\kappa M_1\ R_m\ M_1' : \Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma' \qquad \Gamma \vdash_{\mathsf{P}} M_2\ R_m\ M_2' : \sigma \qquad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_\kappa M_1 \cdot M_2\ R_m\ M_1' \cdot M_2' : [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash_\kappa M_1\ R_m\ M_1' : \sigma_1 \quad \Gamma \vdash_\kappa M_2\ R_m\ M_2' : \sigma_2}{\Gamma \vdash_\kappa \langle M_1, M_2\rangle\ R_m\ \langle M_1', M_2'\rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M\ R_m\ M' : \Sigma\alpha{:}\sigma_1.\sigma_2}{\Gamma \vdash_{\mathsf{P}} \pi_1 M\ R_m\ \pi_1 M' : \sigma_1}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M\ R_m\ M' : \Sigma\alpha{:}\sigma_1.\sigma_2 \quad \Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash_{\mathsf{P}} \pi_2 M\ R_m\ \pi_2 M' : [\pi_1 c/\alpha]\sigma_2}$$

$$\frac{\Gamma \vdash e\ R_t\ e' : \exists\alpha{:}k.\tau \qquad \Gamma, \alpha{:}k, x{:}\tau \vdash_{\mathsf{I}} M\ R_m\ M' : \sigma \qquad \Gamma \vdash \sigma : \mathsf{sig}}{\begin{array}{l}\Gamma \vdash_{\mathsf{I}} \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(M : \sigma) \\ \qquad R_m\ \mathsf{unpack}\,[\alpha, x] = e'\,\mathsf{in}\,(M' : \sigma) : \sigma\end{array}}$$

$$\frac{\Gamma \vdash e\ R_t\ e' : \tau \quad \Gamma, x{:}\tau \vdash_\kappa M\ R_m\ M' : \sigma}{\Gamma \vdash_\kappa \mathsf{let}\,x = e\,\mathsf{in}\,M\ R_m\ \mathsf{let}\,x = e'\,\mathsf{in}\,M' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1\ R_m\ M_1' : \sigma \qquad \Gamma, \alpha/m{:}\sigma \vdash_{\mathsf{I}} M_2\ R_m\ M_2' : \sigma' \qquad \Gamma \vdash \sigma' : \mathsf{sig}}{\begin{array}{l}\Gamma \vdash_{\mathsf{I}} \mathsf{let}\,\alpha/m = M_1\,\mathsf{in}\,(M_2 : \sigma') \\ \qquad R_m\ \mathsf{let}\,\alpha/m = M_1'\,\mathsf{in}\,(M_2' : \sigma') : \sigma'\end{array}}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M\ R_m\ M' : \sigma}{\Gamma \vdash_{\mathsf{I}} (M :> \sigma)\ R_m\ (M' :> \sigma) : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M\ R_m\ M' : \sigma}{\Gamma \vdash_{\mathsf{I}} M\ R_m\ M' : \sigma} \qquad \frac{\Gamma \vdash_\kappa M\ R_m\ M' : \sigma \quad \Gamma \vdash \sigma \le \sigma'}{\Gamma \vdash_\kappa M\ R_m\ M' : \sigma'}$$

References

[1] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[2] Karl Crary. Modules, abstraction, and parametric polymorphism. In *Forty-Fourth ACM Symposium on Principles of Programming Languages*, Paris, France, January 2017.

[3] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *1999 SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, Atlanta, May 1999.

[4] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[5] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 2005.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 236–249, New Orleans, Louisiana, January 2003.

[7] Martin Elsman. Static interpretation of modules. In *1999 ACM International Conference on Functional Programming*, pages 208–219, Paris, France, 1999.

[8] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large. In *2018 ACM International Conference on Functional Programming*, 2018.

[9] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[10] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[11] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[12] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[13] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[14] Robert Harper and Benjamin C. Pierce. Design considerations for ML-Style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[15] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 2000. Extended version published as CMU technical report CMU-CS-97-147.

[16] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Thirty-Fourth ACM Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[17] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[18] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[19] Xavier Leroy. A proposal for recursive modules in Objective Caml. Available at `http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf`, 2003.

[20] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 4.03, Documentation and user's manual*. Institut National de Recherche en Informatique et Automatique (INRIA), 2016.

[21] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, January 1986.

[22] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1994.

[23] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[24] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[25] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation. *ACM Computing Surveys*, 2019. To appear.

[26] Andrew Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

[27] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5), September 2014.

[28] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, March 1998.

[29] Claudio V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4), 2000.

[30] Claudio V. Russo. Recursive structures for Standard ML. In *2001 ACM International Conference on Functional Programming*, 2001.

[31] Chung-chieh Shan. Higher-order modules in System F_ω and Haskell. Available at `homes.soic.indiana.edu/ccshan/xlate/xlate.pdf`, May 2006.

[32] Zhong Shao. Typed cross-module compilation. In *1998 ACM International Conference on Functional Programming*, pages 141–152, Baltimore, Maryland, September 1998.

[33] Zhong Shao. Transparent modules with fully syntactic signatures. In *1999 ACM International Conference on Functional Programming*, pages 220–232, Paris, September 1999.

[34] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, January 2000. Extended version published as CMU technical report CMU-CS-99-155.

[35] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4), October 2006. An earlier version appeared in the 2000 Symposium on Principles of Programming Languages.