Program Modules*†
(DRAFT)

Robert Harper

July 1, 2026

1 Introduction

The ML modules system is a singular achievement that addresses the central question in language design in an elegant, effective manner:

> *How can the tension between modular decomposition and flexible composition of program components be resolved?*

Ideally, a program may be composed of a collection of components that are readily adaptable from their use in other programs, and that contribute to that collection new components that may themselves be used in other programs. Realistically, to borrow a phrase from Mrs Thatcher, *there is no alternative*.

The pursuit of reusability in programming has taken many twists and turns, ranging from pathetic editor templates[1] to baroque "object-oriented" methodologies,[2] but none have succeeded to do anything but impose unworkable methodological stringencies that provoke only backlash and backsliding. And yet an elegant design by MacQueen et al. (2020) has been there all along, disregarded by all but a few *cognoscenti*. It is good to review the main principles that inform MacQueen's design.

The first point, prior to any other considerations, is that

> *Modularity is a matter of types.*

Languages without types[3] are incapable of modular (de)composition, period. Why? Types specify *assumptions* needed by a client of a component, and *specifications* of what must be achieved by an implementor. Clients can proceed with their work based on these assumptions; implementors are obliged to satisfy them whenever the need arises. They can proceed independently, but they collaborate on the *what* by separating it from the *how*. Any approach to modularity that fails to recognize this is, as many have proved to be, doomed to irrelevancy.

The second point, elegantly formulated in MacQueen's design, is that

> *Modules are never completely independent; rather, they must cohere to form a useful whole.*

†Special thanks to Runming Li for many helpful discussions and suggestions. Thanks also to Harrison Grodin, David MacQueen, and Jon Sterling for their comments and suggestions.

[1] Because they have no properties.

[2] Which also have no properties.

[3] More accurately, with only one type.

The lexer and the parser in a compiler must share a common notion of symbol as the coin of their realm. The layers of a network protocol must share a common understanding of a packet as consisting of a header and a payload, one layer's header being part of another's payload. The keys of one dictionary may well be the stored values of another; they cannot be combined unless they agree on the type of keys. The central insight expressed in MacQueen's design is that

> *Coherence is a relationship, not a construction: two components may share a type without knowing what type they share.*

The coherence of a collection of modules does not require that they all know each other's business; it requires only that critical pathways coincide. An interpreter need not know what is the type of symbols shared by the lexer and parser, only that they share a common notion, whatever it may be. A client of two dictionaries may require that they keys of one be the values of another, but need not know what that type is. A component may require *that* others agree on some type information without being able to say *what* they agree on.

Most approaches to modularity, such as they are, achieve coherence *analytically* by construction, explicitly providing the common parts to components that must share them. This has two grave implications, both of which conflict with modularity. First, *the lowest-level components in a design must be given the highest-level scope and visibility so that they can be threaded through the remaining parts of the program that uses them.* The program is "turned inside-out" by this demand, the opposite of what is required for incremental development. Second, *all components must be parameterized by any aspect of their design that might in some future scenario be required to be shared with another component.* Needless to say, such anticipatory development is practically impossible.

MacQueen's design, contrarily, achieves coherence *synthetically* by the *post hoc* imposition of *sharing requirements* on any aspects of any modules in a composite. If values of one dictionary must be the keys of another (so that one can insert in the one using the result of a lookup in the other), *simply say so* by an equation that demands the two types be the same. *This can be achieved without knowing what those types are.* Thus, neither must the program structure be everted, nor must the components anticipate all possible future relationships that might one day be required to hold.

It is a simple, elegant design that has been studiously ignored by the software industry for decades, to its own detriment. Once the absurdities of "object-oriented" methods became too obvious to ignore, the reactionary move was to abandon any form of composition whatsoever in favor of uni-typed, so-called "dynamic," languages. But neither are types incompatible with interactive implementation (as has been amply evident in ML since the 1970's), nor are they incompatible with efficient compilation (known otherwise since the 1990's), nor do they reduce the expressive power of a language (the exact opposite is true), they are, rather, of the essence for achieving modular program development.

2 Overview

There is much too much to be said about the design of a practical program module system than can possibly be reproduced here, but some remarks are in order by way of motivation for what follows. A realistic programming language has a layered structure facilitating convenient usage and maintenance of code, much of which is ignored in these studies. The most obvious is parsing concrete into (first-order) abstract syntax, but more significantly the process of *elaboration* of the external language to the internal type theory that captures its essential semantic structures. For example, elaboration of SML code includes these transformations:

1. Resolving the binding and scope of identifiers into projections from, and polymorphic instantiation of, variables

2. Expanding derived forms in terms of primitive notions of an underlying type theory.

3. Inferring omitted type information from context, and representing polymorphic generalization via modules.

4. Defining concrete datatypes and pattern matching in terms of recursive sums and modules.

5. Implementing coercive signature matching as functions that rearrange and project from modules.

There is no clear dividing line between these concepts, or between what is regarded as primitive or derived; the choices made here are intended to isolate the semantic structure of the language in as simple a form as possible for analysis.

The result of elaboration is a program written in a dependent type theory of modules and signatures with these features:

1. A distinction between *pure expressions* and *impure computations*, expressed using a *lax* modality classifying computations that may have an *effect* when executed such as divergence or mutation of a persistent memory.

2. A *universe* of *small types* that classify values and computations of the "core" language, all of which have simple (non-dependent) types.

3. A *phase distinction* isolating the *static* components of a module by collapsing the dynamic components to a point in the static phase.

4. *Extension signatures* that specify the static component of a module to be as given, with explicit coercions into and out of the extension.

5. Closure of signatures under *dependent* product and (generative) function types and their associated constructs.

Thus, the architecture of a language is given in terms modules and signatures, with core language constructs arising as elements of a universe of small types. Experience has show that, contrarily to common practices, which invariably attempt to slap on some sort of modularity mechanism after-the-fact, *a proper language design starts with modules, which is then fleshed out with core programming concepts.*

The remainder of this note specifies the type theory of modules in a manner that is compatible with extension by computational constructs, including partiality and other effects, that are not considered here.

3 Statics and Dynamics

The following forms of judgments are indexed by a *phase*, ϕ, which may be either $\mathtt{st}$, indicating the static phase, or $\top$, indicating the default phase:

1. $\vdash^{\phi} \Gamma\ \mathsf{ctx}$: a valid context Γ in phase ϕ declares variables with their signatures, with the meaning that variables range over *valu(abl)es* of that signature.

2. $\Gamma \vdash^{\phi} A\ \mathsf{sig}$ and $\Gamma \vdash^{\phi} A \equiv A'\ \mathsf{sig}$: classifiers are *signatures*, A, in phase ϕ that may involve variables from the given context; equality of signatures is influenced by the context and the phase.

3. $\Gamma \vdash^{\phi} M : A$ and $\Gamma \vdash^{\phi} M \equiv M' : A$: elements are modules M inhabiting signatures, A, relative to a context Γ, and are deemed equal as such.

4. $\Gamma \vdash^{\phi} E \div A$ and $\Gamma \vdash^{\phi} E \equiv E' \div A$: computations E returning modules of signature A, relative to a context Γ, and are deemed equal as such.

As a special case, the meta-variable τ will range over elements of the universe, and the meta-variable e will range over elements of those types.

The statics and judgmental equality are defined in Figures 4 through 8. The structural rules in Figure 4 specify that the typing and equality judgments respect equality of the classifier, characteristically for a dependent type theory. Omitted from that figure are rules specifying that judgmental equality of signatures and modules is reflexive, symmetric, and transitive, and that it is compatible with all constructs of the language, which is to say that it is a congruence. Figure 5 defines the forms of signature in the language. The signature $\mathtt{Type}$ classifies the "small" types, which serve as indices to signatures $\mathtt{Val}(\tau)$ of values of that type. The type structure is intended to be illustrative, rather than comprehensive; it includes products and functions and computations, which are defined in terms of the same module-level concepts 6 and includes a type of natural numbers given in Figure 7. The natural numbers type illustrates their limited universal properties, namely that the recursor inhabits only small types, and may not be used to define a module *per se*.[4] Figures 10 and 11 provide the dependent product and (generative) function signatures used to classify module hierarchies and functions acting on modules. Module computations (Figure 9) are specified in the "lax" style, with no particular effects included for the sake of simplicity. Finally, extension signatures (Figure 8) are used to constrain the "static" aspects of a module to be as specified: the signature $\mathtt{Ext}(A;M)$ classifies modules of signature A that, in the static phase, are judgmentally equal to M. The influence of the $\mathtt{st}$ phase is to collapse all value modules and all module computations to a point, thereby eliminating their influence on equality of two modules—they are equal in the static phase exactly when their static aspects (types) are equal. Because some types and signatures may be empty, the static phase introduces an explicit "center" to which all elements are equated in the static phase.

Lemma 1 (Static Anti-Monotonicity)**.** *Suppose that $\psi \vdash \phi$.*

1. *If $\vdash^{\phi} \Gamma\ \mathit{ctx}$, then $\vdash^{\psi} \Gamma\ \mathit{ctx}$.*

2. *If $\Gamma \vdash^{\phi} A\ \mathit{sig}$ then $\Gamma \vdash^{\psi} A\ \mathit{sig}$, and if $\Gamma \vdash^{\phi} A \equiv A'\ \mathit{sig}$, then $\Gamma \vdash^{\psi} A \equiv A'\ \mathit{sig}$.*

3. *If $\Gamma \vdash^{\phi} M : A$, then $\Gamma \vdash^{\psi} M : A$, and if $\Gamma \vdash^{\phi} M \equiv M' : A$, then $\Gamma \vdash^{\psi} M \equiv M' : A$.*

4. *If $\Gamma \vdash^{\phi} E \div A$, then $\Gamma \vdash^{\psi} E \div A$, and if $\Gamma \vdash^{\phi} E \equiv E' \div A$, then $\Gamma \vdash^{\psi} E \equiv E' \div A$.*

At this, the initial, stage of development, the only non-trivial entailment is $\mathtt{st} \vdash \top$, and the lemma states that what holds in general, also holds in the static phase. However, it will be arranged that there are strictly more equations derivable in the static phase than otherwise; moreover, once additional phases are included, such as the behavioral phase used to exclude cost in Harper (2024), the lemma

[4] Such further extensions are possible, for example a module conditional on a run-time condition may be useful for, say, system initialization purposes.

has further implications. In fact the lemma serves more to record an invariant holding of all possible extensions of the language with phases.

The dynamics of modules is unremarkable, given the lax framework and the largely standard type structure. Variables range over elements of types, which are tantamount to values, and substitution is defined only for such values. Computations sequence execution, to allow for effects to be added later. Signatures must be evaluated to ensure the proper interpretation of small product, function, and computation types. Thus, the dynamics consists of these assertions:

1. Evaluation of signatures: $A \Downarrow B$ with B val.
2. Evaluation of closed module expressions: $M \Downarrow V$ with V val.
3. Execution of closed module computations: $E \longmapsto E'$ with $\mathtt{ret}(V)$ final when V val.

Signature evaluation simplifies all closed types of the form $\mathtt{Val}(\tau)$ as follows:[5]

- $\mathtt{Val}(\mathtt{unit}) \Downarrow \mathtt{Unit}$
- $\mathtt{Val}(\tau_1 \times \tau_2) \Downarrow \mathtt{Val}(\tau_1) \times \mathtt{Val}(\tau_2)$
- $\mathtt{Val}(\tau_1 \to \tau_2) \Downarrow \mathtt{Val}(\tau_1) \to \mathtt{Val}(\tau_2)$
- $\mathtt{Val}(\mathtt{comp}(\tau)) \Downarrow \mathtt{Comp}(\mathtt{Val}(\tau))$

Thus, the only remaining closed value signature, $\mathtt{Val}(\mathtt{nat})$, is defined directly with a recursor that computes values of another small type.

Module evaluation is as in Harper (2025a), and expresses head reduction of closed expressions. Module computation is formulated as a transition system on closed expressions, but its form would change according to the specific effects, such as charging for cost.

4 Extensionality

Signatures and modules are interpreted extensionally as outlined in Harper (2025a), albeit using phases as Kripke worlds to account for phase-specific requirements. The need for extensional equality of signatures arises from extension signatures, $\mathtt{Ext}(A\,;M)$, which are deemed equal whenever their module components are (statically) equal.

The definitions of these judgments are given in Figure 1. These definitions make use of the following shorthand notations:

- $x : A \gg^{\phi} B \doteq B'$ sig iff if $M \doteq M' \in A\,[\phi]$ then $[M/x]B \doteq [M'/x]B'$ sig $[\phi]$.
- $x : A \gg^{\phi} E \doteq E' \mathrel{\widetilde{\in}} B$ iff if $M \doteq M' \in A\,[\phi]$ then $[M/x]E \doteq [M/x]E' \mathrel{\widetilde{\in}} [M/x]B\,[\phi]$.

Lemma 2 (Anti-Monotonicity)**.** *Suppose that $\psi \vdash \phi$.*

- *If $A \doteq A'$ sig $[\phi]$, then $A \doteq A'$ sig $[\psi]$.*

[5] It would also be possible to "eagerly" evaluate these constructs, without changing the meaning.

- The signature of small types:
  - If $A \Downarrow \mathtt{Type}$ and $A' \Downarrow \mathtt{Type}$, then $A \doteq A' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Type}$, then $\tau \doteq \tau' \in A\,[\phi]$ if
    * $\tau, \tau' \Downarrow \mathtt{nat}$; or $\tau, \tau' \Downarrow \mathtt{unit}$; or
    * $\tau \Downarrow \mathtt{comp}(\sigma)$ and $\tau' \Downarrow \mathtt{comp}(\sigma')$ with $\sigma \doteq \sigma' \in \mathtt{Type}\,[\phi]$; or
    * $\tau \Downarrow \tau_1 \times \tau_2$ and $\tau' \Downarrow \tau_1' \times \tau_2'$ with $\tau_1 \doteq \tau_1' \in \mathtt{Type}\,[\phi]$ and $\tau_2 \doteq \tau_2' \in \mathtt{Type}\,[\phi]$; or
    * $\tau \Downarrow \tau_1 \to \tau_2$ and $\tau' \Downarrow \tau_1' \to \tau_2'$ with $\tau_1 \doteq \tau_1' \in \mathtt{Type}\,[\phi]$ and $\tau_2 \doteq \tau_2' \in \mathtt{Type}\,[\phi]$.
- The signature of values of natural number type:
  - If $A, A' \Downarrow \mathtt{Val}(\mathtt{nat})$ then $A \doteq A' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Val}(\mathtt{nat})$, then $M \doteq M' \in A\,[\phi]$ iff $\phi \vdash \mathtt{st}$ or
    * $M, M' \Downarrow \mathtt{zero}$; or
    * $M \Downarrow \mathtt{succ}(N)$ and $M' \Downarrow \mathtt{succ}(N')$ with $N \doteq N' \in \mathtt{Val}(\mathtt{nat})\,[\phi]$.
- The signature of suspended module computations:
  - If $A \Downarrow \mathtt{Comp}(B)$ and $A' \Downarrow \mathtt{Comp}(B')$ then $A \doteq A' \,\mathsf{sig}\,[\phi]$ iff $B \doteq B' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Comp}(B)$ then $M \doteq M' \in A\,[\phi]$ iff $\phi \vdash \mathtt{st}$ or $M \Downarrow \mathtt{comp}(E)$ and $M' \Downarrow \mathtt{comp}(E')$ and $E \doteq E' \mathrel{\widetilde{\in}} B\,[\phi]$.
- The signature of modules statically equal to a given module:
  - If $A \Downarrow \mathtt{Ext}(B\,;N)$ and $A' \Downarrow \mathtt{Ext}(B'\,;N')$ then $A \doteq A' \,\mathsf{sig}\,[\phi]$. iff $B \doteq B' \,\mathsf{sig}\,[\phi]$ and $N \doteq N' \in B\,[\mathtt{st}]$.
  - If $A \Downarrow \mathtt{Ext}(B\,;P)$, then $M \doteq M' \in A\,[\phi]$ iff $M \Downarrow \mathtt{is}(N)$ and $M' \Downarrow \mathtt{is}(N')$ and $N \doteq P \in B\,[\mathtt{st}]$ and $P \doteq N' \in B\,[\mathtt{st}]$ and $N \doteq N' \in B\,[\phi]$.
- The unit signature:
  - If $A, A' \Downarrow \mathtt{Unit}$, then $A \doteq A' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Unit}$, then $M \doteq M' \in A\,[\phi]$ iff either $\phi \vdash \mathtt{st}$ or $M, M' \Downarrow \langle\rangle$.
- The signature of substructures:
  - If $A \Downarrow \mathtt{Str}(A_1\,; x.A_2)$ and $A' \Downarrow \mathtt{Str}(A_1'\,; x.A_2')$ then $A \doteq A' \,\mathsf{sig}\,[\phi]$ iff $A_1 \doteq A_1' \,\mathsf{sig}\,[\phi]$ and $x : A_1 \gg^{\phi} A_2 \doteq A_2' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Str}(A_1\,; x.A_2)$, then $M \doteq M' \in A\,[\phi]$ iff $M \cdot \mathtt{1} \doteq M' \cdot \mathtt{1} \in A_1\,[\phi]$ and $M \cdot \mathtt{2} \doteq M' \cdot \mathtt{2} \in [M \cdot \mathtt{1}/x]A_2\,[\phi]$.
- The signature of (generative) functors:
  - If $A \Downarrow \mathtt{Fun}(A_1\,; x.A_2)$ and $A' \Downarrow \mathtt{Fun}(A_1'\,; x.A_2')$ then $A \doteq A' \,\mathsf{sig}\,[\phi]$ iff $A_1 \doteq A_1' \,\mathsf{sig}\,[\phi]$ and $x : A_1 \gg^{\phi} A_2 \doteq A_2' \,\mathsf{sig}\,[\phi]$.
  - If $A \Downarrow \mathtt{Fun}(A_1\,; x.A_2)$, then $M \doteq M' \in A\,[\phi]$ iff for all $\psi \vdash \phi$, $x : A_1 \gg^{\psi} \mathtt{app}(M\,; x) \doteq \mathtt{app}(M'\,; x) \mathrel{\widetilde{\in}} A_2$.
- Module computations:
  - $E \doteq E' \mathrel{\widetilde{\in}} A\,[\phi]$ iff $\phi \vdash \mathtt{st}$ or $E \longmapsto^{*} \mathtt{ret}(M)$ and $E' \longmapsto^{*} \mathtt{ret}(M')$ and $M \doteq M' \in A\,[\phi]$.

Figure 1: Extensional Equality of Signatures and Modules

- *If* $M \doteq M' \in A\,[\phi]$*, then* $M \doteq M' \in A\,[\psi]$.
- *If* $E \doteq E' \mathrel{\widetilde{\in}} A\,[\phi]$*, then* $E \doteq E' \mathrel{\widetilde{\in}} A\,[\psi]$.

By thinking of contexts Γ as iterated closed product types, and substitutions $\vdash \gamma : \Gamma$ as dependently typed tuples of terms of that type, these definitions extend to extensional context equality, $\Gamma \doteq \Gamma\,\mathsf{sig}\,[\phi]$, and extensional substitution equality, $\gamma \doteq \gamma' \in \Gamma\,[\phi]$, which are in turn used to define hypothetical extensional equality judgments.

Definition 3.

- $\Gamma \gg^{\phi} A \doteq A'$ *sig, presupposing* $\Gamma \doteq \Gamma$ *sig* $[\phi]$*, means that if* $\gamma \doteq \gamma' \in \Gamma\,[\phi]$*, then* $\widehat{\gamma}A \doteq \widehat{\gamma'}A'$ *sig* $[\phi]$.
- $\Gamma \gg^{\phi} M \doteq M' \in A$*, presupposing* $\Gamma \gg^{\phi} A \doteq A'$ *sig, means that if* $\gamma \doteq \gamma' \in \Gamma\,[\phi]$*, then* $\widehat{\gamma}M \doteq \widehat{\gamma'}M' \in \widehat{\gamma}A\,[\phi]$.
- $\Gamma \gg^{\phi} E \doteq E' \mathrel{\widetilde{\in}} A$*, presupposing* $\Gamma \gg^{\phi} A \doteq A'$ *sig, means that if* $\gamma \doteq \gamma' \in \Gamma\,[\phi]$*, then* $\widehat{\gamma}E \doteq \widehat{\gamma'}E' \mathrel{\widetilde{\in}} \widehat{\gamma}A\,[\phi]$.

These coincide with the definitions given in Figure 1 when Γ is empty, and agree with the *ad hoc* forms of hypothetical judgment used therein.

Theorem 4 (Fundamental Theorem)**.**

- *If* $\Gamma \vdash^{\phi} A$ *sig then* $\Gamma \gg^{\phi} A \doteq A$ *sig, and if* $\Gamma \vdash^{\phi} A \equiv A'$ *sig then* $\Gamma \gg A \doteq A'$ *sig.*
- *If* $\Gamma \vdash^{\phi} M : A$ *then* $\Gamma \gg^{\phi} M \doteq M \in A$*, and if* $\Gamma \vdash^{\phi} M \equiv M' : A$ *then* $\Gamma \gg^{\phi} M \doteq M' \in A$.
- *If* $\Gamma \vdash^{\phi} E \div A$ *then* $\Gamma \gg^{\phi} E \doteq E \mathrel{\widetilde{\in}} A$*, and if* $\Gamma \vdash^{\phi} E \equiv E' \div A$ *then* $\Gamma \gg^{\phi} E \doteq E' \mathrel{\widetilde{\in}} A$.

The fundamental theorem implies that all three forms of extensional equality are symmetric and transitive, and that they are all reflexive on well-typed instances. Moreover, it ensures that extensionally equal signatures classify the same modules and computations.

5 Parametricity

The abstract meaning of a signature or module is given by its extensional equivalence class, so that, for example, any two formulations of the doubling function on the natural numbers are regarded as having the same meaning because they are extensionally equivalent at their type. The treatment of parametricity is based on binary logical relations between extensional equivalence classes of modules, generalizing the formulation given by Reynolds (Harper, 2025b, 2026) from relating values of possibly disparate types to relating modules of disparate signatures that contain not only values, but also types, computations of modules, hierarchies of modules, and functional transformations of modules. Being a matter of correctness of executable code, parametricity makes no use of phases itself—it is formulated at the "general" phase, treating signatures and modules extensionally in the general phase. Just as in Reynolds's formulation, extensional equality will be a (homogeneous) parametricity relation on values of a type. Because the type theory of modules includes a universe of "small" types, the full-scale treatment of parametricity requires consideration of *(binary) type families*, which classify *evidence* for the "truth" of their instances, with relations being the special case in which the evidence is immaterial beyond its existence.

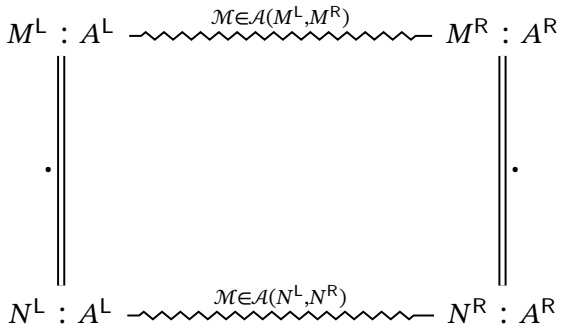

Figure 2: Heterogeneous Relations Respect Homogeneous Equality

The diagram in Figure 2 illustrates the overall architecture, with the horizontal direction representing heterogeneous relations between modules of disparate signatures, and the vertical direction representing homogeneous extensional equivalences between modules that are to be respected by those relations. Note that extensional equivalence between modules of a given signature is itself one such relation, exactly because it is symmetric and transitive—the path from top left to top right by descending vertically, crossing horizontally, and ascending vertically along extensional equalities results in an extensional equality.

The formulation of parametricity families for modules is given in Figure 3. It defines a collection of *correspondences* between two signatures, each of the same outermost form, and defines the *canonical evidence* for each of these correspondences by populating instances of these families. The source of nontrivial evidence for families is precisely the binary relations between the values of two small types that are at most true of any values of these types, exactly as in Reynolds' formulation. These pieces of evidence are organized into structures of evidence for the correspondences of structures of modules, and into functions on evidence for correspondences between functors acting on modules. Correspondences between extension signatures, which specify static equational constraints on modules, are regarded as correspondences between their underlying signatures, the constraints having no bearing on what is fundamentally a relationship between their dynamic components. Encapsulated computations are related based on their returned values, there being no effects in their formulation in this skeletal setup. Bearing in mind that the formation of correspondences, and the correspondences themselves, are defined on *extensional equivalence classes (in the general phase)* of signatures, module values, and module computations as defined in Section 4, which has the effect of affording very simple definitions of these notions.

The definitions in Figure 3 make use of two special forms of hypothetical judgment, defined as follows:

- $\chi : \mathcal{A}_1(x_{\mathsf{L}}, x_{\mathsf{R}}) \gg \mathcal{A}_2 \in \textsc{sig}(A_2^{\mathsf{L}}, A_2^{\mathsf{R}})$ iff $\mathcal{M}_1 \in \mathcal{A}_1(M_1^{\mathsf{L}}, M_1^{\mathsf{R}})$ implies
$$[\mathcal{M}_1/\chi]\mathcal{A}_2 \in \textsc{sig}([M_1^{\mathsf{L}}/x_{\mathsf{L}}]A_2^{\mathsf{L}}, [M_1^{\mathsf{R}}/x_{\mathsf{R}}]A_2^{\mathsf{R}}).$$
- $\chi : \mathcal{A}_1(x_{\mathsf{L}}, x_{\mathsf{R}}) \gg \mathcal{M}_2 \in \mathcal{A}_2(M_2^{\mathsf{L}}, M_2^{\mathsf{R}})$ iff $\mathcal{M}_1 \in \mathcal{A}_1(M_1^{\mathsf{L}}, M_1^{\mathsf{R}})$ implies
$$[\mathcal{M}_1/\chi]\mathcal{M}_2 \in ([\mathcal{M}_1/\chi]\mathcal{A}_2)([M_1^{\mathsf{L}}/x_{\mathsf{L}}]M_2^{\mathsf{L}}, [M_1^{\mathsf{R}}/x_{\mathsf{R}}]M_2^{\mathsf{R}}).$$

- Small types correspond without restriction:[6]
  - $\textsc{typ} \in \textsc{sig}(\mathtt{Type}, \mathtt{Type})$
  - $\mathcal{R} \in \textsc{typ}(\tau_{\mathsf{L}}, \tau_{\mathsf{R}})$ iff $\mathcal{R} : \mathtt{Val}(\tau_{\mathsf{L}})/\dot{=} \leftrightarrow \mathtt{Val}(\tau_{\mathsf{R}})/\dot{=}$
- Relations between small types induce correspondences between their values:
  - $\textsc{val}[\mathcal{R}] \in \textsc{sig}(\mathtt{Val}(\tau_{\mathsf{L}}), \mathtt{Val}(\tau_{\mathsf{R}}))$ iff $\mathcal{R} \in \textsc{typ}(\tau_{\mathsf{L}}, \tau_{\mathsf{R}})$
  - $\textsc{thus} \in \textsc{val}[\mathcal{R}](M_{\mathsf{L}}, M_{\mathsf{R}})$ iff $\mathcal{R}(M_{\mathsf{L}}, M_{\mathsf{R}})$ true
- Correspondences on a signature induce correspondences between their computations:
  - $\textsc{comp}[\mathcal{A}] \in \textsc{sig}(\mathtt{Comp}(A_{\mathsf{L}}), \mathtt{Comp}(A_{\mathsf{R}}))$ iff $\mathcal{A} \in \textsc{sig}(A_{\mathsf{L}}, A_{\mathsf{R}})$
  - $\textsc{susp}(\mathcal{E}) \in \textsc{comp}[\mathcal{A}](\mathtt{comp}(M_{\mathsf{L}}), \mathtt{comp}(M_{\mathsf{R}}))$ iff $\mathcal{E} \in \textsc{do}[\mathcal{A}](E_{\mathsf{L}}, E_{\mathsf{R}})$
  - $\textsc{ret}(\mathcal{M}) \in \textsc{do}[\mathcal{A}](\mathtt{ret}(M_{\mathsf{L}}), \mathtt{ret}(M_{\mathsf{R}}))$ iff $\mathcal{M} \in \mathcal{A}(M_{\mathsf{L}}, M_{\mathsf{R}})$
- Extension signatures disregard static constraints:[7]
  - $\textsc{ext}[\mathcal{A}] \in \textsc{sig}(\mathtt{Ext}(A_{\mathsf{L}}\,; M_{\mathsf{L}}), \mathtt{Ext}(A_{\mathsf{R}}\,; M_{\mathsf{R}}))$ iff $\mathcal{A} \in \textsc{sig}(A_{\mathsf{L}}, A_{\mathsf{R}})$
  - $\textsc{is}(\mathcal{M}) \in \textsc{ext}[\mathcal{A}](\mathtt{is}(N_{\mathsf{L}}), \mathtt{is}(N_{\mathsf{R}}))$ iff $\mathcal{M} \in \mathcal{A}(N_{\mathsf{L}}, N_{\mathsf{R}})$
- The unit signature admits only the identity correspondence:
  - $\textsc{unit} \in \textsc{sig}(\mathtt{Unit}, \mathtt{Unit})$
  - $\textsc{triv} \in \textsc{unit}(\mathtt{triv}, \mathtt{triv})$
- Structure signatures admit structures of correspondences:
  - $\textsc{str}[\mathcal{A}_1\,; \chi.\mathcal{A}_2] \in \textsc{sig}(\mathtt{Str}(A_1^{\mathsf{L}}\,; x_{\mathsf{L}}.A_2^{\mathsf{L}}), \mathtt{Str}(A_1^{\mathsf{R}}\,; x_{\mathsf{R}}.A_2^{\mathsf{R}}))$ iff $\mathcal{A}_1 \in \textsc{sig}(A_1^{\mathsf{L}}, A_1^{\mathsf{R}})$ and $\chi : \mathcal{A}_1(x_{\mathsf{L}}, x_{\mathsf{R}}) \gg \mathcal{A}_2 \in \textsc{sig}(A_2^{\mathsf{L}}, A_2^{\mathsf{R}})$
  - $\textsc{pair}(\mathcal{M}_1\,; \mathcal{M}_2) \in \textsc{str}[\mathcal{A}_1\,; \chi.\mathcal{A}_2](\mathtt{str}(M_1^{\mathsf{L}}\,; M_2^{\mathsf{L}}), \mathtt{str}(M_1^{\mathsf{R}}\,; M_2^{\mathsf{R}}))$ iff $\mathcal{M}_1 \in \mathcal{A}_1(M_1^{\mathsf{L}}, M_1^{\mathsf{R}})$ and $\mathcal{M}_2 \in [\mathcal{M}_1/\chi]\mathcal{A}_2(M_2^{\mathsf{L}}, M_2^{\mathsf{R}})$
- Functor signatures admit functions of correspondences:
  - $\textsc{fun}[\mathcal{A}_1\,; \chi.\mathcal{A}_2] \in \textsc{sig}(\mathtt{Fun}(A_1^{\mathsf{L}}\,; x_{\mathsf{L}}.A_2^{\mathsf{L}}), \mathtt{Fun}(A_1^{\mathsf{R}}\,; x_{\mathsf{R}}.A_2^{\mathsf{R}}))$ iff $\mathcal{A}_1 \in \textsc{sig}(A_1^{\mathsf{L}}, A_1^{\mathsf{R}})$ and $\chi : \mathcal{A}_1(x_{\mathsf{L}}, x_{\mathsf{R}}) \gg \mathcal{A}_2 \in \textsc{sig}(A_2^{\mathsf{L}}, A_2^{\mathsf{R}})$
  - $\textsc{lam}(\chi.\mathcal{E}_2) \in \textsc{fun}[\mathcal{A}_1\,; \chi.\mathcal{A}_2](\mathtt{fun}(x_{\mathsf{L}}.E_2^{\mathsf{L}}), \mathtt{fun}(x_{\mathsf{R}}.E_2^{\mathsf{R}}))$ iff $\chi : \mathcal{A}_1(x_{\mathsf{L}}, x_{\mathsf{R}}) \gg \mathcal{E}_2 \in \textsc{do}[\mathcal{A}_2](E_2^{\mathsf{L}}, E_2^{\mathsf{R}})$

Figure 3: Correspondences and Their Canonical Evidence

Example 1 (Queues, Two Ways)**.** *Define* `QUEUE` *to be the signature*

$$q : \mathtt{Type} \times \mathtt{Val}(q) \times \mathtt{Val}(\mathtt{nat} \to q \to q) \times \mathtt{Val}(q \to \mathtt{opt}(\mathtt{nat} \times q)).$$

The semantics gives rise to $\mathcal{Q} \in \textsc{sig}(\mathtt{QUEUE}, \mathtt{QUEUE})\ [\phi]$ *for any phase* ϕ *given by*

$$\mathcal{Q} \triangleq \textsc{str}[\textsc{typ}\ ; \chi.\, \textsc{str}[\mathcal{A}_{emp}\ ; \textsc{str}[\mathcal{A}_{ins}\ ; \mathcal{A}_{rem}]]],$$

where

$$\begin{aligned}
\mathcal{A}_{emp} &= \textsc{val}[\chi] \\
\mathcal{A}_{ins} &= \textsc{fun}[\textsc{val}[\textsc{Nat}]\ ; \textsc{fun}[\textsc{val}[\chi]\ ; \textsc{val}[\chi]]] \\
\mathcal{A}_{rem} &= \textsc{fun}[\textsc{val}[\chi]\ ; \textsc{opt}[\textsc{str}[\textsc{val}[\textsc{Nat}]\ ; \textsc{val}[\chi]]]].^{8}
\end{aligned}$$

The implementation Q_L : `QUEUE` *of queues as lists is given by the module*

$$\mathtt{str}(\mathtt{list}(\mathtt{nat})\,; \mathtt{str}(emp_L\,; \mathtt{str}(ins_L\,; rem_L))),$$

where emp_L*,* ins_L*, and* rem_L *are the "list-based" implementations of the empty, insert, and remove operations specified in* `QUEUE`*. Similarly, the implementation* Q_R : `QUEUE` *of queues as pairs of lists is given by the module*

$$\mathtt{str}(\mathtt{list}(\mathtt{nat}) \times \mathtt{list}(\mathtt{nat})\,; \mathtt{str}(emp_R\,; \mathtt{str}(ins_R\,; rem_R))),$$

where emp_R*,* ins_R*, and* rem_R *are the "pair of lists" implementations of the same queue operations.*

Now let

$$\mathcal{R} \in \textsc{typ}(\mathtt{list}(\mathtt{nat}), \mathtt{list}(\mathtt{nat}) \times \mathtt{list}(\mathtt{nat}))$$

be the candidate relation between the two implementation types such that $\mathcal{R}(xs, \langle bs, fs\rangle)$ true *whenever* xs *is the concatenation of* bs *with the reversal of* fs*. This gives rise to evidence for the correspondence of the two implementations given by*

$$\textsc{pair}(\mathcal{R}\,; \textsc{pair}(\pi_{emp}\,; \textsc{pair}(\pi_{ins}\,; \pi_{rem}))) \in \mathcal{Q}(Q_L, Q_R)\ [\phi],$$

where

$$\begin{aligned}
\pi_{emp} &\in [\mathcal{R}/\chi]\mathcal{A}_{emp}(emp_L, emp_R)\ [\phi] \\
\pi_{ins} &\in [\mathcal{R}/\chi]\mathcal{A}_{ins}(ins_L, ins_R)\ [\phi] \\
\pi_{rem} &\in [\mathcal{R}/\chi]\mathcal{A}_{rem}(rem_L, rem_R)\ [\phi],
\end{aligned}$$

in accordance with the semantics of dependent products. After substitution of $\mathcal{R}$ *for* χ*, and expanding definitions, these requirements may be restated as*

$$\begin{aligned}
\pi_{emp} &\in \textsc{val}[\mathcal{R}](emp_L, emp_R)\ [\phi] \\
\pi_{ins} &\in \textsc{fun}[\textsc{val}[\textsc{Nat}]\ ; \textsc{fun}[\textsc{val}[\mathcal{R}]\ ; \textsc{val}[\mathcal{R}]]](ins_L, ins_R)\ [\phi] \\
\pi_{rem} &\in \textsc{fun}[\textsc{val}[\mathcal{R}]\ ; \textsc{opt}[\textsc{str}[\textsc{val}[\textsc{Nat}]\ ; \textsc{val}[\mathcal{R}]]]](rem_L, rem_R)\ [\phi].
\end{aligned}$$

For example, because $\textsc{refl} \in \mathcal{R}(\mathtt{nil}, \langle \mathtt{nil}, \mathtt{nil}\rangle)$*, it follows that* $\pi_{emp} = \textsc{thus} \in \textsc{val}[\mathcal{R}](emp_L, emp_R)$*. The other operations are handled similarly, thereby showing that the two implementations of queues are "co-correct," equally right or equally wrong, ensuring that no client can distinguish them.*

[8] The definition of $\textsc{opt}[\mathcal{A}]$ correspondng to the option type is omitted here.

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Cost effects and phases. Unpublished lecture note., January 2024. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/cost.pdf.

Robert Harper. Dependent types for programming and proving. (Unpublished lecture note), April 2025a. URL https://www.cs.cmu.edu/~rwh/courses/atpl/notes/dependency.pdf.

Robert Harper. Reynolds's parametricity theorem, directly. Unpublished lecture note, January 2025b. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/reynolds.pdf.

Robert Harper. Tait computability structures. Unpublished lecture note., February 2025c. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/tait-structures.pdf.

Robert Harper. Variables types for genericity and abstraction. Unpublished lecture note, February 2026. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/variabletypes.pdf.

David MacQueen, Robert Harper, and John Reppy. The history of Standard ML. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386336. URL https://doi.org/10.1145/3386336.

Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *J. ACM*, 68(6), October 2021. ISSN 0004-5411. doi: 10.1145/3474834. URL https://doi.org/10.1145/3474834.

EMP-CTX
$$\frac{}{\vdash^{\phi} \varepsilon\ \mathsf{ctx}}$$

EXT-CTX
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx} \qquad \Gamma \vdash^{\phi} A\ \mathsf{sig}}{\vdash^{\phi} \Gamma, x : A\ \mathsf{ctx}}$$

VAR
$$\frac{\Gamma \vdash^{\phi} A\ \mathsf{sig}}{\Gamma, x : A \vdash^{\phi} x : A}$$

OF-EQ
$$\frac{\Gamma \vdash^{\phi} M : A' \qquad \Gamma \vdash^{\phi} A' \equiv A\ \mathsf{sig}}{\Gamma \vdash^{\phi} M : A}$$

EQ-EQ
$$\frac{\Gamma \vdash^{\phi} M \equiv M' : A' \qquad \Gamma \vdash^{\phi} A' \equiv A\ \mathsf{sig}}{\Gamma \vdash^{\phi} M \equiv M' : A}$$

Figure 4: Structural Rules

TYPE-SIG
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{Type}\ \mathsf{sig}}$$

VAL-SIG
$$\frac{\Gamma \vdash^{\phi} \tau : \mathtt{Type}}{\Gamma \vdash^{\phi} \mathtt{Val}(\tau)\ \mathsf{sig}}$$

COMP-SIG
$$\frac{\Gamma \vdash^{\phi} A\ \mathsf{sig}}{\Gamma \vdash^{\phi} \mathtt{Comp}(A)\ \mathsf{sig}}$$

EXT-SIG
$$\frac{\Gamma \vdash^{\phi} A\ \mathsf{sig} \qquad \Gamma \vdash^{\mathtt{st}} M : A}{\Gamma \vdash^{\phi} \mathtt{Ext}(A; M)\ \mathsf{sig}}$$

UNIT-SIG
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{Unit}\ \mathsf{sig}}$$

STR-SIG
$$\frac{\Gamma \vdash^{\phi} A_1\ \mathsf{sig} \qquad \Gamma, x : A_1 \vdash^{\phi} A_2\ \mathsf{sig}}{\Gamma \vdash^{\phi} \mathtt{Str}(A_1; x.A_2)\ \mathsf{sig}}$$

FUN-SIG
$$\frac{\Gamma \vdash^{\phi} A_1\ \mathsf{sig} \qquad \Gamma, x : A_1 \vdash^{\phi} A_2\ \mathsf{sig}}{\Gamma \vdash^{\phi} \mathtt{Fun}(A_1; x.A_2)\ \mathsf{sig}}$$

Figure 5: Signatures

NAT-TYP
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{nat} : \mathtt{Type}}$$

UNIT-TYP
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{unit} : \mathtt{Type}}$$

UNIT-TYP-EQ
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{Val}(\mathtt{unit}) \equiv \mathtt{Unit}\ \mathsf{sig}}$$

PROD-TYP
$$\frac{\Gamma \vdash^{\phi} \tau_1 : \mathtt{Type} \qquad \Gamma \vdash^{\phi} \tau_2 : \mathtt{Type}}{\Gamma \vdash^{\phi} \tau_1 \times \tau_2 : \mathtt{Type}}$$

PROD-TYP-EQ
$$\frac{\Gamma \vdash^{\phi} \tau_1 : \mathtt{Type} \qquad \Gamma \vdash^{\phi} \tau_2 : \mathtt{Type}}{\Gamma \vdash^{\phi} \mathtt{Val}(\tau_1 \times \tau_2) \equiv \mathtt{Val}(\tau_1) \times \mathtt{Val}(\tau_2)\ \mathsf{sig}}$$

FUN-TYP
$$\frac{\Gamma \vdash^{\phi} \tau_1 : \mathtt{Type} \qquad \Gamma \vdash^{\phi} \tau_2 : \mathtt{Type}}{\Gamma \vdash^{\phi} \tau_1 \to \tau_2 : \mathtt{Type}}$$

FUN-TYP-EQ
$$\frac{\Gamma \vdash^{\phi} \tau_1 : \mathtt{Type} \qquad \Gamma \vdash^{\phi} \tau_2 : \mathtt{Type}}{\Gamma \vdash^{\phi} \mathtt{Val}(\tau_1 \to \tau_2) \equiv \mathtt{Val}(\tau_1) \to \mathtt{Val}(\tau_2)\ \mathsf{sig}}$$

COMP-TYP
$$\frac{\Gamma \vdash^{\phi} \tau : \mathtt{Type}}{\Gamma \vdash^{\phi} \mathtt{comp}(\tau) : \mathtt{Type}}$$

COMP-TYP-DEF
$$\frac{\Gamma \vdash^{\phi} \tau : \mathtt{Type}}{\Gamma \vdash^{\phi} \mathtt{Val}(\mathtt{comp}(\tau)) \equiv \mathtt{Comp}(\mathtt{Val}(\tau))\ \mathsf{sig}}$$

VAL-AST
$$\frac{\Gamma \vdash^{\phi} \tau : \mathtt{Type}}{\Gamma \vdash^{\mathtt{st}} *\colon \mathtt{Val}(\tau)}$$

VAL-ST
$$\frac{\Gamma \vdash^{\phi} e : \mathtt{Val}(\tau)}{\Gamma \vdash^{\mathtt{st}} e \equiv *\colon \mathtt{Val}(\tau)}$$

Figure 6: Small Types

ZERO-VAL
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{zero} : \mathtt{Val}(\mathtt{nat})}$$

SUCC-VAL
$$\frac{\Gamma \vdash^{\phi} e : \mathtt{Val}(\mathtt{nat})}{\Gamma \vdash^{\phi} \mathtt{succ}(e) : \mathtt{Val}(\mathtt{nat})}$$

REC-VAL
$$\frac{\Gamma \vdash^{\phi} e : \mathtt{Val}(\mathtt{nat}) \qquad \Gamma \vdash^{\phi} \rho : \mathtt{Type} \qquad \Gamma \vdash^{\phi} e_0 : \mathtt{Val}(\rho) \qquad \Gamma, x : \mathtt{Val}(\rho) \vdash^{\phi} e_1 : \mathtt{Val}(\rho)}{\Gamma \vdash^{\phi} \mathtt{rec}(e; e_0; x.e_1) : \mathtt{Val}(\rho)}$$

REC-ZERO
$$\frac{\Gamma \vdash^{\phi} \rho : \mathtt{Type} \qquad \Gamma \vdash^{\phi} e_0 : \mathtt{Val}(\rho) \qquad \Gamma, x : \mathtt{Val}(\rho) \vdash^{\phi} e_1 : \mathtt{Val}(\rho)}{\Gamma \vdash^{\phi} \mathtt{rec}(\mathtt{zero}; e_0; x.e_1) \equiv e_0 : \mathtt{Val}(\rho)}$$

REC-SUCC
$$\frac{\Gamma \vdash^{\phi} e : \mathtt{Val}(\mathtt{nat}) \qquad \Gamma \vdash^{\phi} \rho : \mathtt{Type} \qquad \Gamma \vdash^{\phi} e_0 : \mathtt{Val}(\rho) \qquad \Gamma, x : \mathtt{Val}(\rho) \vdash^{\phi} e_1 : \mathtt{Val}(\rho)}{\Gamma \vdash^{\phi} \mathtt{rec}(\mathtt{succ}(e); e_0; x.e_1) \equiv [\mathtt{rec}(e; e_0; x.e_1)/x]e_1 : \mathtt{Val}(\rho)}$$

Figure 7: Natural Numbers

EXT-EQ
$$\frac{\Gamma \vdash^{\phi} A \equiv A' \,\mathsf{sig} \qquad \Gamma \vdash^{\mathtt{st}} M \equiv M' : A}{\Gamma \vdash^{\phi} \mathtt{Ext}(A; M) \equiv \mathtt{Ext}(A'; M') \,\mathsf{sig}}$$

EXT-IS
$$\frac{\Gamma \vdash^{\mathtt{st}} M : A \qquad \Gamma \vdash^{\phi} N : A \qquad \Gamma \vdash^{\mathtt{st}} N \equiv M : A}{\Gamma \vdash^{\phi} \mathtt{is}(N) : \mathtt{Ext}(A; M)}$$

EXT-AS
$$\frac{\Gamma \vdash^{\phi} N : \mathtt{Ext}(A; M)}{\Gamma \vdash^{\phi} \mathtt{as}(N) : A}$$

EXT-AS-EQ
$$\frac{\Gamma \vdash^{\phi} N : \mathtt{Ext}(A; M)}{\Gamma \vdash^{\mathtt{st}} \mathtt{as}(N) \equiv M : A}$$

EXT-BETA
$$\frac{\Gamma \vdash^{\mathtt{st}} M : A \qquad \Gamma \vdash^{\phi} N : A \qquad \Gamma \vdash^{\mathtt{st}} N \equiv M : A}{\Gamma \vdash^{\phi} \mathtt{as}(\mathtt{is}(N)) \equiv N : A}$$

EXT-ETA
$$\frac{\Gamma \vdash^{\phi} N : \mathtt{Ext}(A; M)}{\Gamma \vdash^{\phi} \mathtt{is}(\mathtt{as}(N)) \equiv N : \mathtt{Ext}(A; M)}$$

Figure 8: Extensions

$$\textsc{comp}\ \frac{\Gamma \vdash^{\phi} E \div A}{\Gamma \vdash^{\phi} \mathtt{comp}(E) : \mathtt{Comp}(A)}$$

$$\textsc{ret}\ \frac{\Gamma \vdash^{\phi} M : A}{\Gamma \vdash^{\phi} \mathtt{ret}(M) \div A} \qquad \textsc{bnd}\ \frac{\Gamma \vdash^{\phi} M_1 : \mathtt{Comp}(A_1) \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 : A_2}{\Gamma \vdash^{\phi} \mathtt{bnd}(M_1 ; x.E_2) \div A_2}$$

$$\textsc{ret-bnd}\ \frac{\Gamma \vdash^{\phi} M_1 : A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 \div A_2}{\Gamma \vdash^{\phi} \mathtt{bnd}(\mathtt{comp}(\mathtt{ret}(M_1)) ; x.E_2) \equiv [M_1/x]E_2 \div A_2} \qquad \textsc{bnd-ret}\ \frac{\Gamma \vdash^{\phi} M : \mathtt{Comp}(A)}{\Gamma \vdash^{\phi} M \equiv \mathtt{comp}(\mathtt{bnd}(M ; x.\, \mathtt{ret}(x))) : \mathtt{Comp}(A)}$$

$$\textsc{bnd-assoc}\ \frac{\Gamma \vdash^{\phi} M_1 : \mathtt{Comp}(A_1) \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 \div A_2 \qquad \Gamma, x_2 : A_2 \vdash^{\phi} E_3 \div A_3}{\Gamma \vdash^{\phi} \mathtt{bnd}(\mathtt{comp}(\mathtt{bnd}(M_1 ; x_1.E_2)) ; x_2.E_3) \equiv \mathtt{bnd}(M_1 ; x_1.\, \mathtt{bnd}(\mathtt{comp}(E_2) ; x_2.E_3)) \div A_3}$$

$$\textsc{comp-star}\ \frac{\Gamma \vdash^{\phi} A\,\mathsf{sig}}{\Gamma \vdash^{\mathtt{st}} \star \div A} \qquad \textsc{comp-st}\ \frac{\Gamma \vdash^{\phi} E \div A}{\Gamma \vdash^{\mathtt{st}} E \equiv \star \div A}$$

Figure 9: Computations

UNIT-IN
$$\frac{\vdash^{\phi} \Gamma\ \mathsf{ctx}}{\Gamma \vdash^{\phi} \mathtt{triv} : \mathtt{Unit}}$$

UNIT-ETA
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Unit}}{\Gamma \vdash^{\phi} M \equiv \mathtt{triv} : \mathtt{Unit}}$$

STR-IN
$$\frac{\Gamma \vdash^{\phi} M_1 : A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} A_2\ \mathsf{sig} \qquad \Gamma \vdash^{\phi} M_2 : [M_1/x]A_2}{\Gamma \vdash^{\phi} \mathtt{str}(M_1 ; M_2) : \mathtt{Str}(A_1 ; x.A_2)}$$

STR-EL-1
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Str}(A_1 ; x.A_2)}{\Gamma \vdash^{\phi} M \cdot 1 : A_1}$$

STR-EL-2
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Str}(A_1 ; x.A_2)}{\Gamma \vdash^{\phi} M \cdot 2 : [M \cdot 1/x]A_2}$$

STR-BETA-1
$$\frac{\Gamma \vdash^{\phi} M_1 : A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} A_2\ \mathsf{sig} \qquad \Gamma \vdash^{\phi} M_2 : [M_1/x]A_2}{\Gamma \vdash^{\phi} \mathtt{str}(M_1 ; M_2) \cdot 1 \equiv M_1 : A_1}$$

STR-BETA-2
$$\frac{\Gamma \vdash^{\phi} M_1 : A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} A_2\ \mathsf{sig} \qquad \Gamma \vdash^{\phi} M_2 : [M_1/x]A_2}{\Gamma \vdash^{\phi} \mathtt{str}(M_1 ; M_2) \cdot 2 \equiv M_2 : [M_1/x]A_2}$$

STR-ETA
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Str}(A_1 ; x.A_2)}{\Gamma \vdash^{\phi} M \equiv \mathtt{str}(M \cdot 1 ; M \cdot 2) : \mathtt{Str}(A_1 ; x.A_2)}$$

Figure 10: Structures

FUN-IN
$$\frac{\Gamma \vdash^{\phi} A_1\ \mathsf{sig} \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 \mathrel{\div} A_2}{\Gamma \vdash^{\phi} \mathtt{fun}(x.E_2) : \mathtt{Fun}(A_1 ; x.A_2)}$$

FUN-EL
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Fun}(A_1 ; x.A_2) \qquad \Gamma \vdash^{\phi} M_1 : A_1}{\Gamma \vdash^{\phi} \mathtt{app}(M ; M_1) \mathrel{\div} [M_1/x]A_2}$$

FUN-BETA
$$\frac{\Gamma, x : A_1 \vdash^{\phi} E_2 : A_2 \qquad \Gamma \vdash^{\phi} M_1 : A_1}{\Gamma \vdash^{\phi} \mathtt{app}(\mathtt{fun}(x.E_2) ; M_1) \equiv [M_1/x]E_2 \mathrel{\div} [M_1/x]A_2}$$

FUN-ETA
$$\frac{\Gamma \vdash^{\phi} M : \mathtt{Fun}(A_1 ; x.A_2)}{\Gamma \vdash^{\phi} M \equiv \mathtt{fun}(x.\, \mathtt{app}(M ; x)) : \mathtt{Fun}(A_1 ; x.A_2)}$$

Figure 11: Generative Functors