Higher-order modules in System F_ω and Haskell

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Abstract

To argue that it is practical to extend core Haskell to support higher-order modules, we translate Dreyer, Crary, and Harper's higher-order module system (2002, 2003) into System F_ω. Our translation is the first to fully treat generative functors (with existential types) alongside applicative ones (with Skolemized types). Applicative functors correspond to higher-order polymorphism in idiomatic Haskell—abstract, higher-kinded type constructors, accompanied by term combinators. Higher-order functors correspond to higher-rank polymorphism in F_ω, which is practical to add to Haskell and has been added (Peyton Jones and Shields 2004).

The difference between generative and applicative functors boils down to whether existential type variables scope under or over the typing context. Thus we can express functor bodies that mix generative and applicative parts, which are inconvenient to simulate in Dreyer et al.'s language. Also, we elude the avoidance problem (the lack of minimal supersignatures) because existential quantification is primitive in F_ω. Although modules are first-class values, type sharing and the phase distinction are both preserved: the programmer can make fine-grained trade-offs between opaque and transparent types, and two modules are statically equivalent just in case they have the same type. Our work can guide further improvements to modularity in Haskell and ML.

1 Introduction

The ML family of programming languages is well-known for its sophisticated module systems. They not only allow a program to be divided into reusable parts, but also provide *higher-order* facilities to encapsulate transformations on these parts into *functors*, which can in turn be reused and transformed. These facilities have proven utility in real-world applications, such as a network protocol stack (Biagioni, Harper, Lee, and Milnes 1994) and an extensible interpreter (Ramsey 2003).

Types for modules and functors are called *signatures*. Signatures specify interfaces to modules and functors; as such, they serve as machine-checkable documentation for the programmer as well as guidance for separate compilation. Signatures can range from being fully *transparent* to being fully *opaque*, depending on how much (type) information about the implementation is exposed to clients of the module. When signatures are not fully transparent, they enforce abstraction barriers and ensure representation independence. For example, a symbol table module that maps between strings and symbols might expose the fact that strings are character lists, but not the fact that symbols are integers. If the implementation of the symbol table changes, say to represent symbols with pointers instead, then clients are guaranteed to remain compatible.

Statically-typed higher-order module systems are complex to design and a topic of ongoing research. Some central issues that have emerged are the following.

Generative vs applicative functors A *generative* functor (MacQueen 1986) with opacity in its body, when applied multiple times to the same input, yields type-incompatible results. By contrast, an *applicative* functor (Leroy 1995) yields type-compatible results. Both kinds of functors are called for in practice: A generative functor is more appropriate for creating symbol tables, because multiple symbol tables should have incompatible symbol types, so that different mappings are not confused with each other. An applicative functor is more appropriate to map an ordered type τ to the type of τ-sets, so that the types of τ_1-sets and τ_2-sets are compatible as long as the types τ_1 and τ_2 are.

The avoidance problem For purposes like type-checking let-module expressions, it is often useful to find the *minimal supersignature* σ' of a given signature σ that avoids mentioning some module variable s. That is, given σ and s, we seek a most informative σ' among the supersignatures of σ that avoid mentioning s. The *avoidance problem* (Ghelli and Pierce 1998; Lillibridge 1997) is that some signatures in some module systems have no minimal supersignature.

First-class modules ML-style module systems distinguish a core language for programming in the small from modularity mechanisms for programming in the large. The core language houses terms, which inhabit types; the module language houses modules, which inhabit signatures. This design needs additional machinery before modules can be stored in data structures, passed to and returned from functions, reconfigured based on run-time demands, and so on. A potentially simpler alternative is to unify the core language with the module language. Shields and Peyton Jones (2001, 2002) explore such a design for Haskell, which has a weak module language but a rich core language whose type system has attracted many proposed and implemented extensions.

Type sharing A statically typed module system needs to be able to decide whether two given types are compatible. For instance, a functor that maps an ordered type τ to the type of τ-sets identifies the ordered type in the input module with the element type in the output module for type-checking purposes, even if the functor is generative rather than applicative. Such type information is tricky to propagate in the presence of higher-order functors (Aspinall 1997; Shao 1999a,b; Stone 2000; Stone and Harper 2000).

The phase distinction Assuming type-checking should be decidable, and given that program execution can be undecidable, the compilation phase of a program, which is guaranteed to terminate, must be distinct from the execution phase, which may not. In particular, type compatibility must be determined with only compile-time information, not run-time computations. A simple way to enforce this *phase distinction* is *phase separation*: to split programs into a compile-time part and a run-time part, and make sure the former does not depend on the latter (Harper, Mitchell, and Moggi 1990).

1.1 Unity in search of clarity

The ML community has explored a great variety of module system designs. Motivated by the diversity of these efforts and apparent contradictions among them, Dreyer, Crary, and Harper recently proposed a unified formalism (2002, 2003) that deals with many design issues, including those above, in a single coherent design:

1. Generative and applicative type abstraction are treated as two different kinds of computational effects, *strong sealing* and *weak sealing*. Applicative functors are a subtype of generative ones: any functor can be generative, but the body of an applicative functor must be judged dynamically pure.
2. The avoidance problem is carefully circumvented by requiring supersignature annotations. This requirement is burdensome, so the programmer codes in an external language with *existential signatures* that does not require annotations. An *elaboration* algorithm translates this external language back to the internal language.
3. Following the ML tradition, modules and signatures are second-class, that is, distinguished from values and types. However, modules can be packed into first-class values using existential quantification (Mitchell, Meldal, and Madhav 1991), and a more convenient open-scope unpack construct can be added to the language as an orthogonal feature.
4. Type sharing information is propagated using *singleton signatures* (Aspinall 1997; Stone 2000; Stone and Harper 2000). Only compatible modules match each singleton signature.
5. The phase distinction is respected by considering only the compile-time aspects—*static equivalence*—of two modules when comparing them for type compatibility.

Because first-class modules are packed existentially, unpacking is generative in this system. Dreyer et al. thus argue that first-class modules propagate less type information than second-class modules, and are less expressive in this regard (unless the system is changed to break the phase distinction).

The Dreyer-Crary-Harper language represents one cutting edge in the design of module systems and provides an attractive basis for future research. However, it is formidable: one type system contains 73 proof rules for 9 judgment forms. The notions of abstraction effects and static equivalence are intertwined with other features, so it is unclear, say, what it would mean to replace static equivalence with dynamic equivalence, or why there are two kinds of abstraction effects and not one or three. Also, Dreyer et al. raise without answering the natural question of whether existential signatures can be incorporated into the internal language, obviating elaboration while preserving decidability. Finally, Dreyer et al.'s claim that first-class modules are not as expressive as second-class ones in their system does not entail that such must be the case in any module system that respects the phase distinction.

1.2 Contributions

This paper presents a type-directed translation from the Dreyer-Crary-Harper language to System F_ω (Girard 1972; Reynolds 1974) that elucidates the design issues above, answers open questions, and points the way to future improvements to modularity in ML and Haskell.

Generative versus applicative functors We fully treat both generative and applicative functors for the first time, encoding generative functors using existential types (Mitchell and Plotkin 1988; Russo 1998; Shao 1999a,b; Shields and Peyton Jones 2001, 2002) and applicative functors using Skolemized types (Jones 1995a, 1996; Russo 1998; Shao 1999a,b). As we explain in §4.2–3, applicative functors correspond to the idiomatic encoding of abstract data types in Haskell, so-called *higher-order polymorphism*—abstract, higher-kinded type constructors, accompanied by term combinators. The crucial distinction between generative and ap-

plicative abstraction turns out to be whether existential variables scope under or over the typing context. To express the latter, we make the novel move of mapping the typing context of modules to the right (rather than left) of the F_ω turnstile. Functors then come in not two varieties but a spectrum of possibilities: F_ω can express useful functors that are generative or applicative on a per-type-variable basis, which are inconvenient to simulate in Dreyer et al.'s language.

The avoidance problem Existential types are primitive in System F_ω, unlike in the Dreyer-Crary-Harper language. Thus, as we explain in §4.2, modular programming in F_ω does not need elaboration to deal with the avoidance problem. Although type reconstruction for Curry-style System F and F_ω is undecidable (Urzyczyn 1993; Wells 1999; these results intuitively explain why Dreyer et al. encounter undecidability when trying to make existential signatures primitive in their language) and it is impractical to require Church-style type annotations everywhere, extensions to the Hindley-Milner-Damas type system have been devised that provide the power of F or F_ω in exchange for a moderate number of type annotations (Le Botlan and Rémy 2003; Odersky and Läufer 1996), including an implemented extension to Haskell's core language (Jones 1997; Peyton Jones and Shields 2004). The success of higher-order polymorphism in Haskell prompts us to answer Dreyer et al.'s question positively: existential signatures *can* be incorporated in a decidable internal language with principal signatures and moderate syntactic overhead.

First-class modules, type sharing, and the phase distinction System F_ω has no intrinsic notion of modules and signatures, only terms and types, so all modules are first-class in our translation. Following Shao's phase-splitting approach (1997, 1998, 1999a,b), modules translate to a combination of types and terms, a split that enforces the phase distinction by separating compile-time and run-time information. As noted in §4.3 and contra Dreyer et al., our encoding expresses just as much type sharing as second-class modules. Sharing occurs exactly when existential quantification is absent, so the programmer can choose between sharing transparent types and packing opaque types on a per-type-variable basis. As one might expect, the phase distinction boils down to the non-dependent nature of F_ω: types cannot depend on terms.

Higher-order modules in Haskell In recent work, Shields and Peyton Jones (2001, 2002) propose several light extensions to Haskell that together encode a first-class, higher-order module system. Unfortunately, they only encode generative functors. The present paper can be thought to first desugar that work to F_ω, then extend it to applicative functors. We present our translation in F_ω, rather than Haskell extended with arbitrary-rank polymorphism (Peyton Jones and Shields 2004), because we find the explicit yet concise notation of (Church-style) F_ω a less distracting substrate. Besides, using F_ω relates our work more directly to other translations from higher-order modules into F_ω-like languages (Crary, Harper, and Puri 1999; Harper et al. 1990; Shao 1997, 1998, 1999a,b) and allows type-based compilation techniques to apply across modules (Shao 1997, 1998).

Our work helps Haskell programmers and implementors in practice by showing how higher-order polymorphism encodes applicative functors. This mapping underscores the importance of higher-kinded types for programming in the large. Although we do not recommend that Haskell programmers create higher-order modules through our translation, we do recommend that they create higher-order *signatures* so. This emphasis on higher-kinded types also informs language design, for example favoring record extension over qualified types to express sharing among type records (§5.1).

Organization This paper is organized as follows.

§2 introduces Dreyer et al.'s source language.
§3 introduces Girard and Reynolds's target language.
§4 presents the formal translation.
§5 reviews related work and advises future module systems.

2 The source language

Dreyer et al.'s language (2002, 2003) has the syntax below.

Types	τ	$::= \mathsf{Typ}\, M \mid \Pi s{:}\sigma.\, \tau \mid \tau_1 \times \tau_2 \mid \langle\!\lvert\sigma\rvert\!\rangle$
Terms	e	$::= \mathsf{Val}\, M \mid \langle e_1, e_2\rangle \mid \pi_1 M \mid \pi_2 M \mid eM$
		$\mid \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) \mid \mathsf{fix}\ f(s{:}\sigma){:}\tau.\, e$
		$\mid \mathsf{pack}\ M\ \mathsf{as}\ \langle\!\lvert\sigma\rvert\!\rangle$
Signatures	σ, ρ	$::= 1 \mid [\![T]\!] \mid [\![\tau]\!] \mid \mathfrak{S}(M) \mid \Sigma s{:}\sigma.\, \rho$
		$\mid \Pi^{\mathsf{par}} s{:}\sigma.\, \rho \mid \Pi^{\mathsf{tot}} s{:}\sigma.\, \rho$
Modules	M, N	$::= s \mid \langle\rangle \mid [\tau] \mid [e : \tau] \mid \lambda s{:}\sigma.\, M \mid MN$
		$\mid \langle s = M, N\rangle \mid \pi_1 M \mid \pi_2 M$
		$\mid \mathsf{unpack}\ e\ \mathsf{as}\ \sigma \mid M :> \sigma \mid M :: \sigma$
		$\mid \mathsf{let}\ s = M\ \mathsf{in}\ (N : \sigma)$
Contexts	Γ	$::= \bullet \mid \Gamma,\ s : \sigma$

We use the metavariables s and f for module variables. As usual, we take α-equivalent expressions to be identical, and write $E[M/s]$ for the capture-avoiding substitution of a module expression M for a module variable s in a type, term, signature, or module expression E.

The key to understanding this work is signatures. As defined above, a signature in this language can be one of the following. Table 1 expresses these cases in ML-like syntax.

1. The trivial signature 1 is inhabited by the module $\langle\rangle$.
2. The signature $[\![T]\!]$ is inhabited by modules $[\tau]$ with a single type component τ. That is, a module M with the signature $[\![T]\!]$ specifies a type $\mathsf{Typ}\, M$.
3. The signature $[\![\tau]\!]$ is inhabited by modules $[e : \tau]$ with a single value component e, of type τ. That is, a module M of signature $[\![\tau]\!]$ specifies a value $\mathsf{Val}\, M$ of type τ.
4. The signature $\mathfrak{S}(M)$ is only valid when the module M has the signature $[\![T]\!]$, that is, specifies some type τ. It is inhabited by modules $[\tau]$ that specify the same type τ. Such *singleton* signatures encode type sharing.

Table 1: Modules and signatures in the Dreyer-Crary-Harper language and ML-like syntax

	Dreyer et al.'s language		ML-like syntax	
	Modules	Signatures	Modules (**struct** ... **end**)	Signatures (**sig** ... **end**)
Trivial	$\langle\rangle : 1$		:	
Type	$[\tau] : [\![T]\!]$		**type** $t = \tau$	: **type** t
Value	$[e : \tau] : [\![\tau]\!]$		**val** $v = e : \tau$	: **val** $v : \tau$
Singleton	$M : \mathfrak{S}(M)$		**type** $t = M.t$	: **type** $t = M.t$
Structure	$\langle s = M, N\rangle : \Sigma s{:}\sigma.\ \rho$		**structure** $s = M$ **structure** $s' = N$	: **structure** $s : \sigma$ **structure** $s' : \rho$
Functor	$\lambda s{:}\sigma.\ M : \Pi^{\mathsf{tot/par}} s{:}\sigma.\ \rho$		**functor** $f(s : \sigma)\ M$	: **functor** $f(s : \sigma) : \rho$

5. The signature $\Sigma s{:}\sigma.\ \rho$ binds the module variable s in the signature ρ. It is inhabited by modules $\langle s = M, N\rangle$ whose two components have the signatures σ and ρ. Such *structure signatures* allow later parts of a module definition to refer back to earlier parts. The two components of a structure are selected using π_1 and π_2.
6. The signatures $\Pi^{\mathsf{tot}} s{:}\sigma.\ \rho$ and $\Pi^{\mathsf{par}} s{:}\sigma.\ \rho$ encode applicative and generative functors; tot and par stand for "total" and "partial". Functor modules are of the form $\lambda s{:}\sigma.\ M$. The argument module s binds into the return signature ρ, so the latter ("the type of τ-sets") can refer to the former ("an ordered type τ").

In structure and functor signatures (items 5 and 6 above), the module variable s binds into the signature ρ, which can thus refer to s in singleton signatures like $\mathfrak{S}(s)$. Such references evoke, and can be expressed using, dependent types (MacQueen 1986). However, this language respects the phase distinction: type compatibility is decidable at compile time.

Fig. 1 shows Dreyer et al.'s example of an ML signature and a matching module, and how this idealized language expresses them. Because the left hand side of a structure binds into the right, the signature for t can refer to the type component of s and match the module [bool × int]. This example involves no functors: the occurrences of Π indicate functions at the core language level, created using fix.

2.1 Two varieties of type abstraction

Dreyer et al. treat type abstraction as computational effects. Applicative and generative abstraction incur different effects. Generative functors, when invoked, incur *dynamic* effects, resulting from *strong sealing* in the body, written $M :> \sigma$. Applicative functors, when created, incur *static* effects, resulting from *weak sealing* in the body, written $M :: \sigma$.

Strong and weak sealing both ascribe a signature σ to a module M, possibly abstracting away some type components in M. Accordingly, we model both using existential quantification in F_ω (Mitchell and Plotkin 1988). Informally, however, ascription happens when applicative functors are created but when generative functors are invoked, so each call to a generative functor may create "new" abstract types.

For example, the left hand side of Fig. 2 defines an applicative functor SetFun in ML syntax, which turns any type-with-ordering module (signature ORD) into a set-type module (signature SET) for the same type. In Dreyer et al.'s language, the module expression for SetFun takes the form

$$\lambda \text{Elem}{:}\{\text{ORD}\}.\ (\ldots :: \{\text{SET } \textbf{where type elem} = \text{Elem.elem}\}), \quad (1)$$

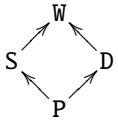
(braces indicate ellipsis) and matches a signature of the form

$$\Pi^{\text{tot}} \text{Elem}{:}\{\text{ORD}\}.\ \{\text{SET } \textbf{where type elem} = \text{Elem.elem}\}. \quad (2)$$

The functor body in (1) uses weak sealing. If we remove the sealing operation, leaving only "..." in the body, then the module (1) would match signatures more specific than (2) that inappropriately expose the returned set type. In other words, (2) would no longer be the *principal* signature of (1).

Because (1) uses weak sealing, the functor is "total" (tot in (2)), or applicative. When SetFun is applied twice to IntOrd at the bottom of Fig. 2, the resulting types IntSet1.set and IntSet2.set are interchangeable, although opaque. Weak sealing made the type system forget that IntSet1.set and IntSet2.set are both just "int list", while remembering that IntSet1.set and IntSet2.set are compatible. If weak sealing :: in (1) is changed to strong sealing :>, then the functor would be "partial" (par instead in (2)), or generative. The types IntSet1.set and IntSet2.set would then be incompatible.

Applicative functors can be implicitly coerced to generative ones by a subtyping relation ≤. Nevertheless, dynamic and static effects are treated as logically independent of each other: a module expression may be judged dynamically pure, statically pure, both, or neither. Thus there are four judgment forms for module expressions. All four follow the general format $\Gamma \vdash_{\mathtt{k}} M : \sigma$, where k is P, S, D, or W. These letters classify module expressions as: both dynamically and statically pure, statically pure, dynamically pure, or just well-formed. These purity levels form the lattice to the right, in which lower purity judgments entail higher ones. This ordering is written ⊑; meet and join are written ⊓ and ⊔.



Neither dynamic nor static effect correlates with computational effects at the core language level, such as mutable references, input/output, and control. This can be seen from the fact that a module expression M with the signature $[\![\tau]\!]$, where τ is any type, can always be turned into the equivalent module expression [Val $M : \tau$], which is judged pure.

sig		**struct**	
type s	$\Sigma s{:}[\![T]\!].$	**type** s = bool	$\langle s = [\mathrm{bool}],$
type t = s × int	$\Sigma t{:}\mathcal{S}([\mathsf{Typ}\,s{\times}\mathrm{int}]).$	**type** t = bool × int	$\langle t = [\mathrm{bool}\times\mathrm{int}],$
structure S : **sig**	$\Sigma S:($	**structure** S = **struct**	$\langle S =$
type u	$\Sigma u{:}[\![T]\!].$	**type** u = string	$\langle u = [\mathrm{string}],$
val f : u → s	$\Sigma f{:}[\![\Pi y{:}[\![\mathsf{Typ}u]\!].\,\mathsf{Typ}\,s]\!].$	**val** f = (fn y : u ⇒ true)	$\langle f = [\mathsf{fix}\ f(y{:}[\![\mathsf{Typ}u]\!]){:}\mathsf{Typ}\,s.\ \mathrm{true}],$
end	$1).$	**end**	$\langle\rangle\rangle,$
val g : t → S.u	$\Sigma g{:}[\![\Pi y{:}[\![\mathsf{Typ}t]\!].\,\mathsf{Typ}(\pi_1 S)]\!].$	**val** g = (fn y : t ⇒ "hello")	$\langle g = [\mathsf{fix}\ g(y{:}[\![\mathsf{Typ}t]\!]){:}\mathsf{Typ}(\pi_1 S).\ \text{“hello”}],$
end	1	**end**	$\langle\rangle\rangle\rangle\rangle$
(a)	(b)	(c)	(d)

Figure 1: A signature (a,b) and a matching module (c,d), in ML (a,c) and Dreyer et al.'s language (b,d)

```
signature ORD = sig
  type elem
  val compare : elem × elem → order
end
signature SET = sig (* persistent sets *)
  type elem
  type set
  val empty : set
  val insert : elem × set → set ...
end
functor SetFun(Elem : ORD)
  :: SET where type elem = Elem.elem = struct
  type elem = Elem.elem
  type set = elem list ...
end
structure IntOrd = struct
  type elem = int
  val compare = Int.compare
end
structure IntSet1 = SetFun(IntOrd)
structure IntSet2 = SetFun(IntOrd)
```

```
data ORD
  elem = ORD {
  compare :: (elem, elem) → Ordering
}
data SET {– persistent sets –}
  elem
  set = SET {
  empty :: set,
  insert :: (elem, set) → set, ...
}
data SETFUN
  = SETFUN (∃f. ∀elem. ORD elem → SET elem (f elem))
setFun =
  SETFUN (··· :: ∀elem. ORD elem → SET elem [elem])
main =
  case setFun of {SETFUN setFun′ →

  let intOrd = ORD compareInt :: ORD Int in

  let intSet1 = setFun′ intOrd in
  let intSet2 = setFun′ intOrd in ... }
```

Figure 2: An applicative functor that turns an ordered type into a set type. To the left is Dreyer et al.'s sample code using weak sealing (2002, 2003). To the right is a translation into Haskell using Skolem type constructors, described in this paper. Because the functor is applicative, not generative, the bottom two lines produce compatible modules.

Following Mitchell and Plotkin's pioneering work (1988) and earlier type-theoretic analyses of modules (Russo 1998; Shao 1999a,b; Shields and Peyton Jones 2001, 2002), our translation maps generative functors to functions returning existential types. Also, along lines suggested by Jones (1995a, 1996) and Russo (1998), and extending Shao's encoding (1999a,b), we map applicative functors to functions involving *Skolemized* types. As a preview, the right half of Fig. 2 shows the mapping at work on the applicative functor SetFun, in Haskell syntax. This example continues in §4.2.

3 The target language

Our syntax for System F_ω is fairly standard, as follows.

Kinds	$\kappa ::= \star \mid \kappa_1 \Rightarrow \kappa_2$
Types	$\tau ::= a \mid \tau_1 \to \tau_2 \mid \forall a{::}\kappa.\,\tau \mid \lambda a{::}\kappa.\,\tau \mid \tau_1\tau_2$ $\mid 1 \mid \tau_1 \times \tau_2 \mid \exists a{::}\kappa.\,\tau$
Terms	$e ::= x \mid \lambda x{:}\tau.\,e \mid e_1 e_2 \mid \Lambda a{::}\kappa.\,e \mid e\tau$ $\mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e$ $\mid \langle a = \tau_1, e : \tau_2\rangle \mid \mathsf{case}\ e_1\ \mathsf{of}\ \langle a,x\rangle.\ e_2$
Contexts	$\Gamma ::= \Gamma,\ a :: \kappa \mid \Gamma,\ x : \tau \mid \bullet$

We use the metavariables a, b, t for type variables, and x, y, z, w for term variables. We also write $\vec{s}, \vec{t}, \vec{f}$ for sequences of F_ω type variables, and $\bar{s}, \bar{t}, \bar{f}$ for F_ω term variables, even though s and f denote module variables in the Dreyer-Crary-Harper language, because our translation maps a module to a type sequence and a term. As with the Dreyer-Crary-Harper language, we identify α-equivalent expressions, and denote substitution by $[e/x]$ and $[\tau/a]$ postfix.

Product types and existential types are included for convenience, but for our purposes they can be translated away using universal quantification in continuation-passing style. For brevity, we sometimes omit the type annotation ": τ_2" in

Table 2: Translating judgment forms from the Dreyer-Crary-Harper language to System F_ω

	Dreyer et al.'s language	$\rightsquigarrow$	System F_ω
Well-formed contexts	$\Gamma \vdash \text{ok}$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \text{ok}$
Well-formed types	$\Gamma \vdash \tau\ \text{type}$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{\tau} :: \star$
Type equivalence	$\Gamma \vdash \tau_1 \equiv \tau_2$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{\tau} :: \star$
Well-formed terms	$\Gamma \vdash e : \tau$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{e} : \bar{\tau}$
Well-formed signatures	$\Gamma \vdash \sigma\ \text{sig}$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{\sigma} :: \kappa$
Signature equivalence	$\Gamma \vdash \sigma_1 \equiv \sigma_2$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{\sigma} :: \kappa$
Signature subtyping[†]	$\Gamma \vdash \sigma_1 \leq \sigma_2$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma},\ \vec{s} :: \hat{\bar{\sigma}}_1,\ \bar{s} : \bar{\sigma}_1 \vec{s} \vdash e : \bar{\sigma}_2 \vec{\tau}$
Well-formed modules[‡]	$\Gamma \vdash_{\mathtt{k}} M : \sigma$	$\rightsquigarrow$	$\Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma}\vec{\tau}$
Module equivalence	$\Gamma \vdash M \cong N : \sigma$	$\rightsquigarrow$	$\Gamma_0,\ \bar{\Gamma} \vdash \bar{\sigma}\vec{\tau} :: \star$

[†]The type variables $\vec{s}$ may appear free in $\vec{\tau}$, but not in $\bar{\sigma}_1$ or $\bar{\sigma}_2$.

[‡]The purity indicator $\mathtt{k}$ is $\mathtt{P}$, $\mathtt{S}$, $\mathtt{D}$, or $\mathtt{W}$. If $\mathtt{k} \sqsubseteq \mathtt{S}$, then $\vec{t}_0$ and $\hat{\kappa}_0$ are empty. If $\mathtt{k} \sqsubseteq \mathtt{D}$, then $\vec{t}_1$ and $\hat{\kappa}_1$ are empty.

the existential term $\langle a = \tau_1, e : \tau_2 \rangle$ when it is clear. We also abbreviate $\mathsf{case}\ e\ \mathsf{of}\ \langle \vec{t}, x \rangle.\ \langle \vec{t} = \vec{t}, e' \rangle$ to $\mathsf{open}\ e\ \mathsf{as}\ \langle \vec{t}, x \rangle.\ e'$. As usual, the quantifiers $\forall$ and $\exists$ scope as far as possible to the right, and the operators $\times$, $\rightarrow$, and $\Rightarrow$ associate to the right.

Higher-rank polymorphism occurs when a term argument has polymorphic type. *Higher-order polymorphism* occurs when a type argument has *higher* (that is, non-$\star$) kind.

Given any kind κ, we write $\hat{\kappa}$ for the sequence of kinds $\kappa_1, \ldots, \kappa_n$ such that κ is equal to $\kappa_1 \Rightarrow \cdots \Rightarrow \kappa_n \Rightarrow \star$. For example, if κ is $(\star \Rightarrow \star) \Rightarrow \star \Rightarrow \star$, then $\hat{\kappa}$ is the two-kind sequence $(\star \Rightarrow \star), \star$. Such kind sequences are only used to present our translation and not a formal part of the target language. They serve the same role as product or record kinds in other formulations. The sequence $\hat{\kappa}$ is empty if and only if κ is $\star$, in which case the kind $\hat{\kappa} \Rightarrow \kappa'$ is just κ'.

We abbreviate a kinding sequence $a_1 :: \kappa_1, \ldots, a_n :: \kappa_n$ to $\vec{a} :: \hat{\kappa}$. For example, if τ is a type, then we denote the type $\lambda a_1 :: \kappa_1 \ldots \lambda a_n :: \kappa_n . \tau$ by $\lambda \vec{a} :: \hat{\kappa} . \tau$. Moreover, if $\vec{\tau}$ is a sequence of types $\tau_1, \ldots, \tau_m$, then we abbreviate the sequence of type abstractions $\lambda \vec{a} :: \hat{\kappa}.\ \tau_1, \ldots, \lambda \vec{a} :: \hat{\kappa}.\ \tau_m$ to just $\lambda \vec{a} :: \hat{\kappa}.\ \vec{\tau}$.

Given a kind sequence $\hat{\kappa}$ and a kind κ', we write $\hat{\kappa} \Rightarrow \kappa'$ for the kind $\kappa_1 \Rightarrow \cdots \Rightarrow \kappa_n \Rightarrow \kappa'$, where n is the length of $\hat{\kappa}$. Thus κ is equal to $\hat{\kappa} \Rightarrow \star$ for any kind κ. When the kindings $\vec{\tau} :: \hat{\kappa}$ and $\tau' :: \hat{\kappa} \Rightarrow \kappa'$ hold, we can apply τ' to $\vec{\tau}$ to get the type $\tau' \tau_1 \cdots \tau_n$, or $\tau' \vec{\tau}$ for short, which has the kind κ'.

Given two kind sequences $\hat{\kappa}$ and $\hat{\kappa}'$, we write $\hat{\kappa} \Rightarrow \hat{\kappa}'$ for the kind sequence $\hat{\kappa} \Rightarrow \kappa'_1, \ldots, \hat{\kappa} \Rightarrow \kappa'_n$, where n is the length of $\hat{\kappa}'$. When the kindings $\vec{\tau} :: \hat{\kappa}$ and $\vec{\tau}' :: \hat{\kappa} \Rightarrow \hat{\kappa}'$ hold, we can apply each τ'_i to $\vec{\tau}$ to get the sequence of types $\tau'_1 \vec{\tau}, \ldots, \tau'_n \vec{\tau}$, or $\vec{\tau}' \vec{\tau}$ for short, of kinds $\hat{\kappa}'$.

If τ is a type of kind κ, then we denote $\hat{\kappa}$ with $\hat{\tau}$. We then shorten $\forall \vec{a} :: \hat{\tau}.\ \tau \vec{a}$ to $\forall \tau$, and $\exists \vec{a} :: \hat{\tau}.\ \tau \vec{a}$ to $\exists \tau$.

Suppose that Γ_1, Γ is a context in two parts. If $\Gamma_1, \Gamma \vdash \tau :: \star$ for some type τ, then we inductively define the type $\Pi \Gamma . \tau$, so that $\Gamma_1 \vdash \Pi \Gamma .\ \tau :: \star$ (reminiscent of the deduction theorem).

$$\begin{aligned} &\Pi(\Gamma, a :: \kappa).\ \tau = \Pi \Gamma .\ \forall a :: \kappa .\ \tau, \\ &\Pi(\Gamma, x : \tau').\ \tau = \Pi \Gamma .\ \tau' \rightarrow \tau, \qquad \Pi \bullet .\ \tau = \tau. \end{aligned} \tag{3}$$

Also, if $\Gamma_1, \Gamma \vdash e : \tau$ for some term e, then we define the term $\Lambda \Gamma .\ e$, such that $\Gamma_1 \vdash \Lambda \Gamma .\ e : \Pi \Gamma .\ \tau$.

$$\begin{aligned} &\Lambda(\Gamma, a :: \kappa).\ e = \Lambda \Gamma .\ \Lambda a :: \kappa .\ e, \\ &\Lambda(\Gamma, x : \tau').\ e = \Lambda \Gamma .\ \lambda x : \tau' .\ e, \qquad \Lambda \bullet .\ e = e. \end{aligned} \tag{4}$$

In the reverse direction, if $\Gamma_1 \vdash e : \Pi \Gamma .\ \tau$ for some term e, then we define the term $e\Gamma$, such that $\Gamma_1,\ \Gamma \vdash e\Gamma : \tau$.

$$e(\Gamma,\ a :: \kappa) = (e\Gamma)a, \quad e(\Gamma,\ x : \tau') = (e\Gamma)x, \quad e\bullet = e. \tag{5}$$

At the kind level, given a context Γ and kind κ, we define the kind $\Gamma \Rightarrow \kappa$ below.

$$\begin{aligned} &(\Gamma,\ a :: \kappa') \Rightarrow \kappa = \Gamma \Rightarrow \kappa' \Rightarrow \kappa, \\ &(\Gamma,\ x : \tau) \Rightarrow \kappa = \Gamma \Rightarrow \kappa, \qquad \bullet \Rightarrow \kappa = \kappa. \end{aligned} \tag{6}$$

Also, if $\Gamma_1, \Gamma \vdash \tau :: \kappa$ for some type τ, then we define the type $\lambda \Gamma .\ \tau$, such that $\Gamma_1 \vdash \lambda \Gamma .\ \tau :: \Gamma \Rightarrow \kappa$.

$$\begin{aligned} &\lambda(\Gamma, a :: \kappa').\ \tau = \lambda \Gamma .\ \lambda a :: \kappa' .\ \tau, \\ &\lambda(\Gamma, x : \tau').\ \tau = \lambda \Gamma .\ \tau, \qquad \lambda \bullet .\ \tau = \tau. \end{aligned} \tag{7}$$

In the reverse direction, if $\Gamma_1 \vdash \tau :: \Gamma \Rightarrow \kappa$ for some type τ, then we define the type $\tau\Gamma$, such that $\Gamma_1,\ \Gamma \vdash \tau\Gamma :: \kappa$.

$$\tau(\Gamma,\ a :: \kappa') = (\tau\Gamma)a, \quad \tau(\Gamma,\ x : \tau') = \tau\Gamma, \quad \tau\bullet = \tau. \tag{8}$$

We also write $\Gamma \Rightarrow \hat{\kappa}$ to distribute over a sequence of kinds $\hat{\kappa}$, and $\lambda \Gamma .\ \vec{\tau}$ and $\vec{\tau}\Gamma$ to distribute over a sequence of types $\vec{\tau}$.

4 The translation

Our translation is sensitive to the typing context: the translation of the right hand side of a Dreyer-Crary-Harper judgment depends on how the context on the left hand side is translated. Therefore, we present the mapping as a set of deduction rules for judgment forms. Table 2 summarizes the judgment forms involved. For each judgment form in the source type system, there is a matching translation judgment involving the symbol $\rightsquigarrow$, pronounced "translates to". The part of a translation judgment to the left of $\rightsquigarrow$ is a source judgment; to the right is a target judgment. Each translation rule usually corresponds to a typing rule of the source language, so translation rules are numbered according to how the source rules are numbered in Appendix A of Dreyer et al.'s technical report (2002).

We identify type expressions that are $\beta\eta$-equivalent in this translation. Put syntactically, we write type expressions below as shorthand for long $\beta\eta$-normal form. Put deductively, for each translation judgment form we add a proof rule that allows applying a β or η equation anywhere. For instance, for modules we have the rule

$$\frac{\bar{\Gamma} \equiv \bar{\Gamma}' \qquad \bar{\sigma}\vec{\tau} \equiv \bar{\sigma}'\vec{\tau}' \qquad \Gamma \vdash_{\mathrm{k}} M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1} :: \hat{\kappa}_1.\, \bar{\sigma}\vec{\tau}}{\Gamma \vdash_{\mathrm{k}} M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\, \Pi\bar{\Gamma}'.\, \exists \vec{t_1} :: \hat{\kappa}_1.\, \bar{\sigma}'\vec{\tau}'.} \quad (9)$$

The subsections below explain the judgment forms and deduction rules in more detail. Basically:

- contexts map to contexts;
- types map to types of kind $\star$;
- signatures map to types of all kinds;
- terms map to terms;
- modules map to types alongside terms; and
- compile-time equivalences map to type equivalences.

By structural induction on source-language derivations, we can show that any provable judgment in the source language translates to a provable judgment in F_ω. The concluding judgment of every source-language derivation has a unique translation. Also, the translation is efficiently computable.

Because Dreyer et al. do not specify an operational or denotational semantics for their language, we cannot argue formally that our translation preserves meaning and abstraction, or that it enforces representation independence. The description below serves as an informal argument.

4.1 Contexts and variables

The constant context Γ_0 in Table 2 is defined as follows.

$$\begin{aligned} \Gamma_0 \quad = \quad & \mathsf{Ty} :: \star \Rightarrow \star, \quad \mathsf{ty} : \forall a{::}\star.\, \mathsf{Ty}\, a, \\ & \mathsf{rec} : \forall a{::}\star.\, (a \to a) \to a. \end{aligned} \quad (10)$$

The Ty type constructor and ty term constant are used to encode the static part of a module, with a trivial run-time representation. For example, a module with just a type component $[\tau]$ (with the signature $[\![T]\!]$) maps to the term $\mathsf{ty}\,\bar{\tau}$ (of type $\mathsf{Ty}\,\bar{\tau}$), where $\bar{\tau}$ is the translation of τ. We can easily implement Ty and ty, as $\mathsf{Ty} = \lambda a{::}\star.\, 1$ and $\mathsf{ty} = \Lambda a{::}\star.\, \langle\rangle$.

The polymorphic fixed-point operator rec is used to translate fix (Rule 14 in Table 6 in the appendix). The recursion induces a computational effect—partiality—which we leave implicit in the types. It would be straightforward to make explicit in the types this or other effects such as mutable references, input/output, and control, at the cost of complicating the translation with continuations or a monad.

In the Dreyer-Crary-Harper language, every variable is a module variable, assigned a signature in the context. On the other hand, F_ω distinguishes type variables, which have kinds, from term variables, which have types. As Harper et al. note (1990), this distinction indicates phase separation: types contain compile-time information, and terms contain run-time information. Modules combine compile-time and run-time information, so they map to both types and terms.

Table 3: Translating contexts: $\Gamma \vdash \mathrm{ok} \;\rightsquigarrow\; \Gamma_0,\ \bar{\Gamma} \vdash \mathrm{ok}$

$$\frac{}{\bullet \vdash \mathrm{ok} \;\rightsquigarrow\; \Gamma_0 \vdash \mathrm{ok}}\ 1$$

$$\frac{\Gamma \vdash \sigma\ \mathrm{sig} \;\rightsquigarrow\; \Gamma_0,\ \bar{\Gamma} \vdash \bar{\sigma} :: \kappa}{\Gamma,\ s : \sigma \vdash \mathrm{ok} \;\rightsquigarrow\; \Gamma_0,\ \bar{\Gamma},\ \vec{s} :: \hat{\kappa},\ \bar{s} : \bar{\sigma}\vec{s} \vdash \mathrm{ok}}\ 2$$

For example, as shown in Table 2, translation judgments for pure modules have the form

$$\Gamma \vdash_{\mathrm{P}} M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \Pi\bar{\Gamma}.\, \bar{\sigma}\vec{\tau}. \quad (11)$$

The F_ω-context $\bar{\Gamma}$ translates the context Γ, and the F_ω-type $\bar{\sigma}$ translates the signature σ. Crucially, the module expression M translates to a sequence of type expressions $\vec{\tau}$ (of kinds $\hat{\vec{\sigma}}$) alongside a term $\bar{M}$ (of type $\Pi\bar{\Gamma}.\, \bar{\sigma}\vec{\tau}$). The first of these two parts, the type sequence $\vec{\tau}$, is known at compile time and appears in the type of $\bar{M}$. Not being a dependent type system, F_ω provides no way for the second part $\bar{M}$ to influence $\vec{\tau}$. Thus is enforced phase separation.

The translation of well-formed contexts is shown in Table 3. Respecting the phase distinction as explained above, we translate each module variable s to a sequence $\vec{s}$ of type variables, followed by a term variable $\bar{s}$, whose type may mention $\vec{s}$. Recall from Table 2 that each signature σ translates to a type expression $\bar{\sigma}$, of kind κ not necessarily $\star$. As Rule 2 shows, when a module binding $s : \sigma$ is added to the context, the type arguments that $\bar{\sigma}$ must be applied to before arriving at a type of kind $\star$ become the type variables $\vec{s}$ (the compile-time component of s), and the result $\bar{\sigma}\vec{s}$ of these applications becomes the type of $\bar{s}$ (the run-time component of s). The kind sequence $\hat{\vec{\sigma}}$ is analogous to Shao's *flexroot constructor* (1999a,b). The type sequence $\vec{s}$ is called the *realization* or *flexroot instantiation* of s.

4.2 Types and signatures

Types and signatures both translate to types in F_ω. The rules are shown in Tables 4 and 5. Source types always translate to kind-$\star$ types in the target, because a source type contains a single piece of compile-time information, with no run-time counterpart. Signatures, on the other hand, translate to types of arbitrary kind. A signature σ specifies first the compile-time information $\vec{s} :: \hat{\vec{\sigma}}$ required of a matching module $s : \sigma$, then the run-time information $\bar{s} : \bar{\sigma}\vec{s}$ required.

The signature translation, rule by rule Each rule in Table 5 translates signatures of a different form. Let us examine more closely the translation of each signature form in turn, in the order followed in §2. For reference, we also identify in parentheses the module translation rules for each signature form, discussed in §4.3.

Trivial Rule 20 (40) translates the trivial signature 1 to the unit type 1. The latter has kind $\star$, so a trivial module provides no compile-time information. Its run-time information has the unit type 1 in F_ω.

Table 4: Translating types: $\Gamma \vdash \tau \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau} :: \star$

$$\frac{\Gamma \vdash_{\mathrm{P}} M : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \bar{M} : \Pi\bar{\Gamma}.\ \mathsf{Ty}\,\bar{\tau}}{\Gamma \vdash \mathsf{Typ}\,M \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau} :: \star}\ 3 \qquad \frac{\Gamma,\ s : \sigma \vdash \tau \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \vec{s} :: \hat{\kappa},\ \bar{s} : \bar{\sigma}\vec{s} \vdash \bar{\tau} :: \star}{\Gamma \vdash \Pi s{:}\sigma.\ \tau \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \bar{\tau} :: \star}\ 4$$

$$\frac{\Gamma \vdash \tau_1 \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau}_1 :: \star \qquad \Gamma \vdash \tau_2 \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau}_2 :: \star}{\Gamma \vdash \tau_1 \times \tau_2 \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau}_1 \times \bar{\tau}_2 :: \star}\ 5 \qquad \frac{\Gamma \vdash \sigma \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\sigma} :: \kappa}{\Gamma \vdash \langle\!|\sigma|\!\rangle \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \exists \bar{\sigma} :: \star}\ 6$$

Table 5: Translating signatures: $\Gamma \vdash \sigma \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\sigma} :: \kappa$

$$\frac{\Gamma \vdash \mathrm{ok} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathrm{ok}}{\Gamma \vdash 1 \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash 1 :: \star}\ 20 \qquad \frac{\Gamma \vdash \mathrm{ok} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathsf{Ty} :: \star \Rightarrow \star}\ 21 \qquad \frac{\Gamma \vdash \tau \text{ type} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau} :: \star}{\Gamma \vdash [\![\tau]\!] \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau} :: \star}\ 22$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \bar{M} : \Pi\bar{\Gamma}.\ \mathsf{Ty}\,\bar{\tau}}{\Gamma \vdash \mathfrak{S}(M) \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathsf{Ty}\,\bar{\tau} :: \star}\ 23 \qquad \frac{\Gamma,\ s : \sigma \vdash \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \vec{s} :: \hat{\kappa},\ \bar{s} : \bar{\sigma}\vec{s} \vdash \bar{\rho} :: \kappa'}{\Gamma \vdash \Pi^{\mathsf{par}} s{:}\sigma.\ \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \exists \bar{\rho} :: \star}\ 24^{\mathsf{par}}$$

$$\frac{\Gamma,\ s : \sigma \vdash \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \vec{s} :: \hat{\kappa},\ \bar{s} : \bar{\sigma}\vec{s} \vdash \bar{\rho} :: \kappa'}{\Gamma \vdash \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \lambda \vec{t} :: \hat{\kappa} \Rightarrow \hat{\kappa}'.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \bar{\rho}(\vec{t}\vec{s}) :: (\hat{\kappa} \Rightarrow \hat{\kappa}') \Rightarrow \star}\ 24^{\mathsf{tot}}$$

$$\frac{\Gamma,\ s : \sigma \vdash \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \vec{s} :: \hat{\kappa},\ \bar{s} : \bar{\sigma}\vec{s} \vdash \bar{\rho} :: \kappa'}{\Gamma \vdash \Sigma s{:}\sigma.\ \rho \text{ sig} \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \lambda \vec{s} :: \hat{\kappa}.\ \lambda \vec{t} :: \hat{\kappa}'.\ \bar{\sigma}\vec{s} \times \bar{\rho}\vec{t} :: \hat{\kappa} \Rightarrow \kappa'}\ 25$$

Type Rule 21 (41) translates the type signature $[\![T]\!]$ to the type constructor Ty, of kind $\star \Rightarrow \star$. As this kind indicates, the compile-time component of a $[\![T]\!]$-module is a type $\bar{\tau}$ of kind $\star$ (for example, the type bool for the module s in Fig. 1). The run-time component is then a value of type $\mathsf{Ty}\,\bar{\tau}$ (for example the value ty bool, of type Ty bool), which is trivial, as explained in §4.1.

Value Rule 22 (42) translates the value signature $[\![\tau]\!]$ by translating the source type τ to an F_ω-type $\bar{\tau}$, as detailed in Table 4. The signature $[\![\tau]\!]$ then also maps to $\bar{\tau}$. Because $\bar{\tau}$ has kind $\star$, a $[\![\tau]\!]$-module has empty realization—the signature already specifies the type $\bar{\tau}$. For example, in Fig. 1, because the type $\mathsf{Typ}\,u$ translates to the F_ω-type u_1, the signature $[\![\mathsf{Typ}\,u]\!]$ does too. Similarly, the type $\Pi y{:}[\![\mathsf{Typ}u]\!].\ \mathsf{Typ}\,s$ and the signature $[\![\Pi y{:}[\![\mathsf{Typ}u]\!].\ \mathsf{Typ}\,s]\!]$ both translate to the F_ω-type $u_1 \to s_1$. A module matching that signature (such as f in Fig. 1) corresponds to an F_ω-value of that F_ω-type.

Singleton To translate a singleton signature $\mathfrak{S}(M)$, Rule 23 first translates the module M at the signature $[\![T]\!]$. Since $[\![T]\!]$ translates to Ty by Rule 21, M must translate to a type $\bar{\tau}$ of kind $\star$, along with a trivial value of type $\mathsf{Ty}\,\bar{\tau}$. The signature $\mathfrak{S}(M)$ then translates to that type $\mathsf{Ty}\,\bar{\tau}$, of kind $\star$. A module of this signature provides no compile-time information, because the signature already determines the type $\bar{\tau}$.

It is no coincidence that $\mathsf{Ty}\,\bar{\tau}$ translates not just $\mathfrak{S}(M)$, but also $[\![T]\!]$ applied to the realization $\bar{\tau}$. This way, M matches both the signatures $[\![T]\!]$ and $\mathfrak{S}(M)$. For example, the run-time component in F_ω of the module t in Fig. 1 is the value $\mathsf{ty}(\mathrm{bool} \times \mathrm{int})$, of type $\mathsf{Ty}(\mathrm{bool} \times \mathrm{int})$. This type is Ty applied to bool × int, as well as $\mathsf{Ty}(\mathrm{bool} \times \mathrm{int})$ applied to nothing. Thus t matches both the signatures $[\![T]\!]$ and $\mathfrak{S}([\mathsf{Typ}\ s \times \mathrm{int}])$.

Structure Rule 25 (47–49) translates the structure signature $\Sigma s{:}\sigma.\ \rho$ by translating ρ with σ in the context, say to $\bar{\rho}$, of kind κ'. Higher up in the proof tree, that context is eventually translated using Rule 2 in Table 3, which translates σ, say to $\bar{\sigma}$, of kind κ. A σ-module then has realization kinds $\hat{\kappa}$, and a ρ-module has realization kinds $\hat{\kappa}'$. The combined signature $\Sigma s{:}\sigma.\ \rho$, is realized by a type sequence whose kinds are $\hat{\kappa}$ and $\hat{\kappa}'$ concatenated. Hence the translation of the signature has kind $\hat{\kappa} \Rightarrow \hat{\kappa}' \Rightarrow \star$, or (written another way) $\hat{\kappa} \Rightarrow \kappa'$. At run-time, structure modules are just pairs in F_ω.

For example, the structure S in Fig. 1 combines two components, whose signatures translate to the F_ω-types Ty (of kind $\star \Rightarrow \star$) and $(u_1 \to s_1) \times 1$ (of kind $\star$). Thus S translates to $\lambda u_1 :: \star.\ \mathsf{Ty}\,u_1 \times (u_1 \to s_1) \times 1$, of kind $\star \Rightarrow \star$.

Functor Rules 24^{par} and 24^{tot} (43–46) translate functors. Both rules first translate the body ρ with the argument σ in the context.

Rule 24^{par} is for generative functors. It produces a polymorphic function type from any σ-module—that is, any realization $\vec{s}$ of kinds $\hat{\vec{\sigma}}$, accompanied by a value of type $\bar{\sigma}\vec{s}$—to a ρ-module whose realization is comprised of existential variables only. Invoking such a function twice produces two sets of existential variables—hence, two incompatible ρ-modules. As the kind $\star$ shows, generative functors have empty realization.

Rule 24^{tot} is responsible for applicative functors, which by contrast do provide some compile-time information. An applicative functor is realized by type functions mapping input realizations to output realizations. In other words, the functor Skolemizes the compile-time component of its output. For each output kind κ'_i, the functor provides a type-level Skolem function from $\hat{\kappa}$ to κ'_i, in other words a type of kind $\hat{\kappa} \Rightarrow \kappa'_i$. Hence the compile-time component of the functor is a type sequence of kinds $\hat{\kappa} \Rightarrow \hat{\kappa}'$. The translation of an applicative functor thus has the kind $(\hat{\kappa} \Rightarrow \hat{\kappa}') \Rightarrow \star$. This rule is the sole source of higher-kinded type arguments

in the translation image. Hence we claim that higher-order polymorphism corresponds to applicative abstraction.

The run-time component of an applicative functor is a polymorphic function, like that of a generative functor, but its return type is not existentially quantified, so calling it does not generate a "new" type.

Example We illustrate how we translate functor signatures using the skeletal code for ordered types and sets in Fig. 2. The signature ORD there can be written in the Dreyer-Crary-Harper language as

$$\bullet,\ o : [\![T]\!] \vdash \Sigma e{:}[\![T]\!].\ \Sigma c{:}[\![\Pi x{:}[\![\mathsf{Typ}e \times \mathsf{Typ}e]\!].\ \mathsf{Typ}o]\!].\ 1 \text{ sig} \quad (12)$$

where e, c, and o stand for "elem", "compare", and "order". Fig. 3 in the appendix translates (12) to a type sequent in F_ω:

$$\Gamma_0,\ o_1 :: \star,\ \bar{o} : \mathsf{Ty}\, o_1 \vdash \lambda e_1 :: \star.\ \mathsf{Ty}\, e_1 \times ((e_1 \times e_1) \to o_1) \times 1 :: \star \Rightarrow \star. \quad (13)$$

Similarly, the signature SET in Fig. 2 translates from

$$\bullet \vdash \Sigma e{:}[\![T]\!].\ \Sigma s{:}[\![T]\!].\ \Sigma y{:}[\![\mathsf{Typ}s]\!].\ \Sigma i{:}[\![\Pi x{:}[\![\mathsf{Typ}e \times \mathsf{Typ}s]\!].\ \mathsf{Typ}s]\!].\ 1 \text{ sig} \quad (14)$$

(where s, y, and i abbreviate "set", "empty", and "insert") to

$$\Gamma_0 \vdash \lambda e_1 :: \star.\ \lambda s_1 :: \star.\ \mathsf{Ty}\, e_1 \times \mathsf{Ty}\, s_1 \times s_1 \times ((e_1 \times s_1) \to s_1) \times 1 :: \star \Rightarrow \star \Rightarrow \star. \quad (15)$$

More interesting is the signature of SetFun, an applicative functor. To express its type sharing in the source language, a singleton signature is introduced on the third line below.

$$\bullet,\ o : [\![T]\!] \vdash \Pi^{\mathsf{tot}} elem{:}(\Sigma e{:}[\![T]\!].\Sigma c{:}[\![\Pi x{:}[\![\mathsf{Typ}e \times \mathsf{Typ}e]\!].\ \mathsf{Typ}o]\!].1).\ \Sigma e{:}\mathfrak{S}(\pi_1 elem).\ \Sigma s{:}[\![T]\!].\ \Sigma y{:}[\![\mathsf{Typ}s]\!].\ \Sigma i{:}[\![\Pi x{:}[\![\mathsf{Typ}e \times \mathsf{Typ}s]\!].\ \mathsf{Typ}s]\!].\ 1 \text{ sig} \quad (16)$$

Using Rule 24^{tot}, we can translate this signature into F_ω.

$$\Gamma_0,\ o_1 :: \star,\ \bar{o} : \mathsf{Ty}\, o_1 \vdash \lambda s_1 :: \star \Rightarrow \star.\ \forall e_1 :: \star.\ (\mathsf{Ty}\, e_1 \times ((e_1 \times e_1) \to o_1) \times 1) \to (\mathsf{Ty}\, e_1 \times \mathsf{Ty}(s_1 e_1) \times s_1 e_1 \times ((e_1 \times s_1 e_1) \to s_1 e_1) \times 1) :: (\star \Rightarrow \star) \Rightarrow \star. \quad (17)$$

The type variable s_1 in (15) has kind $\star$, but in (17) it is Skolemized to take the kind $\star \Rightarrow \star$. The additional type argument is the element type e_1: the type of sets is s_1 in (15) but $s_1 e_1$ in (17). The upshot is that applying SetFun to equivalent element types (say, $e_1 \equiv e_1'$) yields equivalent set types (that is, $s_1 e_1 \equiv s_1 e_1'$). If s_1 is instantiated with an existential type variable (as weak sealing does, in §4.3 below, in particular Rule 51), then the type system would not equate these set types with any concrete type like list int, even though it would still equate them with each other, as desired.

If we change SetFun to a generative functor by changing tot to par in (16), then Rule 24^{par} would translate it to

$$\Gamma_0,\ o_1 :: \star,\ \bar{o} : \mathsf{Ty}\, o_1 \vdash \forall e_1 :: \star.\ (\mathsf{Ty}\, e_1 \times ((e_1 \times e_1) \to o_1) \times 1) \to \exists s_1 :: \star.\ \mathsf{Ty}\, e_1 \times \mathsf{Ty}\, s_1 \times s_1 \times ((e_1 \times s_1) \to s_1) \times 1 :: \star. \quad (18)$$

As always with generative functors, this type is of kind $\star$: it no longer takes Skolemized types as realization arguments. As the "$\to \exists$" indicates, applying this SetFun to the same element type twice yields inequivalent set types.

The avoidance problem For a given F_ω-type to be the translation of some signature, it must "name every type before using it". To name a type τ, when τ is of kind $\star$, is to provide (trivial) data of the type $\mathsf{Ty}\,\tau$. For instance, the following type expressions, of kind $\star \Rightarrow \star$, may be translations of some signatures.

$$\lambda s_1 :: \star.\ \mathsf{Ty}\, s_1 \quad (19a)$$

$$\lambda s_1 :: \star.\ \mathsf{Ty}\, s_1 \times s_1 \quad (19b)$$

$$\lambda s_1 :: \star.\ \mathsf{Ty}\, s_1 \times (\forall s_1' :: \star.\ \mathsf{Ty}\, s_1' \to \mathsf{Ty}\, s_1) \quad (19c)$$

Indeed, they are the translations of $[\![T]\!]$, $\Sigma s{:}[\![T]\!].\ [\![s]\!]$, and $\Sigma s{:}[\![T]\!].\ \Pi^{\mathsf{tot}} s'{:}[\![T]\!].\ \mathfrak{S}(s)$. By contrast, the type expressions below, also of kind $\star \Rightarrow \star$, cannot be the translation of any signature because they use s_1 without naming s_1 to the left.

$$\lambda s_1 :: \star.\ s_1 \quad (19d)$$

$$\lambda s_1 :: \star.\ s_1 \times \mathsf{Ty}\, s_1 \quad (19e)$$

$$\lambda s_1 :: \star.\ (\forall s_1' :: \star.\ \mathsf{Ty}\, s_1' \to \mathsf{Ty}\, s_1) \times \mathsf{Ty}\, s_1 \quad (19f)$$

Dreyer et al. uses (19f) to illustrate the avoidance problem. Their example is that the signature

$$\Sigma f{:}(\Pi^{\mathsf{tot}} s'{:}[\![T]\!].\mathfrak{S}(s)).\ \mathfrak{S}(s) \quad (20)$$

has no minimal supersignature that avoids mentioning s. This signature translates to the body of (19f),

$$(\forall s_1' :: \star.\ \mathsf{Ty}\, s_1' \to \mathsf{Ty}\, s_1) \times \mathsf{Ty}\, s_1. \quad (21)$$

To find a minimal supersignature of (20) avoiding s is to find a signature whose translation in F_ω is (21) with s_1 existentially quantified. Because (21) fails to name s_1 before using it, there is no such signature. In general, the avoidance problem occurs when existential quantification puts an F_ω-type beyond the image of the signature translation.

To circumvent the avoidance problem, Harper and Stone (2000) and Dreyer et al. propose that the programmer code in an external language with existential signatures. In the external language, the minimal s-avoiding supersignature of any signature is just that signature with "$\exists s :: \sigma$." in front. On this proposal, an elaboration algorithm translates this external language, which does not enjoy principal signatures, to the internal language, which does not support existential signatures but enjoys principal signatures. The elaborator essentially names types behind the programmer's back.

Instead of two languages related by an elaborator, it would be preferable to use a single module language with both existential signatures and principal signatures. Dreyer et al. briefly consider such a language, and the present paper shows that F_ω is another such language. Unfortunately, in neither language is type reconstruction decidable, and requiring type annotations everywhere is impractical. However, by asking the programmer for a moderate amount of type annotations, type reconstruction for System F and F_ω can be made practical (Le Botlan and Rémy 2003; Oder-

sky and Läufer 1996), in particular implemented by extending Haskell's core language (Jones 1997; Peyton Jones and Shields 2004). Haskell extended thus satisfies Dreyer et al.'s four desiderata: existential signatures, principal signatures, moderate syntactic overhead, and decidable type-checking.

4.3 Terms and modules

As one might expect, terms in the Dreyer-Crary-Harper language translate to terms in F_ω. Modules, on the other hand, translate to a type sequence plus a term. For example, the SetFun functor in Fig. 2 realizes the signature translation (17), of kind $(\star \Rightarrow \star) \Rightarrow \star$, with the list type constructor, of kind $\star \Rightarrow \star$. The details are in Tables 6 and 7 in the appendix.

Sealing and effects In the general case, a module expression M translates to a term $\bar{M}$ of the type

$$\exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\sigma}\vec{\tau}. \tag{22}$$

There are two sequences of existential quantification in this format. The first sequence $\vec{t_0} :: \hat{\kappa}_0$, which scopes over the context $\bar{\Gamma}$, is added to by static effects, or weak sealing. Intuitively, the compiler statically generates these types before $\bar{\Gamma}$ is instantiated by any environment. The second sequence $\vec{t_1} :: \hat{\kappa}_1$, which scopes under the context $\bar{\Gamma}$, is added to by dynamic effects, or strong sealing. These types are dynamically generated within particular environments that instantiate $\bar{\Gamma}$. To accommodate the first sequence, we cannot put $\bar{\Gamma}$ to the left of the F_ω turnstile; instead, we put "$\Pi\bar{\Gamma}$." to the right of the turnstile. This novel move is crucial to our translating weak sealing into F_ω, even though applicative abstraction can be expressed more simply in F_ω outside the translation image.

The difference between static and dynamic abstraction manifests in Rules 51 (for weak sealing) and 52 (for strong sealing). Both kinds of sealing achieve type abstraction by existentially quantifying over the realization $\vec{\tau}$ of a signature $\bar{\sigma}$ by a module $\bar{M}$.

Rule 52 translates strong sealing, which is well-understood in the literature. It replaces each realization type τ_i by a new existential variable t_i that scopes under the context.

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \sigma \ \rightsquigarrow\ \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\sigma}\vec{\tau}}{\begin{array}{l}\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma \ \rightsquigarrow \\ \quad \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle. \\ \qquad \Lambda\bar{\Gamma}.\ \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, y\rangle.\ \langle \vec{t} = \vec{\tau}, y : \bar{\sigma}\vec{t}\rangle \\ \qquad\qquad : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \exists \bar{\sigma}\end{array}}\ 52$$

Subsequently, the type system only remembers that $\bar{M}$ realizes $\bar{\sigma}$ with *some* type sequence: the conclusion type does not depend on $\vec{\tau}$, so representation independence is ensured by an abstraction barrier. This is how Shao (1999a,b) and Shields and Peyton Jones (2001, 2002) encode generativity.

Rule 51 translates weak sealing, which is newly characterized in this work. It replaces each τ_i in the realization with a series of type applications $t_i\bar{\Gamma}\vec{t_1}$, where $\bar{\Gamma}$ and $\vec{t_1}$ are the type variables that scope over τ_i, and the new existential variable t_i scopes over the context.

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \sigma \ \rightsquigarrow\ \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\sigma}\vec{\tau}}{\begin{array}{l}\Gamma \vdash_{\mathtt{k}\sqcup\mathtt{D}} (M :: \sigma) : \sigma \ \rightsquigarrow \\ \quad \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle. \\ \qquad \langle \vec{t} = \lambda\bar{\Gamma}.\ \lambda\vec{t_1} :: \hat{\kappa}_1.\ \vec{\tau}, x : \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\sigma}(t\bar{\Gamma}\vec{t_1})\rangle \\ \qquad\quad : \exists \vec{t_0} :: \hat{\kappa}_0.\ \exists \vec{t} :: \bar{\Gamma} \Rightarrow \hat{\kappa}_1 \Rightarrow \hat{\sigma}.\ \Pi\bar{\Gamma}.\ \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\sigma}(t\bar{\Gamma}\vec{t_1})\end{array}}\ 51$$

Subsequently, the type system only remembers that $\bar{M}$ realizes $\bar{\sigma}$ with sequence of types *that can be abstracted over any type variable in the context*. For example, if $a :: \kappa$ is a type variable in $\bar{\Gamma}$, then $t_i\bar{\Gamma}\vec{t_1}$ can be abstracted over a to give $\lambda a :: \kappa.\ t_i\bar{\Gamma}\vec{t_1}$. This trick is Skolemization, or λ-lifting (Johnsson 1985): whereas strong sealing produces quantified types like $\forall a :: \kappa. \exists b :: \kappa'. \cdots b \cdots$, where a universal variable scopes over an existential variable, weak sealing produces types like $\exists b :: \kappa \Rightarrow \kappa'.\ \forall a :: \kappa.\ \cdots b a \cdots$, where an existential *function* variable scopes over a universal variable. (In Rule 51 above, these universal variables are implicit in $\Pi\bar{\Gamma}$.) As with strong sealing, the conclusion type does not depend on $\vec{\tau}$, so this translation of weak sealing preserves type abstraction and representation independence.

Rule 43 makes applicative functors. It puts no existential quantifier on the result realization $\bar{\rho}\vec{\tau}$. In fact, it leaves the term $\bar{M}$ intact, relying on the dynamic purity of the premise.

$$\frac{\Gamma,\ s : \sigma \vdash_{\mathtt{k}} M : \rho \ \rightsquigarrow\ \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \bar{\rho}\vec{\tau} \qquad \mathtt{k} \sqsubseteq \mathtt{D}}{\begin{array}{l}\Gamma \vdash_{\mathtt{k}} \lambda s{:}\sigma.\ M : \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \ \rightsquigarrow \\ \quad \Gamma_0 \vdash \bar{M} \\ \qquad : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ (\lambda \vec{t} :: \hat{\kappa} \Rightarrow \hat{\rho}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \bar{\rho}(\vec{t}\vec{s}))(\lambda \vec{s} :: \hat{\kappa}.\ \vec{\tau})\end{array}}\ 43$$

When a module expression is dynamically impure, only a generative functor can be made, by Rule 44. A generative functor does existentially quantify over the result realization.

$$\frac{\Gamma,\ s : \sigma \vdash_{\mathtt{k}} M : \rho \ \rightsquigarrow\ \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \exists \vec{t_1} :: \hat{\kappa}_1.\ \bar{\rho}\vec{\tau}}{\begin{array}{l}\Gamma \vdash_{\mathtt{k}\sqcap\mathtt{D}} \lambda s{:}\sigma.\ M : \Pi^{\mathsf{par}} s{:}\sigma.\ \rho \ \rightsquigarrow \\ \quad \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\ \Lambda\bar{\Gamma}.\ \Lambda\vec{s} :: \hat{\kappa}.\ \lambda\bar{s}{:}\bar{\sigma}\vec{s}. \\ \qquad\quad \mathsf{case}\ x\bar{\Gamma}\vec{s}\bar{s}\ \mathsf{of}\ \langle \vec{t_1}, y\rangle.\ \langle \vec{t} = \vec{\tau}, y : \bar{\rho}\vec{t}\rangle \\ \qquad\qquad : \exists \vec{t_0} :: \hat{\kappa}_0.\ \Pi\bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma}\vec{s} \to \exists \bar{\rho}\end{array}}\ 44$$

Our translation's use of Skolemization for weak sealing sets it apart from Shields and Peyton Jones's encoding (2001, 2002), which only treats generative functors, as well as Shao's (1999a,b), which supports both applicative and generative functor signatures, but only strong and not weak sealing. In Shao's system, the programmer creates an applicative functor by first making a transparent functor and then coercing it to an applicative opaque functor by subtyping. As Dreyer et al. note, this technique only works when every component of the functor body is transparent within the body—in particular, if the body does not define any datatypes. Dreyer et al. lift this restriction by introducing weak sealing, which we faithfully encode.

Shao and us both translate applicative functors to higher-kinded type constructors. Special cases of this technique are

widespread in Haskell, showing its utility and usability. For example, container data structures (Okasaki 2000) and monads (Wadler 1997) are customarily represented by type constructors of kind $\star \Rightarrow \star$, and monad transformers (Liang, Hudak, and Jones 1995) have the kind $(\star \Rightarrow \star) \Rightarrow (\star \Rightarrow \star)$. These type transformations are accompanied by term combinators, for example to turn an ordering on τ-values into an ordering on τ-arrays, or to define monad primitives in terms of those of a transformed monad. Some of these term combinators are encapsulated in type classes, while others are left as standalone functions. Either way, the type-level maps correspond to $\vec{t}$ in Rules 24^{tot}, 43, and 45, whereas the term-level maps correspond to $\forall\vec{s}{::}\hat{\kappa}.\ \bar{\sigma}\vec{s} \to \bar{\rho}(\vec{t}\vec{s})$.

Mixing generative and applicative abstraction Our encoding of sealing using two sequences of existential quantifiers leads us to contemplate a hybrid between weak and strong sealing. For example, consider the signature of a functor that creates symbol tables:

functor(String : **sig type** string ... **end**) : **sig**
  type string = String.string
  type symbol
  val insert : string → symbol
  datatype result = FOUND **of** string | NOT_FOUND
  val lookup : symbol → result ...
end = ...

Should this functor be generative or applicative? On one hand, each symbol table contains its own mapping between symbols and strings, so abstract "symbol" types returned by successive invocations of SymbolTable—even on the same String module—should be incompatible with each other. On the other hand, we want to share the "result" datatype across symbol tables with the same string type (while keeping it incompatible with other algebraic datatypes such as "string option"). The Standard ML code above does not.

In the Dreyer-Crary-Harper language, the body of a functor cannot mix generative and applicative components. Such hybrid functors can only be simulated by a pair of functors, one generative and one applicative. This limitation is due to the use of an abstraction effect system with a finite number of purity levels. By contrast, F_ω can directly represent hybrid functors. The signature of a hybrid functor is an F_ω-type

$$\lambda\vec{t}{::}\hat{\kappa}{\Rightarrow}\hat{\kappa}'.\ \forall\vec{s}{::}\hat{\kappa}.\ \bar{\sigma}\vec{s} \to \exists(\bar{\rho}(\vec{t}\vec{s})) :: (\hat{\kappa} \Rightarrow \hat{\kappa}') \Rightarrow \star \qquad (23)$$

(cf. Rules 24^{par} and 24^{tot} in Table 5), where $\bar{\sigma}$ is the input signature (of kind κ) and $\bar{\rho}$ is the output signature (of kind $\hat{\kappa}' \Rightarrow \kappa''$ for some κ''). The type arguments $\vec{t}$ express the applicative part of the functor body, whereas the existential quantifier expresses the generative part. Whether each argument to $\bar{\rho}$ is generative can thus be specified individually.

Signature subtyping Signature subtyping is translated in Table 8 in the appendix. A subtyping relationship $\sigma_1 \le \sigma_2$ translates to an F_ω-term that converts σ_1-modules to σ_2-modules. Because implicit subtyping in the Dreyer-Crary-Harper language is carried out by explicit terms in F_ω, the translation of a module depends on the signature at which it is typed. We need explicit terms to translate subtyping, in particular to convert applicative functors to generative ones.

First-class modules Dreyer et al.'s first-class modules are straightforward to translate (see Rules 6 in Table 4, 50 in Table 7, and 19 in Table 6 in the appendix). For example, if the source signature σ translates to the F_ω-type $\bar{\sigma}$ of kind κ, then a list of σ-modules can be stored in an F_ω-value of type list$(\exists\vec{s}{::}\hat{\kappa}.\ \bar{\sigma}\vec{s})$. Quantifying over $\vec{s}$ makes the type system forget these modules' realizations, so that several modules in a homogeneous list can realize the same signature differently. Unpacking any of them incurs dynamic impurity.

This intentional forgetting of realizations leads Dreyer et al. to state that "first-class modules cannot propagate as much type information as second-class modules can." We blame the reduced propagation of type information not on treating modules as first-class values—that is, storing modules in data structures, passing them to and returning them from functions, and so on—but on homogenizing types by making them opaque. The types of first-class modules can express sharing just fine if we leave them transparent rather than existentially quantifying over them right away (Shao 1999a,b). For example, the F_ω-type $\exists\vec{s}{::}\hat{\kappa}.\,\mathrm{list}(\bar{\sigma}\vec{s})$ expresses a list of statically equivalent σ-modules. Heterogeneous lists are expressed as tuples in F_ω; for example, $\bar{\sigma}\vec{s} \times \bar{\sigma}\vec{s}' \times \bar{\sigma}\vec{s}''$ expresses a heterogeneous list of three σ-modules.

Just as with the generative-applicative distinction, there is a spectrum of options as to how much type information to propagate, from none to all. For example, if $\bar{\sigma}$ has the kind $\hat{\kappa}_1 \Rightarrow \hat{\kappa}_2 \Rightarrow \star$, then the type $\exists\vec{s}_1{::}\hat{\kappa}_1.\ \mathrm{list}(\exists\vec{s}_2{::}\hat{\kappa}_2.\ \bar{\sigma}\vec{s}_1\vec{s}_2)$ expresses a list of *partially* statically equivalent σ-modules—modules that realize σ with the same $\vec{s}_1$ but incompatible $\vec{s}_2$.

As an aside, the way F_ω statically expresses type sharing among modules can be ported back to Dreyer et al.'s system, where (contrary to their statement) singleton signatures enable first-class modules to express as much type sharing as second-class modules can. The appendix shows an example.

4.4 Compile-time equivalences

Four signature forms always map to F_ω-types of kind $\star$, indicating that matching modules have empty realization: the trivial signature 1; the value signature $[\![\tau]\!]$; the singleton signature $\mathfrak{S}(M)$; and the generative functor signature $\Pi^{\mathsf{par}} s{:}\sigma.\rho$. Dreyer et al. call these signatures *unitary*, except they surprisingly exclude singletons.

Table 9 in the appendix translates notions of equivalence. Remarkably, 26 proof rules in the Dreyer-Crary-Harper language are handled by just 3 translation rules. Two types or signatures are equivalent just in case they translate to equivalent types in F_ω (which we regard as equal). Hence type sharing among modules is encoded with type equality in F_ω.

Because our translation segregates compile-time information into types and run-time information into terms, *static*

equivalence among modules ("equivalence for type checking purposes") is also just type equality. Suppose that the signature σ translates to $\bar{\sigma}$, and two modules M and N realize σ with $\vec{\tau}$ and $\vec{\tau}'$. Then M and N are statically equivalent at σ if and only if $\bar{\sigma}\vec{\tau}$ and $\bar{\sigma}\vec{\tau}'$ are equivalent. Put another way, two modules are statically equivalent just in case they translate to terms of the same type. As a special case, two modules at the same unitary (or singleton) signature are always statically equivalent, because $\vec{\tau}$ and $\vec{\tau}'$ are both empty.

Due to singletons, whether two modules are statically equivalent depends on the signature they are compared at. Our translation captures this dependence by making signature subtyping explicit and dependent on the target signature.

5 Discussion: related and future work

This paper translates Dreyer et al.'s module language (2002, 2003) to System F_ω. Module systems have long been elucidated by translation to languages similar to F_ω, but this paper is the first to fully treat generative and applicative abstraction, preserving representation independence for both. We extend Shao's phase-splitting (1999a,b) to deal with applicative functors using Skolemized types, as suggested by Jones (1995a, 1996) and Russo (1998).

To briefly summarize previous translations from module languages to F_ω: Harper et al. (1990) translate two module calculi λ^{ML}_{mod} and λ^{ML}_{str} into an F_ω-like language λ^{ML}. They preserve phase separation to keep type-checking decidable, but did not treat type generativity. Taking Harper et al.'s formalism as their starting point, Crary et al. (1999) study *recursive modules*, which Dreyer et al. and us do not address.

Shao (1999a,b) translates several higher-order module languages into another F_ω-like language FTC, so as to endow the source languages with simple type-theoretic semantics. His system supports fully transparent functors by keeping track of each module's realization. He also treats generative functors using existential quantification, as we do here. However, as Dreyer et al. explain, because Shao's system only has strong sealing, his encoding of applicative functors only works if the functor body is fully transparent. By comparison, this paper completely encodes applicative functors by Skolemizing types, so that they can be existentially quantified outside the scope of the current context.

What is this translation good for? To be sure, our translation is not intended for direct programmer use, like F_ω itself and Dreyer et al.'s internal and external languages. These languages lack basic syntactic sugar like component labels for structures and records. They also provide no way to define synonyms for a signature, or to add sharing constraints to a signature, short of rewriting it at every occurrence. Still, these languages are practical in the sense that they support both generative and applicative functors (as is called for in practice) and feature decidable type-checking.

Whereas Dreyer et al. treat applicative functors by weak sealing inside the body, we treat them by higher-kinded types outside the body. Thus our translation of weak sealing must constantly abstract over the context, which makes for clumsy programming (without implying that any language is easier or harder to use—cf. translations between the Turing machine and the untyped λ-calculus). Hence we do *not* propose that Haskell programmers create terms via our module translation. Rather, our point is that *languages based on F_ω, such as Haskell extended with higher-rank polymorphism, already support higher-order modular programming.* As mentioned in §4.3, examples of the utility and usability of such programming are rife in the literature (for instance Jones 1995b). We *do* recommend that Haskell programmers, now aware that higher-kinded types encode applicative functors, create *types* via our *signature* translation. Having codified this existing practice, we can then guide future work on modularity in Haskell and ML, as we now discuss.

5.1 Higher-order modules in Haskell

What Haskell calls its module system (see Diatchki, Jones, and Hallgren 2002 for a formalization) is a facility for namespace management and (to some extent) separate compilation. It does not support functors. On one hand, Jones (1995a, 1996) uses higher-order polymorphism in Haskell to encode first-class modules, in particular realizations and type-returning functors. Although he notes the connection between generativity and existential types, he does not pursue type abstraction. Sheard and Pasalic (2003) exercise Jones's encoding in two extended programming examples, which they call "strong evidence that type-parameterized modules really work." On the other hand, Shields and Peyton Jones (2001, 2002) encode generative functors with existential types, but drop Jones's use of higher-kinded type constructors to encode type-returning functors.

We encode higher-order functors with higher-rank polymorphism, and applicative functors with higher-order polymorphism. ML-style modules thus share their usability issues with higher-rank and higher-order polymorphism in Haskell. We discuss three of these usability issues here.

Sharing style We pass module realizations as arguments to type constructors, as Jones (1995a, 1996) and Shao (1999a,b) do. Harper and Pierce (2003) term this way of expressing sharing *sharing by construction*. Unlike ML-style *sharing by specification*, sharing by construction does not scale by itself (Harper and Pierce 2003; Jones 1995a; MacQueen 1986), because type arguments can proliferate when composing signatures, and existing code may have to change globally when modifying signatures.

To alleviate this burden, Jones and Shao both turn to *type records*. Type records are records at the type rather than value level (on the latter, see Jones and Peyton Jones 1999 and references therein). If the type constructor $\bar{\sigma}$ has many realization arguments $\tau_1 :: \kappa_1, \ldots, \tau_n :: \kappa_n$, we can collect them

in a record of types $\{l_1 = \tau_1, \ldots, l_n = \tau_n\}$, of the record kind $\{l_1 :: \kappa_1, \ldots, l_n :: \kappa_n\}$. Essentially, $\tau_1, \ldots, \tau_n$ are keyword arguments to $\bar{\sigma}$. For example, the signatures ORD and SET in Fig. 2 can be rewritten to take one argument each, as follows.

```
data ORD (ro :: {elem :: ⋆}) = ORD
  {compare :: (ro.elem, ro.elem) → Ordering}
data SET (rs :: {elem :: ⋆, set :: ⋆}) = SET
  {empty :: rs.set, insert :: (rs.elem, rs.set) → rs.set, ...}
```

We explicitly kind the type arguments ro and rs for clarity. In practice, these kinds can be inferred by an extension of standard kind inference (Peyton Jones 2003; §4.6). Or one could add (record) kind polymorphism to Haskell, so that more general kinds can be expressed and inferred.

Jones proposes to express sharing among type records using either *qualified types* (Jones 1992) or *record extension*. Either proposal addresses the scalability concerns above, by allowing a potentially exponential (in the size of the source code for the module) number of types to be passed together yet a few of them singled out for unification. With qualified types, a polymorphic function that takes as input an ORD and a SET, sharing their elem types, would have the type

```
∀ro::{elem::⋆}. ∀rs::{elem::⋆,set::⋆}.
    (ro.elem ≡ rs.elem) ⇒ ORD ro → SET rs → ⋯ ,
```

wherein ≡ denotes an equality constraint. Using record extension, one would express the same sharing with the type

```
∀ro::{}. ∀rs::{set::⋆}. ∀e::⋆.
  ORD {ro | elem = e} → SET {rs | elem = e} → ⋯ .
```

Here $\{r \mid l = \tau\}$ means to extend the record r by the entry $l = \tau$.

At first glance, qualified types may seem to express sharing more directly, but record extension turns out to be more straightforward for applicative functors. For example, the SetFun functor is easy to type using record extension:

```
∃fs::{elem::⋆}⇒{set::⋆}.
      ∀ro::{}. ∀e::⋆. ORD {ro | elem = e} →
                  SET {fs {ro | elem = e} | elem = e}.
```

(Following the definition of $\hat{\kappa} \Rightarrow \hat{\kappa}'$ in §3, we equate the kinds $\{\text{elem} :: \star\} \Rightarrow \{\text{set} :: \star\}$ and $\{\text{set} :: \{\text{elem} :: \star\} \Rightarrow \star\}$.) By contrast, quantified types seem to need universal equality constraints (which are alien to Haskell, but see Trifonov 2003) and higher-order- or E-unification (which is undecidable):

```
∃fs::{elem::⋆}⇒{elem::⋆,set::⋆}.
    (∀ro::{elem::⋆}. (fs ro).elem ≡ ro.elem) ⇒
          ∀ro::{elem::⋆}. ORD ro → SET (fs ro).
```

For the sake of applicative abstraction, then, sharing among type records should be expressed using record extension.

Existential elimination To ease generative abstraction in Haskell, Shields and Peyton Jones (2001, 2002) propose an open-scope ∃-elimination construct, similar to Dreyer et al.'s unpack. This construct also eases our encoding, so that

```
data SETFUN = SETFUN (∃f. ⟨type⟩)
main = case SETFUN ⟨term⟩ of SETFUN setFun′ → ⋯
```

in Fig. 2 can be replaced with the less clumsy

```
open setFun′ = ⟨term⟩ :: ∃f. ⟨type⟩
main = ⋯
```

This example shows that our work enables this **open** construct to ease not just generative abstraction as Shields and Peyton Jones envision, but also applicative abstraction.

Subtyping On one hand, we translate implicit subtyping in the Dreyer-Crary-Harper language to explicit conversion in F_ω (§4.3). On the other hand, subtyping and *bounded quantification* in F_ω (Cardelli and Wegner 1985) have been extensively studied, in particular using *record subtyping* to model object orientation (Pierce and Turner 1994). Adding record subtyping to F_ω would let us drop unused structure components for signature matching. Further adding bounded quantification (yielding $F^{\leq}_{\omega}$) would give (generative and applicative) *partial abstraction*, and unify object systems with module systems. As with plain F_ω (Peyton Jones and Shields 2004), algorithms for *partial* (especially *local*) type inference (Odersky, Zenger, and Zenger 2001; Pierce and Turner 2000) would make programming in $F^{\leq}_{\omega}$ practical.

5.2 ML-style module systems

This paper answers two questions about ML-style modules that Dreyer et al. left open.

The first question is the *avoidance problem*: can a higher-order module system with decidable type-checking support existential signatures and principal signatures with moderate syntactic overhead? Yes (§4.2): Haskell extended with higher-rank polymorphism (Peyton Jones and Shields 2004) is one such system, though much of the evidence of moderate syntactic overhead leaves the connection between Haskell types and ML signatures implicit.

The second question concerns *type sharing among first-class modules*: can a higher-order module system propagate type sharing information among first-class modules while retaining the phase distinction? Yes (§4.3): System F_ω is one such system, as is Dreyer et al.'s own system. The tradeoff between propagating sharing information decidably and treating modules as first-class values is thus an illusion.

Refined abstraction and sharing As explained in §4.3, it is easy and useful in F_ω and Haskell to mix generative and applicative abstraction, and to fully propagate type sharing information for first-class modules. These patterns are inconvenient to simulate in Dreyer et al.'s language, because it uses a coarse-grained effect system that does not keep track of each type variable separately. Should these patterns be desired in ML-style module systems (as §4.3 argues they are), one might refine Dreyer et al.'s effect system to keep track of individual existential quantifiers. The sealing operations in the module syntax would also need to change.

Type classes Type classes and higher-order polymorphism in Haskell amplify each other's utility (Jones 1995b). Our

translation thus suggests that the ML family would benefit from a type-class facility at the module (rather than core) language level. Such a facility would bridge modules and type classes, like Kahl and Scheffczyk's named type-class instances (2001), but in the opposite direction.

6 Acknowledgements

Thanks to Matthew Fluet, Oleg Kiselyov, Greg Morrisett, Simon Peyton Jones, Norman Ramsey, Stuart Shieber, and 5 anonymous referees. This work is supported by the United States National Science Foundation Grant BCS-0236592.

References

Aspinall, David R. 1997. Type systems for modular programs and specifications. Ph.D. thesis, University of Edinburgh.

Biagioni, Edoardo, Robert Harper, Peter Lee, and Brian G. Milnes. 1994. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM conference on Lisp and functional programming*, 55–64. New York: ACM Press.

Cardelli, Luca, and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* 17(4):471–522.

Crary, Karl, Robert Harper, and Sidd Puri. 1999. What is a recursive module? In *PLDI '99: Proceedings of the ACM conference on programming language design and implementation*, vol. 34(5) of *ACM SIGPLAN Notices*, 50–63. New York: ACM Press.

Diatchki, Iavor S., Mark P. Jones, and Thomas Hallgren. 2002. A formal specification of the Haskell 98 module system. In *Proceedings of the 2002 Haskell workshop*, ed. Manuel Chakravarty, 17–28. New York: ACM Press.

Dreyer, Derek, Karl Crary, and Robert Harper. 2002. A type system for higher-order modules. Tech. Rep. CMU-CS-02-122R, School of Computer Science, Carnegie Mellon University.

———. 2003. A type system for higher-order modules. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 236–249. New York: ACM Press.

Ghelli, Giorgio, and Benjamin C. Pierce. 1998. Bounded existentials and minimal typing. *Theoretical Computer Science* 193(1–2):75–96.

Girard, Jean-Yves. 1972. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII.

Harper, Robert, John C. Mitchell, and Eugenio Moggi. 1990. Higher-order modules and the phase distinction. In *POPL '90: Conference record of the annual ACM symposium on principles of programming languages*, 341–354. New York: ACM Press.

Harper, Robert, and Benjamin C. Pierce. 2003. Design issues in advanced module systems. In *Advanced topics in types and programming languages*, ed. Benjamin C. Pierce. Cambridge: MIT Press. Draft manuscript.

Harper, Robert, and Chris Stone. 2000. A type-theoretic interpretation of Standard ML. In *Proof, language, and interaction: Essays in honour of Robin Milner*, ed. Gordon Plotkin, Colin Stirling, and Mads Tofte, chap. 12. Cambridge: MIT Press.

Johnsson, Thomas. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, ed. Jean-Pierre Jouannaud, 190–205. Lecture Notes in Computer Science 201, Berlin: Springer-Verlag.

Jones, Mark P. 1992. Qualified types: Theory and practice. Ph.D. thesis, Programming Research Group, Oxford University Computing Laboratory. Published in 1994 by Cambridge University Press.

———. 1995a. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell workshop*, ed. Paul Hudak. Tech. Rep. YALEU/DCS/RR-1075, New Haven: Department of Computer Science, Yale University.

———. 1995b. Functional programming with overloading and higher-order polymorphism. In *Advanced functional programming: 1st international spring school on advanced functional programming techniques*, ed. Johan Jeuring and Erik Meijer, 97–136. Lecture Notes in Computer Science 925, Berlin: Springer-Verlag.

———. 1996. Using parameterized signatures to express modular structure. In POPL (1996), 68–78.

———. 1997. First-class polymorphism with type inference. In *POPL '97: Conference record of the annual ACM symposium on principles of programming languages*, 483–496. New York: ACM Press.

Jones, Mark P., and Simon L. Peyton Jones. 1999. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell workshop*, ed. Erik Meijer. Tech. Rep. UU-CS-1999-28, Department of Computer Science, Utrecht University.

Kahl, Wolfram, and Jan Scheffczyk. 2001. Named instances for Haskell type classes. In *Proceedings of the 2001 Haskell workshop*, ed. Ralf Hinze, 71–99. Tech. Rep. UU-CS-2001-23, Department of Computer Science, Utrecht University.

Le Botlan, Didier, and Didier Rémy. 2003. ML^F: Raising ML to the power of System F. In *ICFP '03: Proceedings of the ACM international conference on functional programming*, 27–38. New York: ACM Press.

Leroy, Xavier. 1995. Applicative functors and fully transparent higher-order modules. In POPL (1995), 142–153.

Liang, Sheng, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In POPL (1995), 333–343.

Lillibridge, Mark. 1997. Translucent sums: A foundation for higher-order module systems. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-97-122.

MacQueen, David B. 1986. Using dependent types to express modular structure. In *POPL '86: Conference record of the annual ACM symposium on principles of programming languages*, 277–286. New York: ACM Press.

Mitchell, John C., Sigurd Meldal, and Neel Madhav. 1991. An extension of Standard ML modules with subtyping and inheritance. In *POPL '91: Conference record of the annual ACM symposium on principles of programming languages*, 270–278. New York: ACM Press.

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

Odersky, Martin, and Konstantin Läufer. 1996. Putting type annotations to work. In POPL (1996), 54–67.

Odersky, Martin, Matthias Zenger, and Christoph Zenger. 2001. Colored local type inference. In *POPL '01: Conference record of the annual ACM symposium on principles of programming languages*, 41–53. New York: ACM Press.

Okasaki, Chris. 2000. An overview of Edison. In *Proceedings of the 2000 Haskell workshop*, ed. Graham Hutton, 34–45. Electronic Notes in Theoretical Computer Science 41(1), Amsterdam: Elsevier Science. Also as Tech. Rep. NOTTCS-TR-00-1, School of Computer Science and Information Technology, University of Nottingham.

Peyton Jones, Simon L. 2003. The Haskell 98 language and libraries. *Journal of Functional Programming* 13(1):1–255.

Peyton Jones, Simon L., and Mark B. Shields. 2004. Practical type inference for arbitrary-rank types. `http://research.microsoft.com/~simonpj/papers/putting/`.

Pierce, Benjamin C., and David N. Turner. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming* 4(2):207–247.

———. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22(1):1–44.

POPL. 1995. *POPL '95: Conference record of the annual ACM symposium on principles of programming languages*. New York: ACM Press.

———. 1996. *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*. New York: ACM Press.

Ramsey, Norman. 2003. ML module mania: A type-safe, separately compiled, extensible interpreter. Submitted to the *Journal of Functional Programming*. `http://www.eecs.harvard.edu/~nr/pubs/maniaj-abstract.html`.

Reynolds, John C. 1974. Towards a theory of type structure. In *Programming symposium: Proceedings, colloque sur la programmation*, ed. Bernard Robinet, 408–425. Lecture Notes in Computer Science 19, Berlin: Springer-Verlag.

Russo, Claudio V. 1998. Types for modules. Ph.D. thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh. Also as Tech. Rep. ECS-LFCS-98-389.

Shao, Zhong. 1997. Typed cross-module compilation. Tech. Rep. YALEU/DCS/TR-1126, Department of Computer Science, Yale University, New Haven. Revised Jun. 1998.

———. 1998. Typed cross-module compilation. In *ICFP '98: Proceedings of the ACM international conference on functional programming*, vol. 34(1) of *ACM SIGPLAN Notices*, 141–152. New York: ACM Press.

———. 1999a. Transparent modules with fully syntactic signatures. In *ICFP '99: Proceedings of the ACM international conference on functional programming*, vol. 34(9) of *ACM SIGPLAN Notices*, 220–232. New York: ACM Press.

———. 1999b. Transparent modules with fully syntactic signatures. Tech. Rep. YALEU/DCS/TR-1181, Department of Computer Science, Yale University, New Haven.

Sheard, Tim, and Emir Pasalic. 2003. Two-level types and parameterized modules. *Journal of Functional Programming*. To appear.

Shields, Mark B., and Simon L. Peyton Jones. 2001. First-class modules for Haskell. Tech. Rep., Microsoft Research. `http://www.cse.ogi.edu/~mbs/pub/first_class_modules/`.

———. 2002. First-class modules for Haskell. In *9th international workshop on foundations of object-oriented languages*, 28–40. New York: ACM Press.

Stone, Christopher A. 2000. Singleton kinds and singleton types. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-00-153.

Stone, Christopher A., and Robert Harper. 2000. Deciding type equivalence in a language with singleton kinds. In *POPL '00: Conference record of the annual ACM symposium on principles of programming languages*, 214–227. New York: ACM Press.

Trifonov, Valery. 2003. Simulating quantified class constraints. In *Proceedings of the 2003 Haskell workshop*, 98–102. New York: ACM Press.

Urzyczyn, Paweł. 1993. Type reconstruction in F_ω is undecidable. In *TLCA '93: Proceedings of the international conference on typed lambda calculi and applications*, ed. Marc Bezem and Jan Frisco Groote, 418–432. Lecture Notes in Computer Science 664, Berlin: Springer-Verlag.

Wadler, Philip L. 1997. How to declare an imperative. *ACM Computing Surveys* 29(3):240–263.

Wells, Joe B. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98(1–3):111–156.

Appendix

Type sharing among first-class modules (§4.3)

The way F_ω statically expresses type sharing among modules can be ported back to Dreyer et al.'s system, where (contrary to their statement) singleton signatures enable first-class modules to express as much type sharing as second-class modules can. To continue the example in Fig. 2, even though any list of (type-incompatible) first-class ORD modules matches the signature

$$[\![\mathrm{list}\,\langle\!|\Sigma e{:}[\![T]\!].\ \Sigma c{:}[\![\Pi x{:}[\![\mathsf{Typ}e\times\mathsf{Typ}e]\!].\,\mathsf{Typ}o]\!].\ 1|\!\rangle]\!] \qquad (24)$$

(cf. the ORD signature (12)), only a list of type-equivalent first-class ORD modules matches the signature

$$\Sigma e'{:}[\![T]\!].\quad [\![\mathrm{list}\,\langle\!|\Sigma e{:}\mathsf{S}(e').\ \Sigma c{:}[\![\Pi x{:}[\![\mathsf{Typ}e\times\mathsf{Typ}e]\!].\,\mathsf{Typ}o]\!].\ 1|\!\rangle]\!]. \qquad (25)$$

We use a structure signature here, for lack of existential signatures in Dreyer et al.'s internal language. For usability, it may be possible to extend their elaboration algorithm to insert the first component e' automatically. Just as with the avoidance problem in §4.2, such a maneuver is unnecessary in F_ω because existential signatures are primitive.

$$\dfrac{}{\bullet \vdash \mathsf{ok} \rightsquigarrow \Gamma_0 \vdash \mathsf{ok}}\,1}}{\bullet \vdash [\![T]\!]\ \mathsf{sig} \rightsquigarrow \Gamma_0 \vdash \mathsf{Ty} :: \star \Rightarrow \star}\,21}}{\bullet, o : [\![T]\!] \vdash \mathsf{ok} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1 \vdash \mathsf{ok}}\,2}}{\bullet, o : [\![T]\!] \vdash [\![T]\!]\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1 \vdash \mathsf{Ty} :: \star \Rightarrow \star}\,21}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash \mathsf{ok} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash \mathsf{ok}}\,2}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash_{\mathrm{P}} e : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \Lambda o_1 :: \star.\, \lambda \bar{o} : \mathsf{Ty} o_1.\, \Lambda e_1 :: \star.\, \lambda \bar{e} : \mathsf{Ty} e_1.\, \bar{e} : \forall o_1 :: \star.\, \mathsf{Ty}\, o_1 \to \forall e_1 :: \star.\, \mathsf{Ty}\, e_1 \to \mathsf{Ty}\, e_1}\,39}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash \mathsf{Typ}\, e\ \mathsf{type} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash e_1 :: \star}\,3}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash \mathsf{Typ}\, e \times \mathsf{Typ}\, e\ \mathsf{type} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash e_1 \times e_1 :: \star}\,5}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash [\![\mathsf{Typ}\, e \times \mathsf{Typ}\, e]\!]\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash e_1 \times e_1 :: \star}\,22}}{\bullet, o : [\![T]\!], e : [\![T]\!], x : [\![\mathsf{Typ}\, e \times \mathsf{Typ}\, e]\!] \vdash \mathsf{ok} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1, x : e_1 \times e_1 \vdash \mathsf{ok}}\,2}}{\begin{array}{c}\bullet, o : [\![T]\!], e : [\![T]\!], x : [\![\mathsf{Typ}\, e \times \mathsf{Typ}\, e]\!] \vdash_{\mathrm{P}} o : [\![T]\!] \rightsquigarrow \\ \Gamma_0 \vdash \Lambda o_1 :: \star.\, \lambda \bar{o} : \mathsf{Ty} o_1.\, \Lambda e_1 :: \star.\, \lambda \bar{e} : \mathsf{Ty} e_1.\, \lambda x : e_1 \times e_1.\, \bar{o} : \forall o_1 :: \star.\, \mathsf{Ty}\, o_1 \to \forall e_1 :: \star.\, \mathsf{Ty}\, e_1 \to (e_1 \times e_1) \to \mathsf{Ty}\, o_1\end{array}}\,39}}{\bullet, o : [\![T]\!], e : [\![T]\!], x : [\![\mathsf{Typ}\, e \times \mathsf{Typ}\, e]\!] \vdash \mathsf{Typ}\, o\ \mathsf{type} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1, x : e_1 \times e_1 \vdash o_1 :: \star}\,3}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash \Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ}\, o\ \mathsf{type} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash (e_1 \times e_1) \to o_1 :: \star}\,4}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash [\![\Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ}\, o]\!]\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash (e_1 \times e_1) \to o_1 :: \star}\,22}}{\bullet, o : [\![T]\!], e : [\![T]\!], c : [\![\Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ}\, o]\!] \vdash \mathsf{ok} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1, \bar{c} : (e_1 \times e_1) \to o_1 \vdash \mathsf{ok}}\,2}}{\bullet, o : [\![T]\!], e : [\![T]\!], c : [\![\Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ}\, o]\!] \vdash 1\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1, \bar{c} : (e_1 \times e_1) \to o_1 \vdash 1 :: \star}\,20}}{\bullet, o : [\![T]\!], e : [\![T]\!] \vdash \Sigma c : [\![\Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ} o]\!].\ 1\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1, e_1 :: \star, \bar{e} : \mathsf{Ty}\, e_1 \vdash ((e_1 \times e_1) \to o_1) \times 1 :: \star}\,25}}{\bullet, o : [\![T]\!] \vdash \Sigma e : [\![T]\!].\, \Sigma c : [\![\Pi x : [\![\mathsf{Typ} e \times \mathsf{Typ} e]\!].\ \mathsf{Typ} o]\!].\, 1\ \mathsf{sig} \rightsquigarrow \Gamma_0, o_1 :: \star, \bar{o} : \mathsf{Ty}\, o_1 \vdash \lambda e_1 :: \star.\, \mathsf{Ty}\, e_1 \times ((e_1 \times e_1) \to o_1) \times 1 :: \star \Rightarrow \star}\,25$$

Figure 3: Translating the signature ORD in Fig. 2 into System F_ω

Table 6: Translating terms: $\Gamma \vdash e : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \bar{\tau}$

$$\frac{\Gamma \vdash e : \tau' \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \bar{\tau} \qquad \Gamma \vdash \tau' \equiv \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{\tau} :: \star}{\Gamma \vdash e : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \bar{\tau}}\,11$$

$$\frac{\Gamma \vdash_{\mathrm{k}} M : [\![\tau]\!] \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\, \Pi \bar{\Gamma}.\, \exists \vec{t_1} :: \hat{\kappa}_1.\, \bar{\tau}}{\Gamma \vdash \mathsf{Val}\, M : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathsf{case}\, \bar{M}\, \mathsf{of}\, \langle \vec{t_0}, x \rangle.\, \mathsf{case}\, x \bar{\Gamma}\, \mathsf{of}\, \langle \vec{t_1}, y \rangle.\, y : \bar{\tau}}\,12$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\, \Pi \bar{\Gamma}.\, \exists \vec{t_1} :: \hat{\kappa}_1.\, \bar{\sigma} \vec{\tau} \\ \Gamma,\ s : \sigma \vdash e : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \vec{s} : \hat{\sigma},\ \bar{s} : \bar{\sigma} \vec{s} \vdash \bar{e} : \bar{\tau} \qquad \vec{s}\ \text{does not appear free in}\ \bar{\tau}\end{array}}{\Gamma \vdash \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathsf{case}\, \bar{M}\, \mathsf{of}\, \langle \vec{t_0}, x \rangle.\, \mathsf{case}\, x \bar{\Gamma}\, \mathsf{of}\, \langle \vec{t_1}, \bar{s} \rangle.\, \bar{e}[\vec{\tau}/\vec{s}] : \bar{\tau}}\,13$$

$$\frac{\Gamma,\ f : [\![\Pi s : \sigma.\, \tau]\!],\ s : \sigma \vdash e : \tau \rightsquigarrow \Gamma_0, \bar{\Gamma},\ \bar{f} : \bar{\rho},\ \vec{s} :: \hat{\sigma},\ \bar{s} : \bar{\sigma} \vec{s} \vdash \bar{e} : \bar{\tau}}{\Gamma \vdash \mathsf{fix}\, f(s : \sigma) : \tau.\, e : \Pi s : \sigma.\, \tau \rightsquigarrow \Gamma_0 \vdash \mathsf{rec}(\bar{\rho})(\lambda \bar{f} : \bar{\rho}.\, \Lambda \vec{s} :: \hat{\sigma}.\, \lambda \bar{s} : \bar{\sigma} \vec{s}.\, \bar{e}) : \bar{\rho}}\,14$$

$$\frac{\Gamma \vdash e : \Pi s : \sigma.\, \tau \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \forall \vec{s} :: \hat{\sigma}.\, \bar{\sigma} \vec{s} \to \bar{\tau} \qquad \Gamma \vdash_{\mathrm{P}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \Pi \bar{\Gamma}.\, \bar{\sigma} \vec{\tau}}{\Gamma \vdash eM : \tau[M/s] \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} \vec{\tau} (\bar{M} \bar{\Gamma}) : \bar{\tau}[\vec{\tau}/\vec{s}]}\,15$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e}_1 : \bar{\tau}_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e}_2 : \bar{\tau}_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \langle \bar{e}_1, \bar{e}_2 \rangle : \bar{\tau}_1 \times \bar{\tau}_2}\,16$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \bar{\tau}_1 \times \bar{\tau}_2}{\Gamma \vdash \pi_1 e : \tau_1 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \pi_1 \bar{e} : \bar{\tau}_1}\,17 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \bar{e} : \bar{\tau}_1 \times \bar{\tau}_2}{\Gamma \vdash \pi_2 e : \tau_2 \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \pi_2 \bar{e} : \bar{\tau}_2}\,18$$

$$\frac{\Gamma \vdash_{\mathrm{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t_0} :: \hat{\kappa}_0.\, \Pi \bar{\Gamma}.\, \exists \vec{t_1} :: \hat{\kappa}_1.\, \bar{\sigma} \vec{\tau}}{\Gamma \vdash \mathsf{pack}\, M\, \mathsf{as}\, \langle\!|\sigma|\!\rangle : \langle\!|\sigma|\!\rangle \rightsquigarrow \Gamma_0, \bar{\Gamma} \vdash \mathsf{case}\, \bar{M}\, \mathsf{of}\, \langle \vec{t_0}, x \rangle.\, \mathsf{case}\, x \bar{\Gamma}\, \mathsf{of}\, \langle \vec{t_1}, \bar{s} \rangle.\, \langle \vec{t} = \vec{\tau}, \bar{s} : \bar{\sigma} \vec{t} \rangle : \exists \bar{\sigma}}\,19$$

It may seem that Rule 12 should require that no type variable in $\vec{t_0}$ or $\vec{t_1}$ appear free in $\bar{\tau}$, but that is guaranteed by Validity (Proposition B.23 in Dreyer et al.'s technical report (2002)). In Rule 14, the type $\bar{\rho}$ is always $\forall \vec{s} :: \hat{\sigma}.\, \bar{\sigma} \vec{s} \to \bar{\tau}$.

Table 7: Translating modules: $\Gamma \vdash_{\mathtt{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau}$

$$\frac{\Gamma \vdash \mathsf{ok} \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \mathsf{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s) \rightsquigarrow \Gamma_0 \vdash \Lambda \bar{\Gamma}.\ \bar{s} : \Pi \bar{\Gamma}.\ \bar{\Gamma}(\bar{s})}\ 39 \qquad \frac{\Gamma \vdash \mathsf{ok} \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \mathsf{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle : 1 \rightsquigarrow \Gamma_0 \vdash \Lambda \bar{\Gamma}.\ \langle\rangle : \Pi \bar{\Gamma}.\ 1}\ 40$$

$$\frac{\Gamma \vdash \tau\ \mathsf{type} \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \bar{\tau} :: \star}{\Gamma \vdash_{\mathtt{P}} [\tau] : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \Lambda \bar{\Gamma}.\ \mathsf{ty}\ \bar{\tau} : \Pi \bar{\Gamma}.\ \mathsf{Ty}\ \bar{\tau}}\ 41 \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \bar{e} : \bar{\tau}}{\Gamma \vdash_{\mathtt{P}} [e : \tau] : [\![\tau]\!] \rightsquigarrow \Gamma_0 \vdash \Lambda \bar{\Gamma}.\ \bar{e} : \Pi \bar{\Gamma}.\ \bar{\tau}}\ 42$$

$$\frac{\Gamma,\ s : \sigma \vdash_{\mathtt{k}} M : \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma} \vec{s} \to \bar{\rho} \vec{\tau} \qquad \mathtt{k} \sqsubseteq \mathtt{D}}{\Gamma \vdash_{\mathtt{k}} \lambda s{:}\sigma.\ M : \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ (\lambda \vec{t} :: \hat{\kappa} \Rightarrow \hat{\rho}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma} \vec{s} \to \bar{\rho}(\vec{t}\vec{s}))(\lambda \vec{s} :: \hat{\kappa}.\ \vec{\tau})}\ 43$$

$$\frac{\begin{array}{c}\Gamma,\ s : \sigma \vdash_{\mathtt{k}} M : \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma} \vec{s} \to \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\rho} \vec{\tau} \\ \Gamma \vdash_{\mathtt{k} \sqcap \mathtt{D}} \lambda s{:}\sigma.\ M : \Pi^{\mathsf{par}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \Lambda \vec{s} :: \hat{\kappa}.\ \lambda \bar{s} : \bar{\sigma} \vec{s}.\ \mathsf{case}\ x \bar{\Gamma} \vec{s} \bar{s}\ \mathsf{of}\ \langle \vec{t}_1, y \rangle.\ \langle \vec{t} = \vec{\tau}, y : \bar{\rho} \vec{t} \rangle \\ : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \forall \vec{s} :: \hat{\kappa}.\ \bar{\sigma} \vec{s} \to \exists \bar{\rho}\end{array}}\ 44$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{k}} M : \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ (\lambda \vec{t} :: \hat{\bar{\sigma}} \Rightarrow \hat{\rho}.\ \forall \vec{s} :: \hat{\bar{\sigma}}.\ \bar{\sigma} \vec{s} \to \bar{\rho}(\vec{t}\vec{s})) \vec{\tau} \\ \Gamma \vdash_{\mathtt{P}} N : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{N} : \Pi \bar{\Gamma}.\ \bar{\sigma} \vec{\tau}' \\ \Gamma \vdash_{\mathtt{k}} MN : \rho[N/s] \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, y \rangle.\ y \vec{\tau}' (\bar{N} \bar{\Gamma}) \\ : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ (\lambda \vec{t}' :: \hat{\bar{\sigma}}.\ \bar{\rho}[\vec{\tau}'/\vec{s}](\vec{\tau}\vec{t}'))\vec{\tau}'\end{array}}\ 45$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{k}} M : \Pi^{\mathsf{par}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \forall \vec{s} :: \hat{\bar{\sigma}}.\ \bar{\sigma} \vec{s} \to \exists \bar{\rho} \\ \Gamma \vdash_{\mathtt{P}} N : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{N} : \Pi \bar{\Gamma}.\ \bar{\sigma} \vec{\tau} \\ \Gamma \vdash_{\mathtt{k} \sqcup \mathtt{S}} MN : \rho[N/s] \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, y \rangle.\ y \vec{\tau} (\bar{N} \bar{\Gamma}) \\ : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \exists \vec{t}' :: \hat{\bar{\rho}}.\ \bar{\rho}[\vec{\tau}'/\vec{s}] \vec{t}'\end{array}}\ 46$$

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau} \qquad \Gamma,\ s : \sigma \vdash_{\mathtt{k}} N : \rho \rightsquigarrow \Gamma_0 \vdash \bar{N} : \exists \vec{t}'_0 :: \hat{\kappa}'_0.\ \Pi \bar{\Gamma}.\ \forall \vec{s} :: \hat{\bar{\sigma}}.\ \bar{\sigma} \vec{s} \to \exists \vec{t}'_1 :: \hat{\kappa}'_1.\ \bar{\rho} \vec{\tau}'}{\begin{array}{l}\Gamma \vdash_{\mathtt{k}} \langle s = M, N \rangle : \Sigma s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \mathsf{open}\ \bar{N}\ \mathsf{as}\ \langle \vec{t}'_0, y \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, z \rangle.\ \mathsf{open}\ y \bar{\Gamma} \vec{\tau} z\ \mathsf{as}\ \langle \vec{t}'_1, w \rangle.\ \langle z, w \rangle \\ \quad : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \exists \vec{t}'_0 :: \hat{\kappa}'_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \exists \vec{t}'_1 :: \hat{\kappa}'_1.\ (\lambda \vec{s} :: \hat{\bar{\sigma}}.\ \lambda \vec{t} :: \hat{\bar{\rho}}.\ \bar{\sigma} \vec{s} \times \bar{\rho} \vec{t}) \vec{\tau} \vec{\tau}' [\vec{\tau}/\vec{s}]\end{array}}\ 47$$

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \Sigma s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ (\lambda \vec{s} :: \hat{\bar{\sigma}}.\ \lambda \vec{t} :: \hat{\bar{\rho}}.\ \bar{\sigma} \vec{s} \times \bar{\rho} \vec{t}) \vec{\tau} \vec{\tau}'}{\Gamma \vdash_{\mathtt{k}} \pi_1 M : \sigma \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, y \rangle.\ \pi_1 y : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau}}\ 48$$

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \Sigma s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ (\lambda \vec{s} :: \hat{\bar{\sigma}}.\ \lambda \vec{t} :: \hat{\bar{\rho}}.\ \bar{\sigma} \vec{s} \times \bar{\rho} \vec{t}) \vec{\tau} \vec{\tau}'}{\Gamma \vdash_{\mathtt{k}} \pi_2 M : \sigma \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, y \rangle.\ \pi_2 y : \exists \vec{t}'_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}'_1 :: \hat{\kappa}_1.\ \bar{\rho}[\vec{\tau}/\vec{s}] \vec{\tau}'}\ 49$$

$$\frac{\Gamma \vdash e : \langle\!|\sigma|\!\rangle \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \bar{e} : \exists \bar{\sigma}}{\Gamma \vdash_{\mathtt{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma \rightsquigarrow \Gamma_0 \vdash \Lambda \bar{\Gamma}.\ \bar{e} : \Pi \bar{\Gamma}.\ \exists \bar{\sigma}}\ 50$$

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau}}{\begin{array}{l}\Gamma \vdash_{\mathtt{k} \sqcup \mathtt{D}} (M :: \sigma) : \sigma \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \langle \vec{t} = \lambda \bar{\Gamma}.\ \lambda \vec{t}_1 :: \hat{\kappa}_1.\ \vec{\tau}, x : \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma}(\vec{t} \bar{\Gamma} \vec{t}_1) \rangle \\ \quad : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \exists \vec{t}' :: \bar{\Gamma} \Rightarrow \hat{\kappa}_1 \Rightarrow \hat{\bar{\sigma}}.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma}(\vec{t} \bar{\Gamma} \vec{t}_1)\end{array}}\ 51$$

$$\frac{\Gamma \vdash_{\mathtt{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau}}{\begin{array}{l}\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, y \rangle.\ \langle \vec{t} = \vec{\tau}, y : \bar{\sigma} \vec{t} \rangle \\ \quad : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \exists \bar{\sigma}\end{array}}\ 52$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \bar{M} : \Pi \bar{\Gamma}.\ \mathsf{Ty}\ \bar{\tau}}{\Gamma \vdash_{\mathtt{P}} M : \mathfrak{S}(M) \rightsquigarrow \Gamma_0 \vdash \bar{M} : \Pi \bar{\Gamma}.\ \mathsf{Ty}\ \bar{\tau}}\ 53$$

$$\frac{\Gamma \vdash_{\mathtt{P}} \lambda s{:}\sigma.\ M s : \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \bar{\tau}}{\Gamma \vdash_{\mathtt{P}} M : \Pi^{\mathsf{tot}} s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \bar{\tau}}\ 54 \qquad \frac{\Gamma \vdash_{\mathtt{P}} \langle s = \pi_1 M, \pi_2 M \rangle : \Sigma s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \bar{\tau}}{\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma.\ \rho \rightsquigarrow \Gamma_0 \vdash \bar{M} : \bar{\tau}}\ 55$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{k}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma} \vec{\tau} \\ \Gamma,\ s : \sigma \vdash_{\mathtt{k}} N : \rho \rightsquigarrow \Gamma_0 \vdash \bar{N} : \exists \vec{t}'_0 :: \hat{\kappa}'_0.\ \Pi \bar{\Gamma}.\ \forall \vec{s} :: \hat{\bar{\sigma}}.\ \bar{\sigma} \vec{s} \to \exists \vec{t}'_1 :: \hat{\kappa}'_1.\ \bar{\rho} \vec{\tau}' \qquad \Gamma \vdash \rho\ \mathsf{sig} \rightsquigarrow \Gamma_0,\ \bar{\Gamma} \vdash \rho :: \kappa \\ \Gamma \vdash_{\mathtt{k}} \mathsf{let}\ s = M\ \mathsf{in}\ (N : \rho) : \rho \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \mathsf{open}\ \bar{N}\ \mathsf{as}\ \langle \vec{t}'_0, y \rangle.\ \Lambda \bar{\Gamma}.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, z \rangle.\ \mathsf{open}\ y \bar{\Gamma} \vec{\tau} z\ \mathsf{as}\ \langle \vec{t}'_1, w \rangle.\ w \\ : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \exists \vec{t}'_0 :: \hat{\kappa}'_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \exists \vec{t}'_1 :: \hat{\kappa}'_1.\ \bar{\rho} \vec{\tau}' [\vec{\tau}/\vec{s}]\end{array}}\ 56$$

$$\frac{\Gamma \vdash_{\mathtt{k}'} M : \sigma_1 \rightsquigarrow \Gamma_0 \vdash \bar{M} : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma}_1 \vec{\tau} \qquad \Gamma \vdash \sigma_1 \leq \sigma_2 \rightsquigarrow \Gamma_0,\ \bar{\Gamma},\ \vec{s} :: \hat{\bar{\sigma}}_1,\ \bar{s} : \bar{\sigma}_1 \vec{s} \vdash e : \bar{\sigma}_2 \vec{\tau}' \qquad \mathtt{k}' \sqsubseteq \mathtt{k}}{\Gamma \vdash_{\mathtt{k}} M : \sigma_2 \rightsquigarrow \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t}_0, x \rangle.\ \mathsf{open}\ x \bar{\Gamma}\ \mathsf{as}\ \langle \vec{t}_1, \bar{s} \rangle.\ e[\vec{\tau}/\vec{s}] : \exists \vec{t}_0 :: \hat{\kappa}_0.\ \Pi \bar{\Gamma}.\ \exists \vec{t}_1 :: \hat{\kappa}_1.\ \bar{\sigma}_2 \vec{\tau}' [\vec{\tau}/\vec{s}]}\ 57$$

In the conclusion of Rule 48, it may seem that $\bar{\sigma}$ should be $\bar{\sigma}[\vec{\tau}'/\vec{t}]$ instead, but the binding discipline of structure signatures (translated in Rule 25 in Table 5) guarantees that $\vec{t}$ does not appear free in $\bar{\sigma}$. In the conclusion of Rule 57, it may seem that $\bar{\sigma}_2$ should be $\bar{\sigma}_2[\vec{\tau}/\vec{s}]$ instead, but the subtyping premise guarantees that $\vec{s}$ does not appear free in $\bar{\sigma}_2$.

Table 8: Translating signature subtyping: $\Gamma \vdash \sigma_1 \le \sigma_2 \rightsquigarrow \Gamma_0,\ \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_1,\ \bar s : \bar\sigma_1 \vec s \vdash e : \bar\sigma_2 \vec\tau$

$$\frac{\Gamma \vdash \mathrm{ok} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \le 1 \rightsquigarrow \Gamma_0, \bar\Gamma,\ \bar s : 1 \vdash \bar s : 1}\ 32 \qquad \frac{\Gamma \vdash \mathrm{ok} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \le [\![T]\!] \rightsquigarrow \Gamma_0, \bar\Gamma,\ s_1 :: \star,\ \bar s : \mathsf{Ty}\, s_1 \vdash \bar s : \mathsf{Ty}\, s_1}\ 33$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\tau :: \star}{\Gamma \vdash [\![\tau_1]\!] \le [\![\tau_2]\!] \rightsquigarrow \Gamma_0, \bar\Gamma,\ \bar s : \bar\tau \vdash \bar s : \bar\tau}\ 34$$

$$\frac{\begin{array}{r}\Gamma \vdash \sigma_2 \le \sigma_1 \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_2,\ \bar s : \bar\sigma_2 \vec s \vdash e : \bar\sigma_1 \vec\tau \\ \Gamma,\ s : \sigma_2 \vdash \rho_1 \le \rho_2 \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_2,\ \bar s : \bar\sigma_2 \vec s,\ \vec t :: \hat{\bar\rho}_1,\ \bar t : \bar\rho_1 \vec t \vdash e' : \bar\rho_2 \vec\tau' \\ \Gamma,\ s : \sigma_1 \vdash \rho_1\ \mathrm{sig} \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_1,\ \bar s : \bar\sigma_1 \vec s \vdash \bar\rho_1 :: \kappa\end{array}}{\begin{array}{l}\Gamma \vdash \Pi^{\mathsf{par}} s{:}\sigma_1.\, \rho_1 \le \Pi^{\mathsf{par}} s{:}\sigma_2.\, \rho_2 \rightsquigarrow \\ \quad \Gamma_0, \bar\Gamma,\ x : \forall \vec s :: \hat{\bar\sigma}_1.\, \bar\sigma_1 \vec s \to \exists \bar\rho_1 \vdash \Lambda \vec s :: \hat{\bar\sigma}_2.\, \lambda \bar s{:}\bar\sigma_2 \vec s.\, \mathsf{case}\ x \vec\tau e\ \mathsf{of}\ \langle \vec t, y\rangle.\, \langle \vec t' = \vec\tau', e' : \bar\rho_2 \vec t'\rangle \\ \qquad : \forall \vec s :: \hat{\bar\sigma}_2.\, \bar\sigma_2 \vec s \to \exists \bar\rho_2\end{array}}\ 35$$

$$\begin{array}{l}\Gamma \vdash \Pi^{\mathsf{tot}} s{:}\sigma_1.\, \rho_1 \le \Pi^{\mathsf{tot}} s{:}\sigma_2.\, \rho_2 \rightsquigarrow \\ \quad \Gamma_0, \bar\Gamma,\ \vec a :: \hat{\bar\sigma}_1 \Rightarrow \hat{\bar\rho}_1,\ x : \forall \vec s :: \hat{\bar\sigma}_1.\, \bar\sigma_1 \vec s \to \bar\rho_1(\vec a \vec s) \vdash \Lambda \vec s :: \hat{\bar\sigma}_2.\, \lambda \bar s{:}\bar\sigma_2 \vec s.\, e'[x\vec\tau e/\bar t][\vec a \vec\tau/\vec t] \\ \qquad : (\lambda \vec t' :: \hat{\bar\sigma}_2 \Rightarrow \hat{\bar\rho}_2.\, \forall \vec s :: \hat{\bar\sigma}_2.\, \bar\sigma_2 \vec s \to \bar\rho_2(\vec t' \vec s))(\lambda \vec s :: \hat{\bar\sigma}_2.\, \vec\tau'[\vec a \vec\tau/\vec t])\end{array}$$

$$\begin{array}{l}\Gamma \vdash \Pi^{\mathsf{tot}} s{:}\sigma_1.\, \rho_1 \le \Pi^{\mathsf{par}} s{:}\sigma_2.\, \rho_2 \rightsquigarrow \\ \quad \Gamma_0, \bar\Gamma,\ \vec a :: \hat{\bar\sigma}_1 \Rightarrow \hat{\bar\rho}_1,\ x : \forall \vec s :: \hat{\bar\sigma}_1.\, \bar\sigma_1 \vec s \to \bar\rho_1(\vec a \vec s) \vdash \Lambda \vec s :: \hat{\bar\sigma}_2.\, \lambda \bar s{:}\bar\sigma_2 \vec s.\, \langle \vec t' = \vec\tau'[\vec a \vec\tau/\vec t], e'[x\vec\tau e/\bar t][\vec a \vec\tau/\vec t] : \bar\rho_2 \vec t'\rangle \\ \qquad : \forall \vec s :: \hat{\bar\sigma}_2.\, \bar\sigma_2 \vec s \to \exists \bar\rho_2\end{array}$$

$$\frac{\begin{array}{r}\Gamma \vdash \sigma_1 \le \sigma_2 \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_1,\ \bar s : \bar\sigma_1 \vec s \vdash e : \bar\sigma_2 \vec\tau \\ \Gamma,\ s : \sigma_1 \vdash \rho_1 \le \rho_2 \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_1,\ \bar s : \bar\sigma_1 \vec s,\ \vec t :: \hat{\bar\rho}_1,\ \bar t : \bar\rho_1 \vec t \vdash e' : \bar\rho_2 \vec\tau' \\ \Gamma,\ s : \sigma_2 \vdash \rho_2\ \mathrm{sig} \rightsquigarrow \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_2,\ \bar s : \bar\sigma_2 \vec s \vdash \bar\rho_2 :: \kappa\end{array}}{\begin{array}{l}\Gamma \vdash \Sigma s{:}\sigma_1.\, \rho_1 \le \Sigma s{:}\sigma_2.\, \rho_2 \rightsquigarrow \\ \quad \Gamma_0, \bar\Gamma,\ \vec s :: \hat{\bar\sigma}_1,\ \vec t :: \hat{\bar\rho}_1,\ x : \bar\sigma_1 \vec s \times \bar\rho_1 \vec t \vdash \langle e, e'[\pi_2 x/\bar t]\rangle[\pi_1 x/\bar s] \\ \qquad : (\lambda \vec s' :: \hat{\bar\sigma}_2.\, \lambda \vec t' :: \hat{\bar\rho}_2.\, \bar\sigma_2 \vec s' \times \bar\rho_2 \vec t')\vec\tau\vec\tau'\end{array}}\ 36$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M : [\![T]\!] \rightsquigarrow \Gamma_0 \vdash \bar M : \Pi\bar\Gamma.\, \mathsf{Ty}\, \bar\tau}{\Gamma \vdash \mathfrak{S}(M) \le [\![T]\!] \rightsquigarrow \Gamma_0, \bar\Gamma,\ \bar s : \mathsf{Ty}\, \bar\tau \vdash \bar s : \mathsf{Ty}\, \bar\tau}\ 37 \qquad \frac{\Gamma \vdash M \cong N : [\![T]\!] \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \mathsf{Ty}\, \bar\tau :: \star}{\Gamma \vdash \mathfrak{S}(M) \le \mathfrak{S}(N) \rightsquigarrow \Gamma_0, \bar\Gamma,\ \bar s : \mathsf{Ty}\, \bar\tau \vdash \bar s : \mathsf{Ty}\, \bar\tau}\ 38$$

Rule 35 is really three rules with the same premises but different conclusions.

Table 9: Translating equivalences

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1\ \mathrm{type} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\tau :: \star \\ \Gamma \vdash \tau_2\ \mathrm{type} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\tau :: \star\end{array}}{\Gamma \vdash \tau_1 \equiv \tau_2 \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\tau :: \star}\ 7\text{–}10 \qquad \frac{\begin{array}{l}\Gamma \vdash \sigma_1\ \mathrm{sig} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\sigma :: \kappa \\ \Gamma \vdash \sigma_2\ \mathrm{sig} \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\sigma :: \kappa\end{array}}{\Gamma \vdash \sigma_1 \equiv \sigma_2 \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\sigma :: \kappa}\ 26\text{–}31 \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{P}} M : \sigma \rightsquigarrow \Gamma_0 \vdash \bar M : \Pi\bar\Gamma.\, \bar\sigma\vec\tau \\ \Gamma \vdash_{\mathrm{P}} N : \sigma \rightsquigarrow \Gamma_0 \vdash \bar N : \Pi\bar\Gamma.\, \bar\sigma\vec\tau\end{array}}{\Gamma \vdash M \cong N : \sigma \rightsquigarrow \Gamma_0, \bar\Gamma \vdash \bar\sigma\vec\tau :: \star}\ 58\text{–}73$$