A Focused Solution to the Avoidance Problem

Karl Crary

Carnegie Mellon University

Abstract

In ML-style module type theory, sealing often leads to situations in which type variables must leave scope, and this creates a need for signatures that avoid such variables. Unfortunately, in general there is no best signature that avoids a variable, so modules do not always enjoy principal signatures. This observation is called the *avoidance problem.* In the past, the problem has been circumvented using a variety of devices for moving variables so they can remain in scope. These devices work, but have heretofore lacked a logical foundation. They have also lacked a presentation in which the dynamic semantics is given on the same phrases as the static semantics, which limits their applications.

We can provide a best supersignature avoiding a variable by fiat, by adding an existential signature that is the least upper bound of its instances. This idea is old, but a workable metatheory has not previously been worked out. This work resolves the metatheoretic issues using ideas borrowed from focused logic.

We show that the new theory results in a type discipline very similar to the aforementioned devices used in prior work. In passing, this gives a type-theoretic justification for the generative stamps used in the early days of the static semantics of ML modules. All the proofs are formalized in Coq.

1 Introduction

One of the most famous contributions of Bob Harper's intellectual career has been ML-style modules. With Kevin Mitchell, he was the first to implement the Standard ML module system. He co-authored the type discipline for SML modules [19] that developed into the Definition of Standard ML's stamp-based semantics [31]. And, with various coauthors, he wrote a famous series of papers developing the type-theoretic treatment of ML modules [20, 21, 18, 22, 46, 10, 24].

Harper's thesis was that the static semantics of ML modules could and should be explained by type theory. He argued that type theory—together with proof theory and category theory—provides the most robust framework for reasoning about programming. His thesis was ultimately borne out, at least in part, when, with Chris Stone, he was able to prove the type safety of Standard ML using the type-theoretic definition [22], but the Definition's stamp-based semantics turned out to be unsuitable for a type safety proof [49]. Later (with Daniel Lee) we used his type-theoretic methodology to produce a fully machine-verified proof of type safety for Standard ML [24, 8].

The mid-to-late 1990s and early 2000s were a fruitful time for module-oriented type theory. Restricting our attention only to those developments germane to our present discussion: We saw the development of translucent sums (a.k.a. manifest types) [18, 25, 30, 27], which justify putting type definitions into ML signatures. That notion was soon incorporated in the 1997 revision of the Standard ML definition [32]. Singleton kinds regularized translucency as a conventional kind discipline [46, 45, 3]. We saw principal signatures for higher-order applicative modules [26], and fully syntactic signatures [42], meaning that every module phrase should enjoy a principal signature expressible in the type theory. And we saw applicative functors that could be applied to arguments not in named form [39].

What was lacking was a single type theory that could harmonize all those developments. In 2001, Harper, Derek Dreyer, and I thought we had found that type theory.

The key idea was that sealing a module (*i.e.,* making it accessible only through a specified signature) constitutes a form of computational effect. The notion is that whenever a module is sealed, an adversary might come in and meddle with the module's internals, including its type components, making it impossible for the module's client to rely on its internals.

By viewing sealing as an effect, we were able to harmonize applicative and generative functors in a single setting: applicative functors are total (*i.e.,* applying them generates no effect) while generative functors are partial (*i.e.,* applying them might generate an effect). Also, we were able to use a static-semantic condition (purity), as opposed to a syntactic condition (*e.g.,* values or paths) to limit which modules could have types projected from them.

The avoidance problem The identification of sealing as an effect worked well, and it was eventually published in 2003 [10]. But there was another issue, one that was first identified by Harper and Lillibridge during their translucent sums work, and discussed in Lillibridge's PhD thesis [30]. (A similar issue arises in existential types with bounded quantification [13].) Sometimes a type variable leaves scope. For example, consider the SML code:

```
let
  datatype t = C
in
  struct val x = C end
end
```

What is the type of this program? Arguably it ought to be rejected: since SML datatype declarations are generative, there is no way to refer to `t` outside the let, so the program has no well-scoped type.

But SML does not reject the program. Why not? Alas, the situation is not always so simple. Consider:

```
let
  structure I = ...
    :> sig  type t = int  val x : t  end
in
  struct val y = I.x end
end
```

The signature of the body is `sig val y : I.t end`, but `I` is leaving scope, so that cannot be the type of the let. Since the definition of `I` is sealed, `I` cannot be substituted away. However, since `I.t` is specified as equal to `int`, we can give the let the signature `sig val y : int end`.

This example illustrates that we cannot simply reject a program because a phrase's type/signature mentions a variable that is leaving scope. We must look for an alternative type/signature that does *not* mention that variable.

Without subtyping, all alternatives are equivalent, so one alternative is just as good as another. However, the type theory of modules relies on a supersignature relation, and that means we must find not just any supersignature, but the *best possible* supersignature. For example, given the code:

```
let
  type t = int
in
  struct  type u = t  val x = 12  end
end
```

it would not be good to infer the signature `sig type u val x : u end`. We must infer the better signature `sig type u = int val x : u end`.

The *avoidance problem* is that, in general, there is no best supersignature that avoids mentioning a variable. For example, consider the program:

```
structure M =
  let
    datatype t = C
  in
    ...
    :>
    sig
      type 'a u = t
      datatype v = D of t
      val x : int u
      val y : bool u
    end
  end
```

There are infinitely many incomparable signatures that can be assigned to `M`, but there exists no principal signature. One signature is:

```
sig
  type 'a u
  datatype v = D of int u
  val x : int u
  val y : bool u
end
```

Given this signature, `M.D M.x` type checks and `M.D M.y` does not. However, we could just as well give `D` the type `bool u -> v`, in which case the opposite would happen. Or we could give it the type `string u -> v`, in which case neither would type check. Without mentioning `t`, there is no way to make both type check.

Existential signatures Standard ML resolves this problem by allowing signatures to refer to type variables that are no longer in scope. (OCaml takes a different approach, which I discuss in Appendix A.) For example, in the final program above, Standard ML of New Jersey reports:

```
structure M :
  sig
    type 'a u = ?.M.t
    datatype v = D of ?.M.t
    val x : int u
    val y : bool u
  end
```

Standard ML justifies this using the notion of a generative stamp (the Definition [31] calls them "type names"), a dynamically allocated type identifier that outlives the scope in which it is defined. The odd type `?.M.t` is Standard ML of New Jersey's way to refer to a type stamp that is not accessible through any identifier that remains in scope.

Generative stamps resolve the avoidance problem—if no type ever becomes inaccessible, there is nothing to avoid—but they are quite dissatisfying from a type-theoretic perspective. Furthermore, since some modules must be given signatures that refer to internal, dynamically allocated state, it can be difficult to break programs up into units that can be compiled separately.

In order to put the generative stamps on a sounder type-theoretic footing, Russo [39] invented *existential signatures*. He would give `M` a signature something like:

```
∃t.
  sig
    type 'a u = t
    datatype v = D of t
    val x : int u
    val y : bool u
  end
```

This says that, for some unknown `t`, `M` has the signature given by the existential's body.

Existential signatures were a big improvement over stamps in that signatures now had independent meaning, without reference to dynamically allocated state. Nevertheless, Russo showed existential signatures to be equivalent to the generative stamp mechanism. An existential signature noted that a new type had been created, and each typing rule contained machinery to float the existential bindings upward.

The floating machinery was unusual, and, at the time, did not seem to emerge organically from a conventional type theory (see Section 2), so even though it was adopted by other successful following work, there remained an interest in resolving the avoidance problem using a more conventional type theory. As it turns out, the floating machinery *does* emerge organically from a conventional *focused* type theory. The signature-synthesis algorithm developed and

proven correct here is strikingly similar to Russo's static semantics, so this work can be seen as the type-theoretic vindication of Russo's floating machinery, and, by implication, of the Definition's generative stamps. But that is getting ahead of the story.

Harper and Stone [22] proposed an alternative but related solution. In their elaborative semantics, they would rewrite the program so that type variables never become inaccessible to the type theory. Structure `M` would be rewritten to something like:

```
structure M =
  struct
    structure HIDDEN =
      struct
        datatype t = C
      end

    structure VISIBLE =
      ...
      :>
      sig
        type 'a u = HIDDEN.t
        datatype v = D of HIDDEN.t
        val x : int u
        val y : bool u
      end
  end
```

The elaborator would ensure that any mention of `M` automatically became `M.VISIBLE`, so from the programmer's perspective, `M` seems to contain exactly what the programmer expects and the hidden field is inaccessible. However, from the type theory's perspective, `t` is still accessible (as `M.HIDDEN.t`) so the avoidance problem never arises.

The signature of the elaborated code is:

```
sig
  structure HIDDEN : sig datatype t = C end
  structure VISIBLE :
    sig
      type 'a u = HIDDEN.t
      datatype v = D of HIDDEN.t
      val x : int u
      val y : bool u
    end
end
```

This is very much like Russo's signature, with the whole `HIDDEN/VISIBLE` mechanism playing the role of an existential signature.

Indeed, Harper, Dreyer, and I [10] eventually employed an elaborator much like Harper and Stone's, except we replaced the `HIDDEN/VISIBLE` notation with an existential quantifier. Unlike Russo's existential, ours had no special role in the type theory. The type theory treated it as an ordinary dependent sum, but the elaborator recognized it and knew to look only in its right-hand component for components visible to the programmer.

The one that got away But in 2001, we thought we had a better idea. The idea was to patch the type system so that, by fiat, every signature does have a least supersignature not mentioning any particular variable. We added a signature $\exists m{:}\sigma_1.\sigma_2$ which can be thought of as the union of σ_2 over all choices of $m{:}\sigma_1$. We wrote the new connective using an existential quantifier, since it plays a similar role to Russo's existential, but it can also be thought of as an indexed union type.

The defining property of our existential was how it interacted with subtyping:

$$\frac{\Gamma, m{:}\sigma_1 \vdash \sigma_2 \leq \sigma \quad \Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash \exists m{:}\sigma_1.\sigma_2 \leq \sigma}$$

$$\frac{\Gamma \vdash M : \sigma_1 \quad \Gamma \vdash \sigma \leq [M/m]\sigma_2 \quad \Gamma, m{:}\sigma_1 \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \sigma \leq \exists m{:}\sigma_1.\sigma_2}$$

The witness module M can be a variable, so the latter rule allows us to derive $m{:}\sigma_1 \vdash \sigma_2 \leq \exists m{:}\sigma_1.\sigma_2$. Consequently, we get the following rule for let as a derived rule:

$$\frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma, m{:}\sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ m = M_1\ \mathsf{in}\ M_2 : \exists m{:}\sigma_1.\sigma_2}$$

There were two delicate issues arising from our existential signatures, one minor and one major. The minor issue was the second subtyping rule creates a problem for a subtyping algorithm. We had a solution to this: a simple syntactic restriction that prevents problematic subtyping queries from arising. (More on this in Section 6.3.)

The major issue was the need to extend the metatheory of the singleton-kind type theory [46, 4] to account for existentials. This is difficult, for reasons that go beyond the scope of this paper. However, we thought we had a strategy to side-step the issue.

The above-mentioned syntactic restriction also sufficed to ensure that singletons never arise in an equivalence-checking position. That is, we never need to ask whether $M \approx M' : \exists m{:}\sigma_1.\sigma_2$. We *did* need to deal with existentials in a binding position, such as $m{:}(\exists m'{:}\sigma_1.\sigma_2) \vdash M \approx M' : \sigma$. That problem we intended to deal with using the meta-rule (for arbitrary judgement $\mathcal{J}$):

$$\frac{\Gamma, m'{:}\sigma_1, m : \sigma_2 \vdash \mathcal{J}}{\Gamma, m{:}(\exists m'{:}\sigma_1.\sigma_2) \vdash \mathcal{J}}\ \text{(wrong)}$$

This proposed rule, motivated by a left-rule from sequent calculus, would ensure that the singleton-kind metatheory would never need to see an existential signature, and therefore would not need to deal with them.

Unfortunately, the rule is fatally flawed, because it invalidates the substitution principle. Suppose $\Gamma \vdash M : \exists m'{:}\sigma_1.\sigma_2$ and $\Gamma, m{:}(\exists m'{:}\sigma_1.\sigma_2) \vdash \mathcal{J}$. The substitution principle should give us $\Gamma \vdash [M/m]\mathcal{J}$. To push the induction through the rule above, we need to obtain $M' : \sigma_1$ such that $M : [M'/m']\sigma_2$. But if M is a variable, it may be that no such M' exists.

We attempted to repair the idea by introducing a choice operator (inspired by higher-order logic's choice operator), which would obtain the σ_1 from an $\exists m{:}\sigma_1.\sigma_2$, which would serve as such an M' for substitution. But we were never able to make the choice operator's metatheory work, and, in retrospect, this current work explains why we should never have expected it to work.

Ultimately we adopted an elaborative solution based on the Harper and Stone's strategy. We retained the existential notation but changed its meaning. As far as the type theory was concerned, $\exists m{:}\sigma_1.\sigma_2$ meant exactly the same thing as the module type theory's strong sum $\Sigma m{:}\sigma_1.\sigma_2$. However,

the elaborator, when locating a field in a module, would automatically project the second component from any existential, thereby making the first component inaccessible to the programmer.

This solved the practical problem, circumventing the avoidance problem by keeping any needed variables accessible to the type theory but inaccessible to the programmer. However, the elaborative solution seemed inelegant and hindered abstraction, leaving us wistful for the all-type-theory solution we almost had.

The remainder of the paper is organized as follows. In Section 2 I discuss the philosophical and practical requirements for a "type-theoretic solution." In Section 3 I review focused logic and see the key insight that motivates this work. In Section 4 I review the module type theory I build on. I give the new elements of the type theory in Section 5 and discuss type checking in Section 6. In Section 7 I compare my system carefully with Russo's, and draw some conclusions about Harper's original generative-stamps system. I briefly discuss the Coq formalization in Section 8 and then conclude in Section 9.

2 Type theory and semantic objects

What constitutes an "all-type theory" solution? The version of the work we published in 2003 dealt with the avoidance problem and used types. Why is that not "all-type-theory"?

Indeed, the 2003 paper was but one in a long line that dealt with the avoidance problem in a similar fashion. The Definition maintained access to types leaving scope using generative stamps with global scope. Russo [39] placed the technique on a type-oriented footing using existential types. (Later [40], he also used existential types to explain first-class modules.) The key device was a non-standard rule allowing existential types to float upward, thereby simulating the Definition's global scope. (Much more on this in Section 7.) All the work that followed (including this work), although differing in many important ways, did essentially the same thing when it came to the avoidance problem, albeit presented in different ways.

Harper and Stone [22], the 2003 paper, and Dreyer [9] elaborated modules in such a way that variables were kept available in strong sums even when they became invisible to the programmer. Rossberg *et al.* [38] took elaboration even further, eschewing a module type theory and instead elaborating the entire module system into an F_ω-like target language. Rossberg [37] later added first-class modules. Unlike Stone and Harper, *etc.,* they returned to Russo's existential types, instead of strong sums, but they reorganized so that Russo's floating machinery appears in only two rules.

All of these (as well as the current work) produce a language that is similar as it pertains to avoidance. All of them use types. So why is another take on the avoidance problem still needed?

We can coarsely taxonomize module calculi into two approaches: One is to define a language using whatever data structures (called *semantic objects*) are necessary to get the desired behavior. This approach goes back to Harper [19] and its best known instance is the Definition [31]. The other is to provide a type theory capable of faithfully expressing fundamental elements of modules. In practice, such a type theory cannot account for *all* the elements of modules, so some elaborator invariably is required. This approach goes back to Harper and Mitchell [20].

The question of what distinguishes type theory from semantic objects is philosophical and also practical. The connections between programming languages, logic, and category theory are well-known. Harper refers to these connections as *Computational Trinitarianism* [16]: a single sublime entity with three distinct aspects: types (programming), propositions (communication of ideas), and structures (mathematical constructions). Nothing can be said to be understood until it can be understood through multiple lenses. Thus, no type theory can be canonical until it can be explained in a reasonable logic, or in category theory.[1] The type theory we develop here has a strong connection to focused logic (Section 3), although we will not spell out a Curry-Howard correspondence in detail.

On the practical side, we require that a type theory enjoy a dynamic semantics defined on the same phrases as the static semantics. We are particularly interested in a structured operational semantics [35]. This allows one to use the techniques of type theory without additional complication. There are significant advantages to this: One can define contextual equivalence (and other dynamic equivalences) directly on source programs. Dynamic equivalences are a prerequisite to formal reasoning about the behavior of programs, or about the correctness of compilers [7]. One can also prove a Reynolds-style abstraction theorem [5], using which one can rigorously establish representation-independence results that are one of the main reasons to use modules in the first place. (Adapting those results to the theory developed here is important future work.)

From this viewpoint, Biswas [2], Russo [39], and Russo [40] employ the semantic object approach. So do Rossberg, *et al.* [38] and Rossberg [37], with the valuable innovation that all their "semantic objects" are actually well-typed syntactic objects expressible in F_ω. Thus, they are not merely data structures for a type checker; they provide a dynamic interpretation which establishes type safety and gives a strategy for compilation.

On the other hand, Harper and Stone [22], the 2003 paper, and Dreyer [9] employ type theories, but none of those type theories deal with the avoidance problem except through elaboration.

It may be observed that even with the current work we still do not have an all-type-theory account of modules. Some features, such as named fields, permutation, and width subtyping, are probably not hard to accommodate. Others, notably `open` declarations, are type-theoretically nonsense and will probably always be elaborated.

This work even takes a step backward in one area. In previous work we used effect inference to account for sealing effects, but here—as we will see—we will use a monad instead. This requires that modules using generativity/sealing be written in monadic form. I would argue that focused logic shows that the monad is the "right" way to deal with sealing and generativity. Still, Standard ML works differently, so we must elaborate something that we needn't before. Nevertheless, we clearly come out ahead: Elaborating to monadic form is harmless from an abstraction perspective, while elaborating hidden variables back into scope clearly breaks abstraction.

[1] One wag refers to this program as "type-splaining," a description I cheerfully accept.

3 Focusing

Focused logic is a restricted presentation of sequent calculus. It was first invented by Andreoli [1] as a proof search strategy that employed the notion of *focus* to cut down an otherwise unmanageably large search space. Andreoli applied the technique to linear logic, but it is not limited to that.

The connectives are divided into negative and positive [14], and asynchronous rules (*i.e.,* left rules for positive connectives, right for negative) are employed first. Synchronous rules (*i.e.,* left rules for negative connectives, right for positive) can then be employed, but only after taking focus. Once focus has been taken, only the type under focus can be decomposed, and focus remains on the type(s) into which it is decomposed, until proof search reaches a type of the wrong polarity to remain under focus (*i.e.,* positive on the left, negative on the right). This dramatically cuts back the search space, because the order in which asynchronous rules are applied is immaterial, and once focus is obtained, only a small number of choices are applicable at each step. For readers new to focused logic, a good introduction is Simmons [44].

Years later there was an explosion of interest in focused logic as a variety of new applications were discovered. One important application is for type theories that give control over order-of-evaluation [52]. Another application by Bob Harper and others [29, 41, 43], which directly influenced this work, is to use focusing as a framework to reconcile negative and positive connectives in a logical framework. (The original logical framework [17] supported only negative connectives.) Interestingly, both applications were to some extent anticipated by earlier work, the former by call-by-push-value [28], the latter by Concurrent LF [50, 51], but the focused presentation generalized the earlier work and argued for its canonicity.

The same could be said about the current work: We take the long-standing stamp/existential approach and argue for its canonicity by placing it on a logical foundation.

Strong and weak sums One way to characterize negative and positive connectives is this: Negative connectives are defined by their elimination form (*i.e.,* how they are used), while positive connectives are defined by their introduction form (*i.e.,* how they are constructed). For example, the defining property of a function is that it can be applied to an argument, so the arrow type is negative. On the other hand, the defining property of a disjoint sum is that it *is* either one thing or another, so the sum type is positive.

As a consequence, negative types have clean, open-scope elimination forms (*e.g.,* $M\,N$ or $\pi_1 M$), while positive types have closed-scope elimination forms that pattern-match on the intro (*e.g.,* $\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2$).

One of the two main signature connectives is the strong sum, written in Dreyer *et al.* as $\Sigma m{:}\sigma_1.\sigma_2$. This is a dependent form of product, used to implement the signature of a structure. Thus, the signature:

```
sig
  type t
  val x : t
end
```

is represented something like $\Sigma t{:}\mathsf{Type}.t$. (We need to develop some formalities before we can give its precise representation.)

What makes the sum strong is that its components can be obtained by projection. (Within the type theory we will use π_1 or π_2 instead of the named form—`M.t` or `M.x`—used in ML's external syntax.) Thus, according to the previous discussion, we expect the strong sum to be a negative type. Despite some complications, we can devise a focused logic that realizes that intuition [6].

In contrast, the components of a weak sum are obtained by opening the sum for a particular scope. (This form of sum is also commonly called an existential type [33].) Thus, the weak sum is a positive type.

The key observation that motivates this work is that existential signatures are weak sums, and are therefore positive, not negative. They are defined by how they are constructed: M belongs to $\exists m'{:}\sigma_1.\sigma_2$ exactly when there exists $M' : \sigma_1$ such that $M : [M'/m']\sigma_2$. They are *not* defined by how they are used—in fact, our goal is to use them silently.

In this paper, we carry this observation through to a workable type theory of existential signatures that solves the avoidance problem. Fortunately, we will not require the full syntax of focused logic, which is quite baroque. We only really need three mechanisms: First, we will sort signatures into negative and positive. Second, a lax modality (a.k.a. monad) will encapsulate sealing. Third, the monad's bind operator will be given a typing rule taken directly from the left-inversion rules of focused logic.

None of these mechanisms are precisely new. Each of them, or something similar, appears in prior work on modules: implicitly in Biswas [2], and more explicitly in Russo [39] and later Rossberg, *et al.* [38], among others. The novelty here is that all of these mechanisms work precisely as dictated by focused logic, and the resulting language enjoys a structured operational semantics given directly on the syntax.

Looking back, it is striking how close we came in 2001. The subsignature rules governing existential signatures are used essentially verbatim (although I write them here in a more modern notation). The incorrect existential-splitting rule above is very similar to the left-inversion rule for existentials we use here. And the syntactic restriction to avoid problematic subsignature queries works just as we intended.

As an aside, we can also see now why the choice-operator experiment was doomed: Existential signatures are positive, but the notion of projecting out the first component makes sense only if they are negative.

4 Preliminaries

We will begin with a pure module calculus that does not support sealing or generativity. In the absence of sealing/generativity, all modules are transparent, so the avoidance problem does not arise. We will reintroduce sealing in Section 5.

The pure module calculus is not a contribution of this work. It is a fragment of the system in Crary [5], which in turn is adapted from Dreyer [9]. It omits generative functors, sealing, module bindings at the module level, and existential-type unpacking at the module level. It also omits effect inference in the static semantics, as that will be replaced by a monad.

The module calculus, even after we extend it with sealing, is not intended to account for all the functionality of ML modules. We still assume that an elaborator takes care of

such features as named fields and `open` declarations. I conjecture that named fields could be added without too much difficulty. However, `open` declarations, despite being useful, are type-theoretically nonsense (the language's very binding structure becomes dependent on the static semantics) and will probably always remain the province of elaboration.[2]

The module calculus can be broken into the static expressions: kinds (k), type constructors (c and τ, we will use τ for constructors that are intended to be types), and signatures (σ)—and the dynamic expressions: terms (e) and modules (M). Signatures serve as the types for modules. The calculus is designed to respect the *phase distinction* [21], meaning that the meaning and equivalence of static expressions can be determined without executing any dynamic expressions. The syntax (minus sealing) is given in Figure 1.

Constructors and kinds The constructor and kind portion is the singleton kind calculus of Stone and Harper [46], supplemented with type operators. The type constructors are all standard. The kind Type contains constructors that happen to be types. We use the metavariable τ for constructors that are intended to be types. The kind 1 contains the unit constructor $\star$.

The *singleton kind* [46], $\mathsf{S}(c)$, classifies the constructors that are definitionally equivalent to the constructor c. It is used to model type definitions and type sharing specifications. Singletons create dependencies of kinds on constructors, so function and product kinds take dependent form, $\Pi\alpha{:}k_1.k_2$ and $\Sigma\alpha{:}k_1.k_2$, respectively. As usual, when α does not appear free in k_2, we sometimes write $\Pi\alpha{:}k_1.k_2$ as $k_1 \to k_2$, and $\Sigma\alpha{:}k_1.k_2$ as $k_1 \times k_2$.

Constructor equivalence is induced by beta and extensionality rules, together with rules pertaining to singleton kinds: Singleton introduction says c belongs to $\mathsf{S}(c)$, provided $c : \mathsf{Type}$. (Singletons for higher kinds are definable using primitive singletons and dependent kinds [46].) The singleton elimination rule conversely says $c : \mathsf{S}(c')$ implies $c \equiv c' : \mathsf{Type}$. Consequently, constructor equivalence is context-sensitive. For example, we have $\alpha{:}\mathsf{S}(\mathsf{int}) \vdash \alpha \equiv \mathsf{int} : \mathsf{Type}$. Constructor equivalence also depends on the kind at which constructors are compared. For example, $\lambda\alpha{:}\mathsf{Type}.\alpha$ and $\lambda\alpha{:}\mathsf{Type}.\mathsf{int}$ are non-equivalent at $\mathsf{Type} \to \mathsf{Type}$, but are equivalent at the superkind $\mathsf{S}(\mathsf{int}) \to \mathsf{Type}$.

Terms and modules The syntax for terms is largely standard. Following Crary [7], the type system imposes a value restriction on the body of polymorphic functions by requiring they have the form $\Lambda\alpha{:}k.sv$, where sv is a class of values. The value restriction does not interact with the avoidance problem, so we will not discuss it further. The recursion form $\mathsf{fix}_\tau\, e$ has the type τ, provided e has the type $(\mathsf{unit} \to \tau) \to \tau$, and we evaluate $\mathsf{fix}_\tau\, e$ to $e(\lambda_{:}\mathsf{unit}.\, \mathsf{fix}_\tau\, e)$.[3]

The module language contains static atomic modules ($(\!|\, c \,|\!)$), which contain a single constructor; dynamic atomic modules ($\langle\!|\, e \,|\!\rangle$), which contain a single term; functors; pairs; and let bindings of terms. For simplicity, pairs are non-dependent (but it would easy to add dependent pairs). For example, the ML module:

[2] As it happens, the same elaborative device—identifier resolution—used to deal with `open` can also neatly address named fields [22], but it need not.

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

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

Figure 1: Syntax (minus sealing)

```
struct
  type t = int
  val x = 12
end
```

could be elaborated $\langle (\!| \mathsf{int} |\!), (\!| 12 |\!) \rangle$.[4]

Extraction and twinned variables To extract a term from a dynamic atom, one uses the form $\mathsf{Ext}\, M$. However, to extract a type constructor from a static atom, we employ a judgement rather than a syntactic form. The judgement $\mathsf{Fst}(M) \gg c$ says that c is the *static portion* of M.[5] In particular, $\mathsf{Fst}((\!| c |\!)) \gg c$.

The reason for this design, invented by Dreyer [9], is it removes any syntactic dependency of static phrases (such as types) on terms or modules. By disentangling the singleton kind calculus from all dynamic phrases, we are able to use Stone and Harper's singleton-kind metatheory [46] off the shelf.

For static extraction to work compositionally, we must be able to compute the static portion of any module, not only static-atom literals. The rules for doing so are given in Figure 2. A subtle point arises: For any module variable m, we must be able to compute m's static portion, but of course that cannot be known, so we must have a type-constructor variable prepared to stand in for m's static portion. This leads to the concept of *twinned variables* [24]: every module variable m is twinned with a constructor variable α, written α/m, where α is the static portion of m.[6] However, when α is not used, I will often leave it out. Twinned variables are recorded in the context and static portions depend on them, so the static portion judgement is written $\Gamma \vdash M \gg c$.

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

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

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

[4] plus some sort of metadata supporting field resolution [22].

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

[6] An alternative [21, 9] is to use a naming convention to associate module and constructor variables, but this complicates binding and substitution. Sorting out those complications carefully leaves you with something very close to twinned variables.

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

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

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

$$
\frac{}{\Gamma \vdash \mathsf{Fst}(\{\!| e |\!\}) \gg \star}
$$

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

$$
\frac{\Gamma \vdash \mathsf{Fst}(M_1) \gg c_1 \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\begin{array}{c}\Gamma \vdash \mathsf{Fst}(M_1\, M_2) \gg c_1 c_2 \\ \Gamma \vdash \mathsf{Fst}(\langle M_1, M_2 \rangle) \gg \langle c_1, c_2 \rangle\end{array}}
$$

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

Figure 2: Static portions

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

Semantics The static semantics is given by several judgements, summarized in Figure 4. The full rules are given in Appendix B.

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

5 Positive Signatures and Lax Modules

The signatures we considered in the pure calculus are negative. We now add positive signatures as a new syntactic class:

$$
\begin{array}{lcl}
\text{negative signatures} & \sigma & ::= \quad \cdots \mid \bigcirc\rho \\
\text{positive signatures} & \rho & ::= \quad \downarrow\sigma \mid \exists\alpha{:}k.\rho
\end{array}
$$

As usual in recent presentations of focused logic, we use an explicit down-shift connective ($\downarrow$) to include negative signatures into the positives. It would not be uncommon also to include positive signatures into the negatives using an explicit up-shift ($\uparrow$). However, up-shift often creates sticky technical problems in focused logics [51, 6], problems that

$$
\begin{array}{lcl}
\mathsf{S}(c : k) & : & \mathit{kind} \\
\mathsf{S}(c : 1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{S}(c : \mathsf{Type}) & \stackrel{\text{def}}{=} & \mathsf{S}(c) \\
\mathsf{S}(c : \mathsf{S}(c')) & \stackrel{\text{def}}{=} & \mathsf{S}(c) \\
\mathsf{S}(c : \Pi\alpha{:}k_1.k_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}k_1.\mathsf{S}(c\,\alpha : k_2) \\
\mathsf{S}(c : \Sigma\alpha{:}k_1.k_2) & \stackrel{\text{def}}{=} & \mathsf{S}(\pi_1 c : k_1) \times \mathsf{S}(\pi_2 c : [\pi_1 c/\alpha]k_2) \\
\\
\mathsf{S}(c : \sigma) & : & \mathit{signature} \\
\mathsf{S}(c : 1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{S}(c : (\!|\, k \,|\!)) & \stackrel{\text{def}}{=} & (\!|\, \mathsf{S}(c : k) \,|\!) \\
\mathsf{S}(c : \langle\!|\, \tau \,|\!\rangle) & \stackrel{\text{def}}{=} & \langle\!|\, \tau \,|\!\rangle \\
\mathsf{S}(c : \Pi\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\sigma_1.\mathsf{S}(c\,\alpha : \sigma_2) \\
\mathsf{S}(c : \Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \mathsf{S}(\pi_1 c : \sigma_1) \times \mathsf{S}(\pi_2 c : [\pi_1 c/\alpha]\sigma_2)
\end{array}
$$

Figure 3: Higher-order singletons

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

Figure 4: Static semantic judgements

are solved by replacing the up-shift with a lax modality [12]. The lax modality (also known as a monad [34]) isolates terms that depend on eliminating positive types, so they do not infect the negative types.

It appears that up-shift would create problems in this setting as well. With up-shift, it would be necessary to define a Fst translation for positive types, in particular for existential signatures. This would presumably have to be some sort of existential kind, and extending the singleton-kind theory to support existential kinds is exactly the sort of complication we are hoping to avoid. With a lax modality instead, we will be able to avoid ever asking for the Fst of a positive signature.

But even if up-shift could work out (perhaps it can), it is still worthwhile to adopt the lax modality, because it offers an opportunity to simplify the treatment of sealing. Monads have long been recognized as a mechanism to isolate effectful computations within an otherwise-pure language. In this case the effect of interest is sealing.

We have two sorts of modules. Ordinary modules (M) are pure (they contain no sealing or generativity) and inhabit negative signatures. *Lax modules* (L) may contain sealing/generativity, and inhabit positive signatures. The signature $\bigcirc\rho$ contains modules of the form $\mathsf{circ}\ L$, which is a suspension of the lax module L. (Note that we use the monad only for sealing effects, not for ordinary term-level computational effects.)

$$
\begin{array}{llcl}
\text{modules} & M & ::= & \cdots \mid \mathsf{circ}\ L \\
\text{lax modules} & L & ::= & \mathsf{ret}\ M \mid M :> \sigma \\
 & & \mid & \mathsf{bind}\ \alpha/m \leftarrow M\ \mathsf{in}\ L \\
 & & \mid & \mathsf{unpack}\ [\alpha, x] = e\ \mathsf{in}\ L \\
\text{terms} & e & ::= & \cdots \mid \mathsf{let}\ \alpha/m = L\ \mathsf{in}\ e
\end{array}
$$

There are four forms of lax module. The form $\mathsf{ret} M$ is the usual monadic unit. The other base case is the sealing form $M :> \sigma$. This is like ret except it imposes a signature. The monadic bind forces a suspension, possibly releasing effects. Finally, the form to unpack an existential type is also a lax module, since the term could compute types dynamically and must therefore be treated as generative.

We allow the form to let-bind a module in a term to take a lax module. This is consistent with Crary [5], wherein the module being let-bound is permitted to be impure.

Note that let-binding for lax modules is syntactic sugar:[7]

$$\mathsf{let}\ \alpha/m = L_1\ \mathsf{in}\ L_2 \stackrel{\text{def}}{=} \mathsf{bind}\ \alpha/m \leftarrow \mathsf{circ}\ L_1\ \mathsf{in}\ L_2$$

Also note that in this formulation, we need not have a signature form for generative functors. Instead, a generative functor is just a functor that returns a suspension:

$$\Pi^{\mathsf{gen}}\alpha{:}\sigma.\rho \stackrel{\text{def}}{=} \Pi\alpha{:}\sigma.\bigcirc\rho$$

One aspect of this formulation that may be surprising is the polarization of functor signatures. Ordinarily one would expect the function's domain to be positive, so in the non-dependent case we would expect to see something like $\rho \to \sigma$, instead of the $\sigma \to \sigma'$ we have. However, allowing a non-trivial positive domain would create complications in the dependent case. A functor signature would look

[7] If let-binding for ordinary modules is desired, it can be obtained as syntactic sugar from lambda and application in the usual manner, or it could be added as a primitive to avoid the domain annotation.

like $\Pi pat{:}\rho.\sigma$, where *pat* is a pattern that dismantles a ρ. Without nontrivial positive functor domains, we can get by without ever defining such patterns, and I have no examples that suggest that such a facility would be useful. Moreover, it turns out that the syntactic condition that we will impose (for unrelated reasons) would prevent nontrivial positive domains from arising during type-checking anyway. For these reasons, we limit ourselves to the fragment $(\downarrow\sigma) \to \sigma'$ (and its dependent analogue) and elide the down-shift.

As an aside, one could almost eliminate strong sums in favor of existentials, non-dependent products, and singletons, writing $\Sigma\alpha{:}\sigma_1.\sigma_2$ as $\uparrow(\exists\alpha{:}\,\mathsf{Fst}(\sigma_1).\downarrow(\mathsf{S}(\alpha : \sigma_1) \times \sigma_2))$. Unfortunately, this requires up-shift, which we do not support. Moreover, even if we did support up-shift, the encoding would still run afoul of the syntactic restriction we impose in Section 6.3.

5.1 Typing lax modules

The typing rules for lax modules are given in Figure 5. Down-shift is covariant, as one would expect. We also include subtyping rules for eliminating and introducing existential signatures along the lines discussed in Section 1.

Note that compatibility for existential signatures is admissible:

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho \le \rho' : \mathsf{sig}^+}{\Gamma \vdash \exists\alpha{:}k.\rho \le \exists\alpha{:}k.\rho' : \mathsf{sig}^+}$$

The most important rule, the one that makes the system work, is the typing rule for bind (and a similar rule for let-binding lax modules in terms):

$$\frac{\Gamma \vdash M : \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L : \rho_2 \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\rho_2) = \emptyset}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M\ \mathsf{in}\ L : \rho_2}$$

Here, M is a suspension of a lax module. As a positive signature, ρ_1 must have the form $\exists\alpha_1{:}k_1 \ldots \exists\alpha_n{:}k_n.\downarrow\sigma$. When we type-check the body, we bind α/m with signature σ, but, before that, we introduce $\Gamma' = \alpha_1{:}k_1 \ldots \alpha_n{:}k_n$ so that σ is meaningful. As usual, the resulting signature ρ_2 must not rely on any variables leaving scope.

The *left inversion* judgement $\rho_1 \Rightarrow \Gamma'.\sigma$ splits the positive signature into those component parts, the prefix of kind bindings, and the negative signature that depends on them. This judgement is taken directly from focused logic, except that we do not explicitly employ a pattern. In focused logic, the judgement would typically look like $pat{:}\rho \Rightarrow \Gamma$, where *pat* is a pattern that matches against elements of ρ, thereby determining the names of the variables in Γ. In this system, however, the only accessible binding is that of α/m, so most of the variable names are immaterial. Consequently, it is more convenient to factor the definition differently, dealing with the rump pattern in the bind rule rather than in the left-inversion judgement.

This process of left inversion is very much like the flawed rule from Section 1 that split existential signatures in the context. Recall that flawed rule failed because it invalidated the substitution principle. No such problem occurs here because left inversion operates on positive signatures, not negative, and there is no such thing as a variable ranging over lax modules. Consequently, any lax module with an existential signature must have been introduced into the

---

$\boxed{\Gamma \vdash M : \sigma}$

$$\frac{\Gamma \vdash L : \rho}{\Gamma \vdash \mathsf{circ}\,L : \bigcirc\rho}$$

$\boxed{\Gamma \vdash L : \rho}$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{ret}\,M : \downarrow\sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M :> \sigma : \downarrow\sigma}$$

$$\frac{\Gamma \vdash M : \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L : \rho_2 \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\rho_2) = \emptyset}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M\ \mathsf{in}\ L : \rho_2}$$

$$\frac{\Gamma \vdash e : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash L : \rho \quad \alpha \notin FV(\rho)}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e\ \mathsf{in}\ L : \rho}$$

$$\frac{\Gamma \vdash L : \rho \quad \Gamma \vdash \rho \le \rho' : \mathsf{sig}^+}{\Gamma \vdash L : \rho'}$$

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

$$\frac{\Gamma \vdash L : \rho \quad \rho \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash e : \tau \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\tau) = \emptyset}{\Gamma \vdash \mathsf{let}\,\alpha/m = L\ \mathsf{in}\ e : \tau}$$

$\boxed{\rho \Rightarrow \Gamma.\sigma}$

$$\frac{}{\downarrow\sigma \Rightarrow \epsilon.\sigma} \qquad \frac{\rho \Rightarrow \Gamma.\sigma}{\exists\alpha{:}k.\rho \Rightarrow \alpha{:}k, \Gamma.\sigma}$$

$\boxed{\Gamma \vdash \rho \le \rho' : \mathsf{sig}^+}$

$$\frac{\Gamma \vdash \sigma \le \sigma' : \mathsf{sig}}{\Gamma \vdash \downarrow\sigma \le \downarrow\sigma' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho \le \rho' : \mathsf{sig}^+ \quad \Gamma \vdash \rho' : \mathsf{sig}^+}{\Gamma \vdash \exists\alpha{:}k.\rho \le \rho' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash \rho \le [c/\alpha]\rho' : \mathsf{sig}^+ \quad \Gamma, \alpha{:}k \vdash \rho' : \mathsf{sig}^+}{\Gamma \vdash \rho \le \exists\alpha{:}k.\rho' : \mathsf{sig}^+}$$

Figure 5: Typing Lax Modules

$\boxed{\Gamma \vdash s : \Gamma'}$

$$\frac{}{\Gamma \vdash \emptyset : \epsilon} \qquad \frac{\Gamma \vdash s : \Gamma' \quad \Gamma \vdash c : s(k)}{\Gamma \vdash (s, \alpha \mapsto c) : \Gamma', \alpha{:}k}$$

$$\frac{\Gamma \vdash s : \Gamma' \quad \Gamma \vdash e : s(\tau)}{\Gamma \vdash (s, x \mapsto e) : \Gamma', x{:}\tau}$$

$$\frac{\Gamma \vdash s : \Gamma' \quad \Gamma \vdash M : s(\sigma) \quad \Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash (s, \alpha \mapsto c, m \mapsto M) : \Gamma', \alpha/m{:}\sigma}$$

Figure 6: Substitution typing

existential—it cannot have *begun there* like a variable can—so the witness constructors must exist. Put more formally, the only form a lax-module value can take is $\mathsf{ret}\, M$, and we can show (Lemma 5.2) that if $\mathsf{ret}\, M : \rho$ and $\rho \Rightarrow \Gamma.\sigma$, then there exists an appropriate substitution for Γ.

Moreover, unlike the flawed rule, left inversion is not a free-floating rule. This greatly simplifies the meta-theory, since left inversion is applicable only in certain places (specifically, when typing a bind), instead of everywhere.

As a technical note, we employ the usual convention that variables bound in a context must be distinct. In the case of the bind rule (and also the rule for let-binding a module in a term) that means that the domains of Γ and Γ' are distinct from one another, and from α and m. When this is not the case, one can alpha-vary α/m or the bindings produced by left inversion.

Degenerate static portions We also must revisit modules' static portions in light of positive signatures and lax modules, and provide cases for $\mathsf{Fst}(\bigcirc\rho)$ and $\mathsf{Fst}(\mathsf{circ}\ L)$. Since generative computations create types at run time (at least notionally, and sometimes actually), they have no interesting static component. Since we must nevertheless provide those cases, we make them trivial:

$$\mathsf{Fst}(\bigcirc\rho) \stackrel{\text{def}}{=} 1$$

$$\frac{}{\Gamma \vdash \mathsf{Fst}(\mathsf{circ}\ L) \gg \star}$$

5.2 Metatheory

In order to prove type safety for this system, we require two lemmas pertaining to lax modules. First, we define typing for substitutions as in Figure 6.[8] Next we define canonical forms for positive signatures:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{ret}\, M ::: \downarrow\sigma} \qquad \frac{\Gamma \vdash c : k \quad \Gamma \vdash L ::: [c/\alpha]\rho}{\Gamma \vdash L ::: \exists\alpha{:}k.\rho} \qquad \text{(no other rules)}$$

Then we can state the canonical forms lemma:

Lemma 5.1 (Lax canonical forms) *If* $\Gamma \vdash \mathsf{ret} M : \rho$ *then* $\Gamma \vdash \mathsf{ret}\, M ::: \rho$.

Proof Sketch

By induction on the derivation, using a lemma stating that canonical forms respect subsumption.

[8] The definition is more general than we need right now.

$\Gamma \vdash k \Leftarrow \mathsf{kind}$	Algorithmic kind formation
$\Gamma \vdash k \trianglelefteq k'$	Algorithmic subkind
$\Gamma \vdash c \Rightarrow k$	Algorithmic kind synthesis
$\Gamma \vdash c \Downarrow c'$	Weak-head-normal form
$\Gamma \vdash c \Leftrightarrow c' : k$	Algorithim constructor equivalence
$\Gamma \vdash \sigma \Leftarrow \mathsf{sig}$	Algorithmic signature formation
$\Gamma \vdash \rho \Leftarrow \mathsf{sig}^+$	Algorithmic pos. signature formation
$\Gamma \vdash \sigma \trianglelefteq \sigma'$	Algorithmic subsignature
$\Gamma \vdash \rho \trianglelefteq \rho'$	Algorithmic positive subsignature
$\Gamma \vdash e \Rightarrow \tau$	Algorithmic type synthesis
$\Gamma \vdash M \Rightarrow \sigma$	Algorithmic signature synthesis
$\Gamma \vdash L \Rightarrow \rho$	Algorithmic pos. signature synthesis

Figure 7: Algorithmic Judgements

Using this, we can prove a cut principle for positive signatures. This is the key lemma for type preservation with existential signatures.

Lemma 5.2 (Cut) *If* $\Gamma \vdash \mathsf{ret}\, M : \rho$ *and* $\rho \Rightarrow \Gamma'.\sigma$ *then there exists a substitution* s *such that* $\Gamma \vdash s : \Gamma'$ *and* $\Gamma \vdash M : s(\sigma)$.

Proof Sketch

By induction on the first derivation.

With these in hand, we can prove type safety. The cut lemma arises in the preservation case for bind.

Theorem 5.3 (Type preservation) *If* $\vdash \Gamma\ \mathsf{ok}$ *then:*

- *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e \mapsto e'$ *then* $\Gamma \vdash e' : \tau$.
- *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash M' : \sigma$.
- *If* $\Gamma \vdash L : \rho$ *and* $\Gamma \vdash L \mapsto L'$ *then* $\Gamma \vdash L' : \rho$.
- *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma \vdash \mathsf{Fst}(M) \gg c$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash \mathsf{Fst}(M') \gg c'$ *and* $\Gamma \vdash c \equiv c' : \mathsf{Fst}(\sigma)$.

Theorem 5.4 (Progress) *If* $\vdash e : \tau$ *then either* e *is a value or takes a step. If* $\vdash M : \sigma$ *then either* M *is a value or takes a step. If* $\vdash L : \rho$ *then either* L *is a value or takes a step.*

6 Type Checking and the Avoidance Problem

Our type synthesis algorithm is given by a variety of judgements, summarized in Figure 7. The algorithm for kind and constructor formation, subkinding, and constructor equivalence is taken directly from Stone and Harper [46] (except for our different collection of primitive type operators). The algorithm for term, module, and lax module formation is new. We write it $\Gamma \vdash E \Rightarrow A$, in which E is a term, module, or lax module, and A is its type, principal signature, or principal positive signature.

6.1 Terms

The declarative typing rule for *term-level* unpack is:

$$\frac{\Gamma \vdash e_1 : \exists \alpha{:}k_1.\tau_1 \quad \Gamma, \alpha{:}k_1, x{:}\tau_1 \vdash e_2 : \tau_2 \quad \alpha \notin FV(\tau_2)}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1 \,\mathsf{in}\, e_2 : \tau_2}$$

This requires that the result type τ_2 be well-formed in the ambient context. However, we cannot reject the program if the type we synthesize mentions α, because we must determine if there exists an alternative type that does not mention α. This is like the avoidance problem, but since there is no subtyping, we only need to look for an *equivalent* alternative, not a *best* alternative.

Every kind is inhabited, so suppose $\Gamma \vdash c : k_1$. Certainly $\Gamma \vdash [c/\alpha]\tau_2 : \mathsf{Type}$. We then check whether $\Gamma, \alpha{:}k_1 \vdash \tau_2 \equiv [c/\alpha]\tau_2 : \mathsf{Type}$. If so, $[c/\alpha]\tau_2$ is a suitable alternative.

Conversely suppose there exists an alternative τ_2'. That is, $\Gamma, \alpha{:}k_1 \vdash \tau_2 \equiv \tau_2' : \mathsf{Type}$ and $\Gamma \vdash \tau_2' : \mathsf{Type}$. Then $\Gamma \vdash [c/\alpha]\tau_2 \equiv [c/\alpha]\tau_2' : \mathsf{Type}$ by substitution. But α does not appear free in τ_2', so $[c/\alpha]\tau_2' = \tau_2'$. Then, using weakening, $\Gamma, \alpha{:}k_1 \vdash \tau_2 \equiv \tau_2' = [c/\alpha]\tau_2' \equiv [c/\alpha]\tau_2 : \mathsf{Type}$. Thus, if any alternative exists, then $[c/\alpha]\tau_2$ will serve.

This gives us the type synthesis rule:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash \tau_1 \Downarrow \exists \alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2 \Rightarrow \tau_2 \quad \Gamma, \alpha{:}k \vdash \tau_2 \Leftrightarrow [I_k/\alpha]\tau_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1 \,\mathsf{in}\, e_2 \Rightarrow [I_k/\alpha]\tau_2}$$

Here, the judgement $\Gamma \vdash \tau_1 \Downarrow \exists \alpha{:}k.\tau$ places τ_1 into weak-head-normal form, which is necessary because τ_1 might be equivalent to an existential, but not in the form of an existential. Like the other constructor-oriented judgements, it is taken from Stone and Harper.

The constructor I_k is an arbitrary inhabitant of k. We can define it:

$$\begin{array}{lcl} I_{\mathsf{Type}} & \stackrel{\text{def}}{=} & \mathsf{unit} \\ I_{\mathsf{S}(c)} & \stackrel{\text{def}}{=} & c \\ I_{\Pi\alpha:k_1.k_2} & \stackrel{\text{def}}{=} & \lambda\alpha{:}k_1.I_{k_2} \\ I_{\Sigma\alpha:k_1.k_2} & \stackrel{\text{def}}{=} & \langle I_{k_1}, I_{[I_{k_1}/\alpha]k_2}\rangle \end{array}$$

This suffices for the purposes in this paper. As a practical matter, however, when unit appears in error messages out of the blue, it can be confusing to programmers. It is better, instead, to replace unit with an abstract type that is used for no other purpose.

A similar phenomenon occurs with the rule for let-binding modules in terms:

$$\frac{\Gamma \vdash L : \rho \quad \rho \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash e : \tau \quad (\mathrm{Dom}(\Gamma') \cup \{\alpha\}) \cap FV(\tau) = \emptyset}{\Gamma \vdash \mathsf{let}\,\alpha/m = L \,\mathsf{in}\, e : \tau}$$

except in this case we must substitute for every variable in Γ', as well as α:

$$\frac{\Gamma \vdash L : \rho \quad \rho \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash e \Rightarrow \tau \quad \Gamma, \Gamma', \alpha{:}\,\mathsf{Fst}(\sigma) \vdash \tau \Leftrightarrow I_{\Gamma'}([I_{\mathsf{Fst}(\sigma)}/\alpha]\tau) : \mathsf{Type}}{\Gamma \vdash \mathsf{let}\,\alpha/m = L \,\mathsf{in}\, e \Rightarrow I_{\Gamma'}([I_{\mathsf{Fst}(\sigma)}/\alpha]\tau)}$$

where:

$$\begin{array}{lcl} I_\epsilon & \stackrel{\text{def}}{=} & \emptyset \\ I_{\Gamma,\alpha:k} & \stackrel{\text{def}}{=} & I_\Gamma, \alpha \mapsto I_{I_\Gamma(k)} \end{array}$$

6.2 Modules

Alas, the arbitrary-substitution device works only in the absence of subtyping, so it does not apply to modules. (It also would not work in a core language with subtyping [13].) If σ is a signature that depends on α, there can easily be supersignatures that avoid α and yet are not equivalent to σ. This means that not all alternatives are equivalent, and we must find the best one.

As we saw in Section 1, a best alternative might not exist without existential signatures. Now that we have appropriate notation, we can reprise the example in more compact form. Suppose $\alpha{:}\mathsf{Type}$ and consider the kind $(\mathsf{Type} \to \mathsf{S}(\alpha)) \times \mathsf{S}(\alpha)$. A supersignature avoiding α is $(\mathsf{Type} \to \mathsf{Type}) \times \mathsf{Type}$. But that is not minimal. For any choice of τ, a better alternative is $\Sigma\beta{:}(\mathsf{Type} \to \mathsf{Type}).\mathsf{S}(\beta\,\tau)$. Unfortunately, for different τ these choices are incomparable, so none of them is best.

It follows that, in general, there is no best superkind avoiding a variable. Since the static-atom signature $(\!|-|\!)$ is covariant, it further follows that there is also no best supersignature avoiding a variable.

However, there *is* a best *positive* supersignature avoiding a variable, because we have created one by fiat. If we take the example and wrap it up as a positive signature, we obtain $\downarrow\!(\!|\,(\mathsf{Type} \to \mathsf{S}(\alpha)) \times \mathsf{S}(\alpha)\,|\!)$. A best supersignature avoiding α is just $\exists\alpha{:}k.\downarrow\!(\!|\,(\mathsf{Type} \to \mathsf{S}(\alpha)) \times \mathsf{S}(\alpha)\,|\!)$. There are other best alternatives as well (*e.g.,* $\exists\alpha{:}k.\exists\beta{:}1.\downarrow\!(\!|\,(\mathsf{Type} \to \mathsf{S}(\alpha)) \times \mathsf{S}(\alpha)\,|\!)$) but each one is a subsignature of the other.

Returning to type-checking, the declarative typing rule for module-level unpack is:

$$\frac{\Gamma \vdash e : \exists \alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash L : \rho \quad \alpha \notin FV(\rho)}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e \,\mathsf{in}\, L : \rho}$$

The corresponding signature synthesis rule is:

$$\frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \Downarrow \exists \alpha{:}k.\tau' \quad \Gamma, \alpha{:}k, x{:}\tau' \vdash L \Rightarrow \rho}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e \,\mathsf{in}\, L \Rightarrow \exists \alpha{:}k.\rho}$$

For bind, the situation is more complicated, but fundamentally no different. The declarative rule is:

$$\frac{\Gamma \vdash M : \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L : \rho_2 \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\rho_2) = \emptyset}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M \,\mathsf{in}\, L : \rho_2}$$

The corresponding signature synthesis rule is:

$$\frac{\Gamma \vdash M \Rightarrow \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L \Rightarrow \rho_2}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M \,\mathsf{in}\, L \Rightarrow \exists\Gamma'.\exists\alpha{:}\,\mathsf{Fst}(\sigma).\rho_2}$$

where $\exists\Gamma.\rho$ is shorthand for $\exists\alpha_1{:}k_1 \ldots \exists\alpha_n{:}k_n.\rho$ when $\Gamma = \alpha_1{:}k_1 \ldots \alpha_n{:}k_n$.

6.3 Subsignatures

Existential signatures cleanly solve the avoidance problem, but they create a problem of their own, beyond the metatheoretic difficulties that focusing resolved. The monadic operator is covariant (if it were not, suspensions would not have principal signatures), so we a need subsignature algorithm

for positive signatures, and that algorithm must wrestle with existential signatures.

Unfortunately, it is not hard to show that the positive subsignature problem is undecidable. Consider the judgement:

$$\vdash \downarrow\Pi\alpha{:}(\!|\, k \,|\!).(\!|\, \tau_1 \,|\!) \leq \exists\beta{:}k.\downarrow\Pi\alpha{:}(\!|\, \mathsf{S}(\beta{:}k) \,|\!).(\!|\, \tau_2 \,|\!) : \mathsf{sig}^{+}$$

This is derivable if and only if there exists some $c : k$ such that $[c/\alpha]\tau_1 \equiv [c/\alpha]\tau_2 : \mathsf{Type}$.[9] Thus deciding subsignatures is at least as hard as higher-order unification, which is undecidable [15].

One way to deal with this problem is simply to live with it. Higher-order unification is undecidable, but there do exist complete semi-decision procedures [23, 11]. (Singleton kinds would further complicate unification, but they can be eliminated [3].) So one could simply accept that the type checker might sometimes search endlessly instead of returning a type error. But we would prefer to avoid this behavior.

Fortunately, existential signatures pose a problem only when they appear on the right-hand-side of a subsignature query. (Existentials on the left-hand-side use a parametric inhabitant, rather than a specific inhabitant that must be guessed.) A similar issue arose in Biswas [2], in which a substitution for type variables was uniquely determined only when the unknown variables all came from the left-hand-side. In Biswas, the condition was naturally provided by the syntax. In our setting we need to add an extra syntactic restriction, one that fortunately appears to be no serious limitation in practice: We forbid the text of the program from mentioning existential signatures. To see why that helps, let us say that a signature (either negative or positive) is *synthesis* if existential signatures appear only in positive positions, and *analysis* if they appear only in negative positions. (We will also say a context is synthesis if it gives every module variable a synthesis signature.) If a subsignature query is between a synthesis and analysis signature, in that order, it will never encounter an existential signature on the right.

(Note that the classification of signatures as synthesis and/or analysis is different from the syntactic distinction between negative and positive signatures. Any signature—positive or negative—can be synthesis, analysis, both, or neither.)

Let us say that an expression is *user* if it contains no existential signatures. Clearly a user signature is both synthesis and analysis. Inspection of the algorithm shows that subsignatures are checked only in functor application and sealing, and in each case the prospective supersignature is taken from the text of the program. Therefore the right-hand-side of every subsignature query will be analysis.

We can also show that principal signature synthesis always produces synthesis signatures, provided that the module is user and the context is synthesis:

Lemma 6.1 (Synthesis signatures) *Suppose* Γ *is synthesis,* M *is user, and* $\Gamma \vdash M \Rightarrow \sigma$*. Then* σ *is synthesis.*

Proof Sketch

The domain of every functor is taken from the text of the program, and is therefore user. The invariant that the context is synthesis is maintained because user signatures are synthesis. Conversely, whenever a functor signature is synthesized, its domain will be user, and hence analysis. Thus, if the signature of the body is synthesis, the signature of the functor as a whole will be synthesis.

[9] Assuming k, τ_1, and τ_2 are appropriately well-formed.

Practicum This syntactic restriction is no burden in practice, because in any circumstance the programmer might be inclined to use an existential signature (*e.g.,* $\exists\alpha{:}k.\downarrow\sigma$), he or she could use a strong sum instead (*e.g.,* $\Sigma\alpha{:}(\!|\, k \,|\!).\sigma$). The power existential signatures bring to the table is useful for automatically synthesizing signatures, but it offers nothing new for explicitly specified signatures.

Moreover, our signatures admit full separate compilation—a property closely related to Shao's "fully syntactic" signatures [42]. The term "fully syntactic" might suggest that all signatures must be expressible in syntax available to the programmer, which is certainly not the case here. But Shao actually defines the term in terms of practical considerations: "If we split a program at an arbitrary point, the corresponding interface must be expressible using the underlying signature calculus."

In other words, however a software project might be broken into compilation units, there exist syntactic representations of the signatures needed to mediate the boundaries between units. To clarify this, we must distinguish between two forms of separate compilation [48]:

In separate compilation *per se,* the programmer explicitly specifies an interface for unit `A`, and unit `B` depends on `A` via that interface. The code for `A` might not be available, or even written yet. In *incremental recompilation,* the code for unit `A` is compiled, automatically generating an interface, which unit `B` then depends on. When the code for `B` changes, it can be recompiled without recompiling `A`.

In separate compilation *per se,* the programmer(s) determine `A`'s interface and they can use strong sums in the interface instead of existential signatures. In incremental recompilation, on the other hand, the programmers wish to take `A`'s code *as is* and compile against it, perhaps for build efficiency or perhaps because `A` is upstream code provided only in binary. Either way, `A` itself determines the interface. This is the situation in which "fully syntactic" signatures are relevant.

But existential signatures are syntactic enough for incremental recompilation. When the compiler compiles `A`, it writes out an interface containing the signatures inferred for `A`, which may include existential signatures. Since the interface is *generated,* nothing is every checked against it. The contents of `A` are made available to `B` according to that interface, and that is fine: Recall that the context is required to be synthesis, not user, so it is permitted to contain `A`'s interface, which is synthesis by Lemma 6.1.

In summary, code that cannot be altered need not be checked against a signature, one just compiles against it, thus fitting into an incremental recompilation scenario. And, even though existential signatures cannot appear within user code, they *can* appear within automatically generated interfaces, which makes them syntactic enough for incremental compilation purposes.

6.4 The Algorithm

The type-checking algorithm for terms and modules is given in Figure 8. For signatures the algorithm is straightforward and appears in Appendix C, and for kinds and constructors it is in Stone and Harper [46]. We touched on most of the

$\boxed{\Gamma \vdash e \Rightarrow \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash \star \Rightarrow \mathsf{unit}} \qquad \frac{\Gamma \vdash \tau_1 \Leftarrow \mathsf{Type} \quad \Gamma, x{:}\tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e \Rightarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash \tau_1 \Downarrow \tau \to \tau' \quad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1 e_2 \Rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \Downarrow \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e \Rightarrow \tau_1} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \Downarrow \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 e \Rightarrow \tau_2}$$

$$\frac{\Gamma \vdash k \Leftarrow \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash sv \Rightarrow \tau}{\Gamma \vdash \Lambda\alpha{:}k.sv \Rightarrow \forall\alpha{:}k.\tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \Downarrow \forall\alpha{:}k.\tau' \quad \Gamma \vdash c \Leftarrow k}{\Gamma \vdash e[c] \Rightarrow [c/\alpha]\tau'}$$

$$\frac{\Gamma \vdash k \Leftarrow \mathsf{kind} \quad \Gamma \vdash c \Leftarrow k \quad \Gamma \vdash e \Leftarrow [c/\alpha]\tau \quad \Gamma, \alpha{:}k \vdash \tau \Leftarrow \mathsf{Type}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau \Rightarrow \exists\alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash \tau_1 \Downarrow \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2 \Rightarrow \tau_2 \quad \Gamma, \alpha{:}k \vdash \tau_2 \Leftrightarrow [I_k/\alpha]\tau_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2 \Rightarrow [I_k/\alpha]\tau_2}$$

$$\frac{\Gamma \vdash \tau \Leftarrow \mathsf{Type} \quad \Gamma \vdash e \Leftarrow (\mathsf{unit} \to \tau) \to \tau}{\Gamma \vdash \mathsf{fix}_\tau\, e \Rightarrow \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1\,\mathsf{in}\,e_2 : \tau_2}$$

$$\frac{\Gamma \vdash L : \rho \quad \rho \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash e \Rightarrow \tau \quad \Gamma, \Gamma', \alpha{:}\,\mathsf{Fst}(\sigma) \vdash \tau \Leftrightarrow I_{\Gamma'}([I_{\mathsf{Fst}(\sigma)}/\alpha]\tau) : \mathsf{Type}}{\Gamma \vdash \mathsf{let}\,\alpha/m = L\,\mathsf{in}\,e \Rightarrow I_{\Gamma'}([I_{\mathsf{Fst}(\sigma)}/\alpha]\tau)} \qquad \frac{\Gamma \vdash M \Rightarrow \langle\!|\,\tau\,|\!\rangle}{\Gamma \vdash \mathsf{Ext}(M) \Rightarrow \tau}$$

$$\text{static value} \quad sv \;::=\; \star \mid \lambda x{:}\tau.e \mid \langle sv, sv\rangle \mid \Lambda\alpha{:}k.sv \mid \mathsf{pack}\,[c, sv]\,\mathsf{as}\,\exists\alpha{:}k.\tau$$

$\boxed{\Gamma \vdash M \Rightarrow \sigma}$

$$\frac{\alpha/m{:}\sigma \in \Gamma}{\Gamma \vdash m \Rightarrow \mathsf{S}(\alpha : \sigma)} \qquad \frac{}{\Gamma \vdash \star \Rightarrow 1} \qquad \frac{\Gamma \vdash c \Rightarrow k}{\Gamma \vdash \langle\!|\,c\,|\!\rangle \Rightarrow \langle\!|\,k\,|\!\rangle} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \langle\!|\,e\,|\!\rangle \Rightarrow \langle\!|\,\tau\,|\!\rangle}$$

$$\frac{\Gamma \vdash \sigma_1 \Leftarrow \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash M \Rightarrow \sigma_2}{\Gamma \vdash \lambda\alpha/m{:}\sigma_1.M \Rightarrow \Pi\alpha{:}\sigma_1.\sigma_2} \qquad \frac{\Gamma \vdash M_1 \Rightarrow \Pi\alpha{:}\sigma.\sigma' \quad \Gamma \vdash M_2 \Leftarrow \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash M_1 M_2 \Rightarrow [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash M_1 \Rightarrow \sigma_1 \quad \Gamma \vdash M_2 \Rightarrow \sigma_2}{\Gamma \vdash \langle M_1, M_2\rangle \Rightarrow \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash M \Rightarrow \Sigma\alpha{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_1 M \Rightarrow \sigma_1} \qquad \frac{\Gamma \vdash M \Rightarrow \Sigma\alpha{:}\sigma_1.\sigma_2 \quad \Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \pi_2 M \Rightarrow [\pi_1 c/\alpha]\sigma_2}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma, x{:}\tau \vdash M \Rightarrow \sigma}{\Gamma \vdash \mathsf{let}\, x = e\,\mathsf{in}\,M \Rightarrow \sigma} \qquad \frac{\Gamma \vdash L \Rightarrow \rho}{\Gamma \vdash \mathsf{circ}\, L \Rightarrow \bigcirc\rho}$$

$\boxed{\Gamma \vdash L \Rightarrow \rho}$

$$\frac{\Gamma \vdash M \Rightarrow \sigma}{\Gamma \vdash \mathsf{ret}\, M \Rightarrow \downarrow\sigma} \qquad \frac{\Gamma \vdash \sigma \Leftarrow \mathsf{sig} \quad \Gamma \vdash M \Leftarrow \sigma}{\Gamma \vdash (M :> \sigma) \Rightarrow \downarrow\sigma}$$

$$\frac{\Gamma \vdash M \Rightarrow \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L \Rightarrow \rho_2}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M\,\mathsf{in}\,L \Rightarrow \exists\Gamma'.\exists\alpha{:}\,\mathsf{Fst}(\sigma).\rho_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \Downarrow \exists\alpha{:}k.\tau' \quad \Gamma, \alpha{:}k, x{:}\tau' \vdash L \Rightarrow \rho}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,L \Rightarrow \exists\alpha{:}k.\rho}$$

$\boxed{\Gamma \vdash e \Leftarrow \tau \qquad \Gamma \vdash M \Leftarrow \sigma}$

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \Gamma \vdash \tau' \Leftrightarrow \tau : \mathsf{Type}}{\Gamma \vdash e \Leftarrow \tau} \qquad \frac{\Gamma \vdash M \Rightarrow \sigma' \quad \Gamma \vdash \sigma' \trianglelefteq \sigma}{\Gamma \vdash M \Leftarrow \sigma}$$

$\boxed{\Gamma \vdash \sigma \trianglelefteq \sigma'}$

$$\frac{}{\Gamma \vdash 1 \trianglelefteq 1} \qquad \frac{\Gamma \vdash k \trianglelefteq k'}{\Gamma \vdash \langle\!|\,k\,|\!\rangle \trianglelefteq \langle\!|\,k'\,|\!\rangle} \qquad \frac{\Gamma \vdash \tau \Leftrightarrow \tau' : \mathsf{Type}}{\Gamma \vdash \langle\!|\,\tau\,|\!\rangle \trianglelefteq \langle\!|\,\tau'\,|\!\rangle} \qquad \frac{\Gamma \vdash \rho \trianglelefteq \rho'}{\Gamma \vdash \bigcirc\rho \trianglelefteq \bigcirc\rho'}$$

$$\frac{\Gamma \vdash \sigma_1' \trianglelefteq \sigma_1 \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \trianglelefteq \sigma_2'}{\Gamma \vdash \Pi\alpha{:}\sigma_1.\sigma_2 \trianglelefteq \Pi\alpha{:}\sigma_1'.\sigma_2'} \qquad \frac{\Gamma \vdash \sigma_1 \trianglelefteq \sigma_1' \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \trianglelefteq \sigma_2'}{\Gamma \vdash \Sigma\alpha{:}\sigma_1.\sigma_2 \trianglelefteq \Sigma\alpha{:}\sigma_1'.\sigma_2'}$$

$\boxed{\Gamma \vdash \rho \trianglelefteq \rho'}$

$$\frac{\Gamma \vdash \sigma \trianglelefteq \sigma'}{\Gamma \vdash \downarrow\sigma \trianglelefteq \downarrow\sigma'} \qquad \frac{\Gamma, \alpha{:}k \vdash \rho \trianglelefteq \rho'}{\Gamma \vdash \exists\alpha{:}k.\rho \trianglelefteq \rho'}$$

Figure 8: Type-checking algorithm

key rules above. However, we should discuss one more rule, which is standard in module type theory, but nevertheless may be surprising. When type-checking a variable bound with signature σ, the principal signature is *not* σ. Instead, we must *selfify* it:

$$\frac{\alpha/m{:}\sigma \in \Gamma}{\Gamma \vdash m \Rightarrow \mathsf{S}(\alpha : \sigma)}$$

For example, suppose σ is just $(\!|\,\mathsf{Type}\,|\!)$. Then the best signature for m is not $(\!|\,\mathsf{Type}\,|\!)$, but the subsignature $(\!|\,\mathsf{S}(\alpha)\,|\!)$. In general, selfification (recall Figure 3) fills in all the signature's type components other than those beneath a monad.

We can now state the main results of the paper. The results involving constructors and kinds are taken from Stone and Harper [46] and are repeated here for the sake of completeness. The other results are new.

Theorem 6.2 (Soundness) *Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

1. *If* $\Gamma \vdash k \Leftarrow \mathsf{kind}$ *then* $\Gamma \vdash k : \mathsf{kind}$.
2. *If* $\Gamma \vdash k, k' : \mathsf{kind}$ *and* $\Gamma \vdash k \trianglelefteq k'$ *then* $\Gamma \vdash k \leq k' : \mathsf{kind}$.
3. *If* $\Gamma \vdash c \Rightarrow k$ *then* $\Gamma \vdash c : k$.
4. *If* $\Gamma \vdash c : k$ *and* $\Gamma \vdash c \Downarrow c'$ *then* $\Gamma \vdash c \equiv c' : k$.
5. *If* $\Gamma \vdash c_1, c_2 : k$ *and* $\Gamma \vdash c_1 \Leftrightarrow c_2 : k$ *then* $\Gamma \vdash c_1 \equiv c_2 : k$.
6. *If* $\Gamma \vdash \sigma \Leftarrow \mathsf{sig}$ *then* $\Gamma \vdash \sigma : \mathsf{sig}$.
7. *If* $\Gamma \vdash \rho \Leftarrow \mathsf{sig}^+$ *then* $\Gamma \vdash \rho : \mathsf{sig}^+$.
8. *If* $\Gamma \vdash \sigma, \sigma' : \mathsf{sig}$ *and* $\Gamma \vdash \sigma \trianglelefteq \sigma'$ *then* $\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}$.
9. *If* $\Gamma \vdash \rho, \rho' : \mathsf{sig}^+$ *and* $\Gamma \vdash \rho \trianglelefteq \rho'$ *then* $\Gamma \vdash \rho \leq \rho' : \mathsf{sig}^+$.
10. *If* $\Gamma \vdash e \Rightarrow \tau$ *then* $\Gamma \vdash e : \tau$.
11. *If* $\Gamma \vdash M \Rightarrow \sigma$ *then* $\Gamma \vdash M : \sigma$.
12. *If* $\Gamma \vdash L \Rightarrow \rho$ *then* $\Gamma \vdash L : \rho$.

Theorem 6.3 (Completeness) *Suppose* $\vdash \Gamma\ \mathsf{ok}$. *Then:*

1. *If* $\Gamma \vdash k : \mathsf{kind}$ *then* $\Gamma \vdash k \Leftarrow \mathsf{kind}$.
2. *If* $\Gamma \vdash k \leq k' : \mathsf{kind}$ *then* $\Gamma \vdash k \trianglelefteq k'$.
3. *If* $\Gamma \vdash c : k$ *then* $\Gamma \vdash c \Rightarrow k'$ *and* $\Gamma \vdash k' \leq k : \mathsf{kind}$.
4. *If* $\Gamma \vdash c : k$ *then* $\Gamma \vdash c \Downarrow c'$.
5. *If* $\Gamma \vdash c_1 \equiv c_2 : k$ *then* $\Gamma \vdash c_1 \Leftrightarrow c_2 : k$.
6. *If* $\Gamma \vdash \sigma : \mathsf{sig}$ *then* $\Gamma \vdash \sigma \Leftarrow \mathsf{sig}$.
7. *If* $\Gamma \vdash \rho : \mathsf{sig}^+$ *then* $\Gamma \vdash \rho \Leftarrow \mathsf{sig}^+$.
8. *If* $\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}$ *and* σ *is synthesis and* σ' *is analysis, then* $\Gamma \vdash \sigma \trianglelefteq \sigma'$.
9. *If* $\Gamma \vdash \rho \leq \rho' : \mathsf{sig}^+$ *and* ρ *is synthesis and* ρ' *is analysis, then* $\Gamma \vdash \rho \trianglelefteq \rho'$.
10. *If* $\Gamma \vdash e : \tau$ *and* Γ *is synthesis and* e *is user, then* $\Gamma \vdash e \Rightarrow \tau'$ *and* $\Gamma \vdash \tau' \equiv \tau : \mathsf{Type}$.
11. *If* $\Gamma \vdash M : \sigma$ *and* Γ *is synthesis and* M *is user, then* $\Gamma \vdash M \Rightarrow \sigma'$ *and* $\Gamma \vdash \sigma' \leq \sigma : \mathsf{sig}$.
12. *If* $\Gamma \vdash L : \rho$ *and* Γ *is synthesis and* L *is user, then* $\Gamma \vdash L \Rightarrow \rho'$ *and* $\Gamma \vdash \rho' \leq \rho : \mathsf{sig}^+$.

Corollary 6.4 (Decidability) *If* $\vdash \Gamma\ \mathsf{ok}$ *and* Γ *is synthesis, then:*

1. *If* e *is user, then it is decidable whether there exists* τ *such that* $\Gamma \vdash e : \tau$.
2. *If* M *is user, then it is decidable whether there exists* σ *such that* $\Gamma \vdash M : \sigma$.
3. *If* L *is user, then it is decidable whether there exists* ρ *such that* $\Gamma \vdash L : \rho$.

7 Generative Stamps Revisited

We now revisit Russo's static semantics [39]. Russo gives two static semantics for modules: First he gives a semantics based on generative stamps that is more-or-less equivalent to the one used by the Definition of Standard ML [31]. Then he gives a type-oriented semantics and proves it equivalent to the stamp-based semantics. Russo's approach was largely adopted by later work such as Rossberg, *et al.* [38] and Rossberg [37]. We will observe here that Russo's non-stamp static semantics is also more-or-less the same as our principal signature algorithm.

The most important and complicated case is generative functor application. Russo's rule is:

$$\frac{\begin{array}{l}\mathcal{C} \vdash s : \exists P.\mathcal{S}' \\ \mathcal{C}(F) = \forall Q.\, \mathcal{S}'' \to \mathcal{X} \\ \mathrm{Dom}(\varphi) = Q \\ \mathcal{S}' \succeq \varphi(\mathcal{S}'') \\ \varphi(\mathcal{X}) = \exists P'.\mathcal{S} \\ P \cap FV(\forall Q.\mathcal{S}'' \to \mathcal{X}) = \emptyset \\ P \cap P' = \emptyset\end{array}}{\mathcal{C} \vdash F\, s : \exists (P \cup P').\mathcal{S}}$$

Russo's notation is very different from ours, so let's unpack this. The first premise says the argument has signature $\exists P.\mathcal{S}'$. Here P is a set of type variables and $\mathcal{S}'$ is a transparent signature (*i.e.,* every type field is given a definition). The second premise says the functor has signature $\forall Q.\, \mathcal{S}'' \to \mathcal{X}$. Here, Q is a set of type variables, $\mathcal{S}''$ is the functor's transparent domain, and $\mathcal{X}$ is its possibly non-transparent codomain.

For the application to type-check, there must be a substitution φ that instantiates the variables in Q (third premise), such that the argument's signature "enriches" the functor's instantiated domain (fourth premise). "Enriches" is a synonym (borrowed from the Definition) for "is a supertype of." (Note that the inequality is turned around from how it is used in subtyping.)

The substitution is applied to the codomain, resulting in the signature $\exists P'.\mathcal{S}$ (fifth premise). The resulting signature adds P to the existential prefix, resulting in $\exists (P \cup P').\mathcal{S}$. The remaining two premises deal with variable hygiene.

The most important part of the rule is how the existential prefix P for the argument floats up to become part of the existential prefix for the result. In a more general setting, the existential prefix of the functor floats up as well, but in Standard ML the functor is required to be a bare identifier, so it has no such prefix.

We can rewrite this rule in our notation as:

$$\frac{\begin{array}{l}\Gamma \vdash L : \exists \Gamma'.{\downarrow}\sigma \\ \Gamma \vdash F : \Pi\alpha{:}\sigma_1.{\bigcirc}\rho \\ \Gamma, \Gamma' \vdash c : \mathsf{Fst}(\sigma_1) \\ \Gamma, \Gamma' \vdash \sigma \leq \mathsf{S}(c : \sigma_1) : \mathsf{sig}\end{array}}{\Gamma \vdash FL : \exists \Gamma'.\, [c/\alpha]\rho}$$

Here, s is renamed to L, P to Γ', $\mathcal{S}'$ to σ, and $\mathcal{X}$ to ρ. The signature σ_1 plays the role of both Q and $\mathcal{S}''$. (In our system we do not separate a functor's domain into two pieces as Russo does.) The constructor c plays the role of the substitution φ, which is to indicate how to specialize the functor's type components. On the domain side, $\varphi(\mathcal{S}'')$ becomes $\mathsf{S}(c : \sigma_1)$, and on the codomain side $\varphi(\mathcal{X})$ becomes $[c/\alpha]\rho$. Finally, we have no need to separate P' and $\mathcal{S}$, so they disappear, and we write $[c/\alpha]\rho$ for $\varphi(\mathcal{X}) = \exists P'.\mathcal{S}$ in the conclusion.

In our system, one is not permitted to apply a functor to a lax argument, but we can take that as syntactic sugar:

$$FL \stackrel{\text{def}}{=} \mathsf{bind}\ \alpha/m \leftarrow \mathsf{circ}\ L\ \mathsf{in}\ \mathsf{bind}\ n \leftarrow F\, m\ \mathsf{in}\ \mathsf{ret}\ n$$

With this definition, it is not hard to show that the above rule is sound. The functor has signature:

$$\begin{array}{rl} & \Pi\alpha{:}\sigma_1.{\bigcirc}\rho \\ \leq & \Pi\alpha{:}\mathsf{S}(c{:}\sigma_1).{\bigcirc}\rho \\ \equiv & \Pi\alpha{:}\mathsf{S}(c{:}\sigma_1).{\bigcirc}[c/\alpha]\rho \\ = & \mathsf{S}(c : \sigma_1) \to {\bigcirc}[c/\alpha]\rho \end{array}$$

In the body of the outer bind (wherein the context is $\Gamma, \Gamma', \alpha/m{:}\sigma$), m has signature $\sigma \leq \mathsf{S}(c : \sigma_1)$. Thus $F\, m$ has signature ${\bigcirc}[c/\alpha]\rho$. The inner bind serves to discharge the $\bigcirc$, producing $[c/\alpha]\rho$. Finally, $[c/\alpha]\rho$ is a subsignature of $\exists \Gamma'.[c/\alpha]\rho$, which is closed with respect to everything but Γ.

But the relationship between Russo's system and ours is actually closer than that. If we apply our principal signature algorithm to FL, we obtain the derived rule:[10]

$$\frac{\begin{array}{l}\Gamma \vdash F \Rightarrow \Pi\alpha{:}\sigma_1.{\bigcirc}\rho \\ \Gamma \vdash L \Rightarrow \exists \Gamma'.{\downarrow}\sigma \\ \Gamma, \Gamma', \alpha{:}\,\mathsf{Fst}(\sigma) \vdash \mathsf{S}(\alpha : \sigma) \trianglelefteq \sigma_1 \\ \rho \Rightarrow \Gamma''.\sigma_2\end{array}}{\Gamma \vdash FL \Rightarrow \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\exists \beta{:}\,\mathsf{Fst}(\sigma_2).\mathsf{S}(\beta : \sigma_2)}$$

This derived rule and (our version of) Russo's rule are nearly equivalent. Let us call Russo's rule R, and the derived signature-synthesis rule D, and let us neglect the distinction between the declarative and algorithmic systems.

R is an instance of D: The first two premises are identical. From R's third and fourth premises we can show $\mathsf{S}(\alpha : \sigma) \leq \sigma \leq \mathsf{S}(c : \sigma_1) \leq \sigma_1$, establishing D's third premise. Every positive signature can be inverted, so D's fourth premise holds as well. Thus D applies, and its resulting signature is a subsignature of R's:

$$\begin{array}{rl} & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\exists \beta{:}\,\mathsf{Fst}(\sigma_2).\mathsf{S}(\beta : \sigma_2) \\ \leq & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\exists \beta{:}\,\mathsf{Fst}(\sigma_2).\sigma_2 \\ \leq & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\sigma_2 \\ = & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\rho \\ \leq & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\mathsf{S}(c{:}\sigma_1)).\rho \\ = & \exists \Gamma'.\exists \alpha{:}\mathsf{S}(c{:}\,\mathsf{Fst}(\sigma_1)).\rho \\ \equiv & \exists \Gamma'.\exists \alpha{:}\mathsf{S}(c{:}\,\mathsf{Fst}(\sigma_1)).[c/\alpha]\rho \\ \leq & \exists \Gamma'.[c/\alpha]\rho \end{array}$$

[10] To give the rule this form, we write the left inversion associated with the outer bind into the second premise.

$\boxed{k\ \mathsf{transp}}$

$$\frac{}{1\ \mathsf{transp}} \qquad \frac{}{\mathsf{S}(c)\ \mathsf{transp}} \qquad \text{(No rule for Type.)}$$

$$\frac{k_2\ \mathsf{transp}}{\Pi\alpha{:}k_1.k_2\ \mathsf{transp}} \qquad \frac{k_1\ \mathsf{transp} \quad k_2\ \mathsf{transp}}{\Sigma\alpha{:}k_1.k_2\ \mathsf{transp}}$$

$\boxed{\sigma\ \mathsf{transp}}$

$$\frac{}{1\ \mathsf{transp}} \qquad \frac{k\ \mathsf{transp}}{(\!|\, k \,|\!)\ \mathsf{transp}} \qquad \frac{}{(\!|\, \tau \,|\!)\ \mathsf{transp}} \qquad \frac{}{{\bigcirc}\rho\ \mathsf{transp}}$$

$$\frac{\sigma_2\ \mathsf{transp}}{\Pi\alpha{:}\sigma_1.\sigma_2\ \mathsf{transp}} \qquad \frac{\sigma_1\ \mathsf{transp} \quad \sigma_2\ \mathsf{transp}}{\Sigma\alpha{:}\sigma_1.\sigma_2\ \mathsf{transp}}$$

$\boxed{\rho\ \mathsf{transp}}$

$$\frac{\sigma\ \mathsf{transp}}{{\downarrow}\sigma\ \mathsf{transp}} \qquad \frac{\rho\ \mathsf{transp}}{\exists\alpha{:}k.\rho\ \mathsf{transp}}$$

Figure 9: Transparency

Conversely, D is an instance of R, provided we assume that σ and ρ are transparent, as Russo's system arranges always to be the case: We define transparency using a syntactic condition in Figure 9. The important property of transparent signatures is $\sigma \equiv \mathsf{S}(c : \sigma)$ for every c in $\mathsf{Fst}(\sigma)$:

Lemma 7.1 *If σ is transparent, and if $\vdash \Gamma\ \mathsf{ok}$ and $\Gamma \vdash \sigma : \mathsf{sig}$ and $\Gamma \vdash c : \mathsf{Fst}(\sigma)$, then $\Gamma \vdash \sigma \equiv \mathsf{S}(c : \sigma) : \mathsf{sig}$.*

Every kind is inhabited, so let c belong to $\mathsf{Fst}(\sigma)$. Using substitution on D's third premise, we obtain $\Gamma, \Gamma' \vdash \mathsf{S}(c : \sigma) \leq \sigma_1$. But σ is transparent, so $\mathsf{S}(c : \sigma) \equiv \sigma$. Thus $\sigma \leq \sigma_1$. This establishes R's third premise, since Fst is monotone. Furthermore, higher-order singletons are monotone, so $\sigma \equiv \mathsf{S}(c : \sigma) \leq \mathsf{S}(c : \sigma_1)$, establishing R's fourth premise. Thus R applies, and its resulting signature is a subsignature of D's:

$$\begin{array}{rl} & \exists \Gamma'.[c/\alpha]\rho \\ \leq & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\rho \\ = & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.{\downarrow}\sigma_2 \\ \leq & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\exists \beta{:}\,\mathsf{Fst}(\sigma_2).{\downarrow}\sigma_2 \\ \equiv & \exists \Gamma'.\exists \alpha{:}\,\mathsf{Fst}(\sigma).\exists \Gamma''.\exists \beta{:}\,\mathsf{Fst}(\sigma_2).{\downarrow}\mathsf{S}(\beta : \sigma_2) \end{array}$$

The last step applies because σ_2 is transparent, because ρ is transparent.

There are a lot of moving parts in functor application, and the main point is easily lost in the details. The main point is how the signature-synthesis rule for bind causes the argument's existential prefix to float upward by pulling it off the argument and then putting it back on after the application. A similar, but usually much simpler, story occurs when other forms (*e.g.,* projection) are generalized to lax modules.

We have seen that R is an instance of D, and D is an instance of R provided σ and ρ are transparent. Thus the main difference between Russo's system and our algorithm is Russo's insistence that every module be given a transparent signature. Our algorithm does not do that.

However, it can be induced to do so. The only place our algorithm synthesizes a non-transparent signature is sealing. If we replace every sealed module $M :> \sigma$ with

$\mathsf{bind}\,\alpha/m \leftarrow \mathsf{circ}\,(M :> \sigma)\,\mathsf{in}\,M :> \mathsf{S}(\alpha : \sigma)$, then the algorithm will synthesize $\exists\alpha{:}\,\mathsf{Fst}(\sigma).{\downarrow}\mathsf{S}(\alpha : \sigma)$ (which is transparent), instead of $\downarrow\sigma$ (which might not be). The expanded seal is just as abstract as the original (observe that M still appears under the same seal), and $\exists\alpha{:}\,\mathsf{Fst}(\sigma).{\downarrow}\mathsf{S}(\alpha : \sigma) \leq \downarrow\sigma$, so there seems to be no detriment to the expansion other than complexity.

The upshot of all this is that the type theory developed here is consistent with Russo's type discipline, at least at a coarse level. The floating machinery simulates the effect of the left inversion rule, and vice versa. (Myriad technical differences in presentation between Russo's system and this one—including but not limited to the fact that Russo gives a full source language, not just a core calculus—make this observation hard to formalize, so let us keep it as an informal observation.) Since Russo's discipline is equivalent to a generative-stamp discipline, the type theory developed here is also consistent with that.

A similar story prevails in later work that builds on Russo, such as Rossberg, *et al.* [38]. Again, the effects of left inversion and the floating machinery are broadly consistent. (However, in the specific case discussed here, functor application, they limit both functor and argument to names, so nothing interesting can happen from an avoidance perspective.)

Finally, since Russo's generative-stamps discipline is more-or-less the same as the one in Harper, *et al.* [19] and the Definition [31], we can take this work as a type-theoretic vindication (in the sense of Section 2) of the approach used since the earliest days of the Standard ML module system.

8 Formalization

All the results in this paper are formalized in Coq (version 8.4), and are available at:

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

I implemented binding using deBruijn indices and explicit substitutions. Much of the development is adapted from Crary [5], but replacing effects inference with a monad required pervasive changes. The full development is 41 thousand lines of Coq (including comments and whitespace).

When formalizing the metatheory of singleton kinds, it proved much more convenient to use Stone and Harper's original six-place algorithm for constructor equivalence ($\Gamma \vdash c : k \Leftrightarrow \Gamma' \vdash c' : k'$) [46] than their later set-based algorithm [47], due to the difficulty working with sets in Coq. I also did not use the four-place algorithm that they proved equivalent to both other algorithms, since it would have required additional work and was not necessary for the results here, but I foresee no problem doing so.

9 Concluding Remarks

The avoidance problem is one of the main outstanding problems in ML-style module type theory. We can see now that the approach Harper, Dreyer, and I had in 2001 was very close. The missing elements were the focusing-inspired polarization of signatures and left-inversion judgement. But at the time, the idea that focusing could have an application to module calculi was not known, or at least, was not known to me. I first made the connection in the other direction, using the notion of selfification to permit assigning types to spines in focused logic in the presence of strong sums [6]. My main purpose in that work was to extend LF [17] to support strong sums.

Of course, those missing elements serve to define a language with similar expressive power (as pertains to the avoidance problem) to several that have come before. What we have now is a type theory with a clear connection to logic, and which can serve as a platform for classic type-theoretic constructions such as Reynolds's abstraction theorem [36].

With the avoidance problem resolved, I feel that we have a type theory for ML-style modules that is close to canonical. Ideally the type theory would also support named fields with permutation and width subtyping. That would certainly complicate the formalization, but it seems unlikely that it would add any fundamental difficulties. A thornier question is what becomes of weak sealing [10], which is needed to have opaque datatypes within applicative functors. With effects inference replaced by a monad, there is no longer any obvious place to put the notion of a "static effect." An even thornier desideratum is a treatment of recursive modules. It seems possible that existential signatures could give a clean way to implement the forward declarations necessary to resolve the "double-vision problem" [9], but I have not explored this.

Another question before we can say the theory is canonical is whether existential signatures should have a dual. Our failed type theory of 2001 had a universal signature, not because one was needed, but because symmetry seemed to suggest that there ought to be one. But the focused analysis tells a different story. The existential signature is positive, so its dual is a negative function space. The negative function space, of course, is the ordinary one, so there is nothing to add there. (Indeed, the focused analysis suggests that what is missing is other positive connectives such as disjoint sums and falsehood, but these have never been part of second-class module calculi.) The actual asymmetry arises from the strong sum, which is the normal one in module type theory but (as above) is unusual in focused logic. Its dual would be a *positive* function space, which is very unusual. Licata, Zeilberger, and Harper [29] have looked at the problem a little but many issues remain. Since I have no application for such a connective in the modules setting, I have not explored it.

A OCaml

OCaml could also usefully employ the strategy proposed here. OCaml's current approach to the avoidance problem is different from Standard ML's. Unlike SML, OCaml does not retain access to types that have departed scope. Instead, when a functor is applied to an impure argument, OCaml tries to find a signature that does not mention the argument's type components. If it is unable to find one, it signals a type error.

A signature that OCaml finds is not necessarily principal, even when a principal signature exists. It can also fail to find a signature entirely, even when a principal signature exists. For example:

```
module type S =
   sig
      type t
      val x : t
   end

module Id (X : S) =
   struct
      type t = X.t
      let x (* : t *) = X.x
   end

(* generative functor *)
module F () : S =
   struct
      type t = int
      let x = 12
   end

module M = Id (F ())
```

This code produces the error:

```
This functor has type

     functor (X : S) ->
       sig type t = X.t val x : X.t end

The parameter cannot be eliminated in the
result type.  Please bind the argument to
a module identifier.
```

This need not be an error; `M` could be given the principal signature `sig type t val x : t end`. On the other hand, if the commented-out type annotation in the `x` field of `Id` is present, then OCaml does find the principal solution. Thus, the type system proposed here would allow more programs to type-check, and would produce more predictable behavior.

B Syntax and Static Semantics

kinds

k	::=	1	unit kind
		Type	types
		$\mathsf{S}(c)$	singleton kind
		$\Pi\alpha{:}k.k$	dependent functions
		$\Sigma\alpha{:}k.k$	dependent pairs

constructors

c, τ	::=	α	
		$\star$	unit constructor
		$\lambda\alpha{:}k.c \mid c\,c$	lambda, application
		$\langle c, c\rangle$	pair
		$\pi_1 c \mid \pi_2 c$	projection
		unit	unit type
		$\tau_1 \to \tau_2$	functions
		$\tau_1 \times \tau_2$	products
		$\forall\alpha{:}k.\tau$	universals
		$\exists\alpha{:}k.\tau$	existentials

(negative) signatures

σ	::=	1	unit signature
		$(\!\vert k \vert\!)$	atomic signature
		$\langle\!\vert \tau \vert\!\rangle$	atomic signature
		$\Pi\alpha{:}\sigma.\sigma$	functors
		$\Sigma\alpha{:}\sigma.\sigma$	pairs
		$\bigcirc\rho$	monad

positive signatures

ρ	::=	$\downarrow\sigma$	downshift
		$\exists\alpha{:}k.\rho$	existential signature

terms

e	::=	x	
		$\star$	unit term
		$\lambda x{:}\tau.e \mid e\,e$	lambda, application
		$\langle e, e\rangle$	pair
		$\pi_1 e \mid \pi_2 e$	projection
		$\Lambda\alpha{:}k.e$	polymorphic fun.
		$e[c]$	polymorphic app.
		$\mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau$	existential package
		$\mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,e$	unpack
		$\mathsf{fix}_\tau\,e$	recursion
		$\mathsf{let}\,x = e\,\mathsf{in}\,e$	term binding
		$\mathsf{Ext}\,M$	extraction
		$\mathsf{let}\,\alpha/m = L\,\mathsf{in}\,e$	lax module binding

static values

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

modules

M	::=	m	
		$\star$	unit module
		$(\!\vert c \vert\!)$	atomic module
		$\langle\!\vert e \vert\!\rangle$	atomic module
		$\lambda\alpha/m{:}\sigma.M$	functor
		$M\,M$	functor application
		$\langle M, M\rangle$	pair
		$\pi_1 M \mid \pi_2 M$	projection
		$\mathsf{let}\,x = e\,\mathsf{in}\,M$	term binding
		$\mathsf{circ}\,L$	suspension

lax modules

L	$::=$ $\mathsf{ret}\, M$	monadic unit
	$\mid M :> \sigma$	sealing
	$\mid \mathsf{bind}\, \alpha/m \leftarrow M \,\mathsf{in}\, L$	monadic bind
	$\mid \mathsf{unpack}\, [\alpha, x] = e \,\mathsf{in}\, L$	unpack

contexts

Γ	$::=$ ϵ	empty context
	$\mid \Gamma, \alpha{:}k$	constr. hypothesis
	$\mid \Gamma, x{:}\tau$	term hypothesis
	$\mid \Gamma, \alpha/m{:}\sigma$	module hypothesis

$\boxed{\vdash \Gamma\ \mathsf{ok}}$

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash k : \mathsf{kind}}{\vdash \Gamma, \alpha{:}k\ \mathsf{ok}}$$

$$\frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \tau : \mathsf{Type}}{\vdash \Gamma, x{:}\tau\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \sigma : \mathsf{sig}}{\vdash \Gamma, \alpha/m{:}\sigma\ \mathsf{ok}}$$

$\boxed{\Gamma \vdash k : \mathsf{kind}}$

$$\frac{}{\Gamma \vdash \mathsf{Type} : \mathsf{kind}} \qquad \frac{\Gamma \vdash c : \mathsf{Type}}{\Gamma \vdash \mathsf{S}(c) : \mathsf{kind}} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 : \mathsf{kind}}$$

$\boxed{\Gamma \vdash k \equiv k' : \mathsf{kind}}$

$$\frac{\Gamma \vdash k : \mathsf{kind}}{\Gamma \vdash k \equiv k : \mathsf{kind}} \qquad \frac{\Gamma \vdash k' \equiv k : \mathsf{kind}}{\Gamma \vdash k \equiv k' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \equiv k_2 : \mathsf{kind} \quad \Gamma \vdash k_2 \equiv k_3 : \mathsf{kind}}{\Gamma \vdash k_1 \equiv k_3 : \mathsf{kind}}$$

$$\frac{\Gamma \vdash c \equiv c' : \mathsf{Type}}{\Gamma \vdash \mathsf{S}(c) \equiv \mathsf{S}(c') : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \equiv k_2' : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 \equiv \Pi\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \equiv k_2' : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 \equiv \Sigma\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$\boxed{\Gamma \vdash k \leq k' : \mathsf{kind}}$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind}}{\Gamma \vdash k \leq k' : \mathsf{kind}} \qquad \frac{\Gamma \vdash k \leq k' : \mathsf{kind} \quad \Gamma \vdash k' \leq k'' : \mathsf{kind}}{\Gamma \vdash k \leq k'' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash c : \mathsf{Type}}{\Gamma \vdash \mathsf{S}(c) \leq \mathsf{Type} : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1' \leq k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1' \vdash k_2 \leq k_2' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 : \mathsf{kind} \leq \Pi\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \leq k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \leq k_2' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1' \vdash k_2' : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 \leq \Sigma\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$\boxed{\Gamma \vdash c : k}$

$$\frac{\Gamma(\alpha) = k}{\Gamma \vdash \alpha : k}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash c : k_2}{\Gamma \vdash \lambda\alpha{:}k_1.c : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c_1 : \Pi\alpha{:}k_1.k_2 \quad \Gamma \vdash c_2 : k_1}{\Gamma \vdash c_1 c_2 : [c_2/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : [c_1/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_1 c : k_1} \qquad \frac{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_2 c : [\pi_1 c/\alpha]k_2}$$

$$\frac{}{\Gamma \vdash \star : 1} \qquad \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Type}}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{Type} \quad \Gamma \vdash \tau_2 : \mathsf{Type}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Type}}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{Type} \quad \Gamma \vdash \tau_2 : \mathsf{Type}}{\Gamma \vdash \tau_1 \times \tau_2 : \mathsf{Type}}$$

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

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{Type}}{\Gamma \vdash \exists\alpha{:}k.\tau : \mathsf{Type}}$$

$$\frac{\Gamma \vdash c : \mathsf{Type}}{\Gamma \vdash c : \mathsf{S}(c)}$$

$$\frac{\Gamma \vdash c : \Pi\alpha{:}k_1.k_2' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha : k_2}{\Gamma \vdash c : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash \pi_1 c : k_1 \quad \Gamma \vdash \pi_2 c : [\pi_1 c/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash c : k'}$$

$\boxed{\Gamma \vdash c \equiv c' : k}$

$$\frac{\Gamma \vdash c : k}{\Gamma \vdash c \equiv c : k} \qquad \frac{\Gamma \vdash c' \equiv c : k}{\Gamma \vdash c \equiv c' : k}$$

$$\frac{\Gamma \vdash c \equiv c' : k \quad \Gamma \vdash c' \equiv c'' : k}{\Gamma \vdash c \equiv c'' : k}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash c \equiv c' : k_2}{\Gamma \vdash \lambda\alpha{:}k_1.c \equiv \lambda\alpha{:}k_1'.c' : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \Pi\alpha{:}k_1.k_2 \quad \Gamma \vdash c_2 \equiv c_2' : k_1}{\Gamma \vdash c_1 c_2 \equiv c_1' c_2' : [c_2/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : k_1 \quad \Gamma \vdash c_2 \equiv c_2' : [c_1/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_1 c \equiv \pi_1 c' : k_1}$$

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_2 c \equiv \pi_2 c' : [\pi_1 c/\alpha]k_2}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \mathsf{Type} \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \mathsf{Type}}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \mathsf{Type}}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \mathsf{Type} \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \mathsf{Type}}{\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \mathsf{Type}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau \equiv \tau' : \mathsf{Type}}{\Gamma \vdash \forall\alpha{:}k.\tau \equiv \forall\alpha{:}k'.\tau' : \mathsf{Type}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau \equiv \tau' : \mathsf{Type}}{\Gamma \vdash \exists \alpha{:}k.\tau \equiv \exists \alpha{:}k'.\tau' : \mathsf{Type}}$$

$$\frac{\Gamma \vdash c \equiv c' : \mathsf{Type}}{\Gamma \vdash c \equiv c' : \mathsf{S}(c)} \qquad \frac{\Gamma \vdash c : \mathsf{S}(c')}{\Gamma \vdash c \equiv c' : \mathsf{Type}}$$

$$\frac{\Gamma \vdash c : \Pi \alpha{:}k_1.k_2' \quad \Gamma \vdash c' : \Pi \alpha{:}k_1.k_2'' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha \equiv c'\alpha : k_2}{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha \equiv c'\alpha : k_2}{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash \pi_1 c \equiv \pi_1 c' : k_1 \qquad \Gamma \vdash \pi_2 c \equiv \pi_2 c' : [\pi_1 c/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash c \equiv c' : \Sigma \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : 1 \quad \Gamma \vdash c' : 1}{\Gamma \vdash c \equiv c' : 1}$$

$$\frac{\Gamma \vdash c \equiv c' : k \quad \Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash c \equiv c' : k'}$$

$$\frac{\Gamma, \alpha{:}k_1 \vdash c_2 : k_2 \quad \Gamma \vdash c_1 : k_1}{\Gamma \vdash (\lambda \alpha{:}k_1.c_2)c_1 \equiv [c_1/\alpha]c_2 : [c_1/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_1\langle c_1, c_2\rangle \equiv c_1 : k_1}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_2\langle c_1, c_2\rangle \equiv c_2 : k_2}$$

$\boxed{\Gamma \vdash \sigma : \mathsf{sig}}$

$$\frac{}{\Gamma \vdash 1 : \mathsf{sig}} \qquad \frac{\Gamma \vdash k : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) : \mathsf{sig}} \qquad \frac{\Gamma \vdash \tau : \mathsf{Type}}{\Gamma \vdash \langle\!|\, \tau \,|\!\rangle : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi \alpha{:}\sigma_1.\sigma_2 : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 : \mathsf{sig}} \qquad \frac{\Gamma \vdash \rho : \mathsf{sig}^+}{\Gamma \vdash \bigcirc \rho : \mathsf{sig}}$$

$\boxed{\Gamma \vdash \rho : \mathsf{sig}^+}$

$$\frac{\Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash \downarrow \sigma : \mathsf{sig}^+} \qquad \frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho : \mathsf{sig}^+}{\Gamma \vdash \exists \alpha{:}k.\rho : \mathsf{sig}^+}$$

$\boxed{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}$

$$\frac{\Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash \sigma \equiv \sigma : \mathsf{sig}} \qquad \frac{\Gamma \vdash \sigma' \equiv \sigma : \mathsf{sig}}{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 : \mathsf{sig} \quad \Gamma \vdash \sigma_2 \equiv \sigma_3 : \mathsf{sig}}{\Gamma \vdash \sigma_1 \equiv \sigma_3 : \mathsf{sig}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) \equiv (\!|\, k' \,|\!) : \mathsf{sig}} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \mathsf{Type}}{\Gamma \vdash \langle\!|\, \tau \,|\!\rangle \equiv \langle\!|\, \tau' \,|\!\rangle : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \equiv \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Pi \alpha{:}\sigma_1.\sigma_2 \equiv \Pi \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \equiv \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 \equiv \Sigma \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \rho \equiv \rho' : \mathsf{sig}^+}{\Gamma \vdash \bigcirc \rho \equiv \bigcirc \rho' : \mathsf{sig}}$$

$\boxed{\Gamma \vdash \rho \equiv \rho' : \mathsf{sig}^+}$

$$\frac{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}{\Gamma \vdash \downarrow \sigma \equiv \downarrow \sigma' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho \equiv \rho' : \mathsf{sig}^+}{\Gamma \vdash \exists \alpha{:}k.\rho \equiv \exists \alpha{:}k'.\rho' : \mathsf{sig}^+}$$

$\boxed{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}$

$$\frac{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \leq \sigma_2 : \mathsf{sig} \quad \Gamma \vdash \sigma_2 \leq \sigma_3 : \mathsf{sig}}{\Gamma \vdash \sigma_1 \leq \sigma_3 : \mathsf{sig}}$$

$$\frac{\Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) \leq (\!|\, k \,|\!) : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_1 : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \leq \sigma_2' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi \alpha{:}\sigma_1.\sigma_2 \leq \Pi \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \leq \sigma_1' : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \leq \sigma_2' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 \leq \Sigma \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \rho \leq \rho' : \mathsf{sig}^+}{\Gamma \vdash \bigcirc \rho \leq \bigcirc \rho' : \mathsf{sig}}$$

$\boxed{\Gamma \vdash \rho \leq \rho' : \mathsf{sig}^+}$

$$\frac{\Gamma \vdash \rho \equiv \rho' : \mathsf{sig}^+}{\Gamma \vdash \rho \leq \rho' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash \rho_1 \leq \rho_2 : \mathsf{sig}^+ \quad \Gamma \vdash \rho_2 \leq \rho_3 : \mathsf{sig}^+}{\Gamma \vdash \rho_1 \leq \rho_3 : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}{\Gamma \vdash \downarrow \sigma \leq \downarrow \sigma' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho \leq \rho' : \mathsf{sig}^+ \quad \Gamma \vdash \rho' : \mathsf{sig}^+}{\Gamma \vdash \exists \alpha{:}k.\rho \leq \rho' : \mathsf{sig}^+}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash \rho \leq [c/\alpha]\rho' : \mathsf{sig}^+ \quad \Gamma, \alpha{:}k \vdash \rho' : \mathsf{sig}^+}{\Gamma \vdash \rho \leq \exists \alpha{:}k.\rho' : \mathsf{sig}^+}$$

$\boxed{\rho \Rightarrow \Gamma.\sigma}$

$$\frac{}{\downarrow \sigma \Rightarrow \epsilon.\sigma} \qquad \frac{\rho \Rightarrow \Gamma.\sigma}{\exists \alpha{:}k.\rho \Rightarrow \alpha{:}k, \Gamma.\sigma}$$

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2 : \tau' \quad \alpha \notin FV(\tau')}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2 : \tau'}$$

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

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

$$\frac{\Gamma \vdash L : \rho \quad \rho \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash e : \tau \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\tau) = \emptyset}{\Gamma \vdash \mathsf{let}\,\alpha/m = L\,\mathsf{in}\,e : \tau}$$

$$\frac{\Gamma \vdash M : \langle\!|\,\tau\,|\!\rangle}{\Gamma \vdash \mathsf{Ext}\,M : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{Type}}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash M : \sigma}$

$$\frac{\Gamma(m) = \sigma}{\Gamma \vdash m : \sigma} \qquad \frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\Gamma \vdash c : k}{\Gamma \vdash (\!|\,c\,|\!) : (\!|\,k\,|\!)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle\!|\,e\,|\!\rangle : \langle\!|\,\tau\,|\!\rangle}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash M : \sigma_2}{\Gamma \vdash \lambda\alpha/m{:}\sigma_1.M : \Pi\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash M_1 : \Pi\alpha{:}\sigma.\sigma' \quad \Gamma \vdash M_2 : \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash M_1 M_2 : [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma \vdash M_2 : \sigma_2 \quad \alpha \notin FV(\sigma_2)}{\Gamma \vdash \langle M_1, M_2\rangle : \Sigma\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash M : \Sigma\alpha{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_1 M : \sigma_1}$$

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

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash M : \sigma}{\Gamma \vdash \mathsf{let}\,x = e\,\mathsf{in}\,M : \sigma}$$

$$\frac{\Gamma \vdash L : \rho}{\Gamma \vdash \mathsf{circ}\,L : \bigcirc\rho}$$

$$\frac{\Gamma \vdash M : (\!|\,k'\,|\!) \quad \Gamma \vdash \mathsf{Fst}(M) \gg c \quad \Gamma \vdash c : k}{\Gamma \vdash M : (\!|\,k\,|\!)}$$

$$\frac{\Gamma \vdash M : \Pi\alpha{:}\sigma_1.\sigma_2' \quad \Gamma, \alpha/m{:}\sigma_1 \vdash M\,m : \sigma_2}{\Gamma \vdash M : \Pi\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash \pi_1 M : \sigma_1 \quad \Gamma \vdash \pi_2 M : \sigma_2 \quad \alpha \notin FV(\sigma_2)}{\Gamma \vdash M : \Sigma\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma \le \sigma' : \mathsf{sig}}{\Gamma \vdash M : \sigma'}$$

$\boxed{\Gamma \vdash L : \rho}$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{ret}\,M : \downarrow\sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M :> \sigma : \downarrow\sigma}$$

$$\frac{\Gamma \vdash M : \bigcirc\rho_1 \quad \rho_1 \Rightarrow \Gamma'.\sigma \quad \Gamma, \Gamma', \alpha/m{:}\sigma \vdash L : \rho_2 \quad \mathrm{Dom}(\Gamma', \alpha/m{:}\sigma) \cap FV(\rho_2) = \emptyset}{\Gamma \vdash \mathsf{bind}\,\alpha/m \leftarrow M\,\mathsf{in}\,L : \rho_2}$$

$$\frac{\Gamma \vdash e : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash L : \rho \quad \alpha \notin FV(\rho)}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,L : \rho}$$

$$\frac{\Gamma \vdash L : \rho \quad \Gamma \vdash \rho \le \rho' : \mathsf{sig}^+}{\Gamma \vdash L : \rho'}$$

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

$$\begin{array}{lcl}
\mathsf{Fst}(1) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathsf{Fst}(\langle\!|\,k\,|\!\rangle) & \stackrel{\mathrm{def}}{=} & k \\
\mathsf{Fst}(\langle\!|\,\tau\,|\!\rangle) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathsf{Fst}(\Pi\alpha{:}\sigma_1.\sigma_2) & \stackrel{\mathrm{def}}{=} & \Pi\alpha{:}\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2) \\
\mathsf{Fst}(\Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\mathrm{def}}{=} & \Sigma\alpha{:}\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2) \\
\mathsf{Fst}(\bigcirc\rho) & \stackrel{\mathrm{def}}{=} & 1
\end{array}$$

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

$$\frac{}{\Gamma \vdash \mathsf{Fst}(\langle\!|\,e\,|\!\rangle) \gg \star} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\mathsf{circ}\,L) \gg \star}$$

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

$$\frac{\Gamma \vdash \mathsf{Fst}(M_1) \gg c_1 \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash \mathsf{Fst}(M_1 M_2) \gg c_1 c_2}$$

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

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

C Signature Formation Algorithm

$\boxed{\Gamma \vdash \sigma \Leftarrow \mathsf{sig}}$

$$\frac{}{\Gamma \vdash 1 \Leftarrow \mathsf{sig}} \qquad \frac{\Gamma \vdash k \Leftarrow \mathsf{kind}}{\Gamma \vdash (\!|\,k\,|\!) \Leftarrow \mathsf{sig}} \qquad \frac{\Gamma \vdash \tau \Leftarrow \mathsf{Type}}{\Gamma \vdash \langle\!|\,\tau\,|\!\rangle \Leftarrow \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \Leftarrow \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \Leftarrow \mathsf{sig}}{\Gamma \vdash \Pi\alpha{:}\sigma_1.\sigma_2 \Leftarrow \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \Leftarrow \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \Leftarrow \mathsf{sig}}{\Gamma \vdash \Sigma\alpha{:}\sigma_1.\sigma_2 \Leftarrow \mathsf{sig}}$$

$$\frac{\Gamma \vdash \rho \Leftarrow \mathsf{sig}^+}{\Gamma \vdash \bigcirc\rho \Leftarrow \mathsf{sig}}$$

$\boxed{\Gamma \vdash \rho \Leftarrow \mathsf{sig}^+}$

$$\frac{\Gamma \vdash \sigma \Leftarrow \mathsf{sig}}{\Gamma \vdash \downarrow\sigma \Leftarrow \mathsf{sig}^+} \qquad \frac{\Gamma \vdash k \Leftarrow \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \rho \Leftarrow \mathsf{sig}^+}{\Gamma \vdash \exists\alpha{:}k.\rho \Leftarrow \mathsf{sig}^+}$$

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[2] Sandip K. Biswas. Higher-order functors with transparent signatures. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1995.

[3] Karl Crary. Sound and complete elimination of singleton kinds. *ACM Transactions on Computational Logic*, 8(2), April 2007. An earlier version appeared in 2000 Workshop on Types in Compilation.

[4] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *2009 Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Montreal, 2009.

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

[6] Karl Crary. Strong sums in focused logic. In *Thirty-Third IEEE Symposium on Logic in Computer Science*, Oxford, England, July 2018.

[7] Karl Crary. Fully abstract module compilation. In *Forty-Sixth ACM Symposium on Principles of Programming Languages*, Lisbon, Portugal, January 2019.

[8] Karl Crary and Robert Harper. Mechanized definition of Standard ML. Available at `www.cs.cmu.edu/~crary/papers/mldef-alpha.tar.gz`, 2009.

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

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

[11] Conal M. Elliott. *Extensions and Applications of Higher-order Unification*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1990.

[12] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1), August 1997.

[13] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.

[14] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3), 1993.

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

[16] Robert Harper. The Holy Trinity. Blog post at `existentialtype.wordpress.com/2011/03/27/the-holy-trinity/`, 2011.

[17] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

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

[19] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *TAPSOFT '87: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 250 of *Lecture Notes in Computer Science*. Springer, 1987.

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

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

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

[23] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, June 1975.

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

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

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

[27] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5), September 1996.

[28] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *1999 International Conference on Typed Lambda Calculi and Applications*, April 1999.

[29] Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *Twenty-Third IEEE Symposium on Logic in Computer Science*, 2008.

[30] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1997.

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

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

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

[34] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[35] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[36] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[37] Andreas Rossberg. 1ML — core and modules united. In *2015 ACM International Conference on Functional Programming*, Vancouver, Canada, 2015.

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

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

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

[41] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, Copenhagen, Denmark, January 2011.

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

[43] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, November 2012.

[44] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3), 2014.

[45] Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, August 2000.

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

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

[48] David Swasey, Tom Murphy, VII, Karl Crary, and Robert Harper. A separate compilation extension to standard ml. In *Workshop on ML*, 2006.

[49] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, Philadelphia, Pennsylvania, May 1996.

[50] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002. Revised May 2003.

[51] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2004. Papers from the Third International Workshop on Types for Proofs and Programs, April 2003, Torino, Italy.

[52] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008.