Logical Relations as Types
Proof-Relevant Parametricity for Program Modules

Jonathan Sterling Robert Harper

October 16, 2020

Abstract

The theory of program modules is of interest to language designers not only for its practical importance to programming, but also because it lies at the nexus of three fundamental concerns in language design: *the phase distinction*, *computational effects*, and *type abstraction*. We contribute a fresh "synthetic" take on program modules that treats modules as the fundamental constructs, in which the usual suspects of prior module calculi (kinds, constructors, dynamic programs) are rendered as derived notions in terms of a modal type-theoretic account of the phase distinction. We simplify the account of type abstraction (embodied in the generativity of module functors) through a *lax modality* that encapsulates computational effects, placing *projectibility* of module expressions on a type-theoretic basis.

Our main result is a (significant) proof-relevant and phase-sensitive generalization of the Reynolds abstraction theorem for a calculus of program modules, based on a new kind of logical relation called a *parametricity structure*. Parametricity structures generalize the proof-irrelevant relations of classical parametricity to proof-*relevant* families, where there may be non-trivial evidence witnessing the relatedness of two programs — simplifying the metatheory of strong sums over the collection of types, for although there can be no "relation classifying relations", one easily accommodates a "family classifying small families".

Using the insight that logical relations/parametricity is *itself* a form of phase distinction between the syntactic and the semantic, we contribute a new synthetic approach to phase separated parametricity based on the slogan *logical relations as types*, by iterating our modal account of the phase distinction. We axiomatize a dependent type theory of parametricity structures using two pairs of complementary modalities (syntactic, semantic) and (static, dynamic), substantiated using the topos theoretic *Artin gluing* construction. Then, to construct a simulation between two implementations of an abstract type, one simply programs a third implementation whose type component carries the representation invariant.

1 Introduction

Program modules are the application of dependent type theory with universes to the large-scale structuring of programs. As MacQueen [Mac86] observed, the hierarchical structuring of programs is an instance of dependent sum; consider the example of a type together with a pretty printer:

```
(* SHOW := ∑T:𝒰(T ⇀ string) *)
signature SHOW = sig
  type t
  val show : t ⇀ string
end
```

On the other hand, the parameterization of a program component in another component is an instance of dependent product; for instance, consider a module functor that implements a pretty printer for a product type:

```
(* ShowProd : ∏S₁,S₂:SHOW(π₁(S₁) * π₁(S₂) ⇀ string) *)
functor ShowProd (S1 : SHOW) (S2 : SHOW) : sig
  type t = S1.t * S2.t
  val show : t ⇀ string
end = ...
```

Modules are more than just dependent products, sums, and universes, however: a module language must account for abstraction and the phase distinction, two critical notions that seem to complicate the simple story of modules as dependent types. In Section 1.1, we introduce ModTT, our take on a type theory for program modules, and explain how to view abstraction and generativity in terms of a *lax modality* or strong monad; in Section 1.2, the phase distinction is seen to arise naturally from an *open modality* in the sense of topos theory.

1.1 Abstraction and computational effects

Reynolds famously argued that *"Type structure is a syntactic discipline for enforcing levels of abstraction"* [Rey83]; abstraction is the facility to manage the non-equivalence of types at the boundary between spuriously compatible program fragments — for instance, the boundary between a fragment of a compiler that emits a De Bruijn index (address of a variable counted from the right) and a fragment that accepts a De Bruijn level (the address counted from the left).

1.1.1 Static abstraction via let binding

The primary aspect of abstraction is, then, to prevent the "false linkage" of programs permitted by coincidence of representation; the static distinction between two different uses of the same type can be achieved by the standard rule for (non-dependent) let-binding in type theory:

NON-DEPENDENT LET
$$\frac{\Gamma \vdash A\ \mathit{type} \qquad \Gamma \vdash B\ \mathit{type} \qquad \Gamma \vdash N : A \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\mathsf{let}\ x : A = N\ \mathsf{in}\ M) : B}$$

Static "let abstraction" as above enables the programmer to treat the same type differently in two locations, but share the same values at runtime. For instance, consider the following expression that binds the *integer equality* structure twice, for two different purposes:

$$\begin{array}{l} \mathsf{let}\ \mathsf{DeBruijnLevel} : \mathsf{EQ} = (\mathsf{int}, \mathsf{int\_eq})\ \mathsf{in} \\ \mathsf{let}\ \mathsf{DeBruijnIndex} : \mathsf{EQ} = (\mathsf{int}, \mathsf{int\_eq})\ \mathsf{in} \\ M \end{array}$$

```
functor Namespace (A : ARRAY) :> NAMESPACE = struct
  type symbol = int
  val table = A.new (* allocation size *)
  val defined str = (* see if [str] has already been allocated *)
  val into str = (* hash [str] and insert it into [table] if needed *)
  fun out sym =
    case A.sub (table, sym) of
    | NONE ⇒ raise Impossible
    | SOME str ⇒ str
end
```

Figure 1: A functor that generates a new namespace in Standard ML.

In the scope of M it is not the case that DeBruijnLevel and DeBruijnIndex have the same type component. But at runtime, M will be instantiated with the same type and value components in both positions. In the Standard ML implementation of modules, a more sophisticated form of let binding is elaborated that actually exposes the static identity of the bound term in the body; for this reason, Standard ML programmers use *dynamic* abstraction (Section 1.1.2) via the opaque ascription `M :> S` to negotiate both static and dynamic abstraction situations.

1.1.2 Dynamic abstraction via modal binding

In the presence of computational effects and module functors, it is not always enough to statically distinguish between two "instances" of the same type: the body of a module functor may contain a local state that must be distinctly initiated in every instantiation. Sometimes referred to as *generativity*, the need for this dynamic form of abstraction can be illustrated by means of an ephemeral structure to manage a given namespace in a compiler:

```
signature NAMESPACE = sig
  type symbol
  val defined : string ⇀ bool
  val into : string ⇀ symbol
  val out : symbol ⇀ string
  val eq : symbol * symbol ⇀ bool
end
```

To manage two different namespaces, one requires two distinct copies `NS1, NS2` of the `Namespace` structure. If it were not for the `defined` operator, it would be safe to generate a single `Namespace` structure and bind it to two different module variables: we would have `NS1.symbol` ≠ `NS2.symbol` but at runtime, the same table would be used. However, this behavior becomes observably incorrect in the presence of `defined`, which exposes the internal state of the namespace.

The dynamic effect of initializing the namespace structure once per instantiation has historically been treated in terms of a notion of *projectibility* [DCH03; Har16], restricting when the components of a module expression can be projected; under the generative semantics of module functors, a functor application is never projectible. Projectibility, however, is not a type-theoretic concept because it does not respect substitution!

We argue that it is substantially simpler to present the module calculus with an explicit separation of effects via a lax modality / strong monad $\bigcirc$; concurrent work of Crary supports the same conclusion [Cra20]. ModTT distinguishes between computations $\mathrm{M} \div \sigma$ and values $\mathrm{V} : \sigma$, and mediates between them using the standard rules of the lax modality [FM97]:

$$\frac{\Gamma \vdash \sigma\ \mathit{sig}}{\Gamma \vdash \bigcirc\sigma\ \mathit{sig}} \qquad \frac{\Gamma \vdash \mathrm{V} : \sigma}{\Gamma \vdash \mathsf{ret}(\mathrm{V}) \div \sigma} \qquad \frac{\Gamma \vdash \mathrm{V} : \bigcirc\sigma \qquad \Gamma, \mathrm{X} : \sigma \vdash \mathrm{M} \div \sigma'}{\Gamma \vdash (\mathrm{X} \leftarrow \mathrm{V}; \mathrm{M}) \div \sigma'} \qquad \frac{\Gamma \vdash \mathrm{M} \div \sigma}{\Gamma \vdash \{\mathrm{M}\} : \bigcirc\sigma}$$

In this style, one no longer needs the notion of projectibility: a generative functor is nothing more than a module-level function $\sigma \Rightarrow \bigcirc\tau$, and the result of applying such a function must be bound in the monad before it can be used, so one naturally obtains the generative semantics without resorting to an ad hoc notion of "generative" or "applicative" function space.

```
NS1 ⟵ Namespace (Array);
NS2 ⟵ Namespace (Array); ...
```

1.2 The phase distinction

The division of labor between the lightweight syntactic verification provided by type abstraction and the more thoroughgoing but expensive verification provided by program logics is substantiated by the phase distinction between the static/compiletime and dynamic/runtime parts of a program respectively. Respect for the phase distinction means that there is a well-defined notion of static equivalence of program fragments that is independent of dynamic equivalence; moreover, one must ensure that static equivalence is efficiently decidable for it to be useful in practice.

1.2.1 Explicit phase distinction

The phase distinction calculi of Moggi [Mog89] and Harper, Mitchell, and Moggi [HMM90] capture the separation of static from dynamic in an explicit and intrinsic way: a core calculus of modules is presented with an explicit distinction between (modules, signatures) and (constructors, kinds) in which the latter play the role of the static part of the former. A signature is explicitly split into a (static) kind $k : \mathit{kind}$ and a (dynamic) type $u : k \vdash t(u) : \mathit{type}$ that depends on it, and module value is a pair (c, e) where $c : k$ and $e : t(c)$. Functions of modules are defined by a "twinned" lambda abstraction $\lambda u/x.\mathrm{M}$, and scoping rules are used to ensure that static parts depend only on constructor variables $u : k$ and not on term variables $x : t$.

An unfortunate consequence of the explicit presentation of phase separation is that the rules for type-theoretic connectives (dependent product, dependent sum) become wholely non-standard and it is not immediately clear in which sense these actually *are* dependent product or sum. For instance, one has rules like the following for dependent product:

PI FORMATION*

$$\frac{\Delta \vdash k\ \mathit{kind} \qquad \Delta, u : k \vdash k'(u)\ \mathit{kind} \\ \Delta, u : k; \Gamma \vdash \sigma(u)\ \mathit{type} \qquad \Delta, u : k; \Gamma, u' : k'(u); \Gamma \vdash \sigma'(u, u')\ \mathit{type}}{\Delta; \Gamma \vdash \Pi u/\mathrm{X} : [u : k.\sigma(u)].[u' : k'(u).\sigma'(u, u')] \equiv [k : (\Pi u : k.k'(u)); \Pi u : k.\sigma(u) \to \sigma'(u, v(u))]\ \mathit{sig}}$$

The Grothendieck construction Moggi observed that the explicit phase distinction calculus can be understood as arising from an indexed category in the following sense:

1) One begins with a purely static language, i.e. a category $\mathscr{B}$ whose objects are kinds and whose morphisms are constructors.
2) Next one defines an indexed category $\mathscr{C} : \mathscr{B}^{\mathrm{op}} \longrightarrow \mathbf{Cat}$: for a kind k, the fiber category $\mathscr{C}(k)$ is the collection of signatures with static part k, with morphisms given by functions of module expressions.

Then, the syntactic category of the full calculus is obtained by the *Grothendieck construction* $\mathscr{G} = \int_{\mathscr{B}} \mathscr{C}$, which takes an indexed category to its total category. An object of $\mathscr{G}$ is a pair (k, σ) with $k : \mathscr{B}$ and $\sigma : \mathscr{C}(k)$; a morphism $(k, \sigma) \longrightarrow (k', \sigma')$ is a morphism $c : k \longrightarrow k' : \mathscr{B}$ together with a morphism $\sigma \longrightarrow c^*\sigma' : \mathscr{C}(k)$, where, as usual, c^* is $\mathscr{C}c$.

The benefit of considering $\mathscr{G}$ is that the non-standard rules for type theoretic connectives become a special case of the standard ones: from this perspective, the strange PI FORMATION* rule (with its nonstandard contexts and scoping and variable twinning) above can be seen to be a certain calculation in the Grothendieck construction of a certain dependent product.

1.2.2 Implicit phase distinction

An alternative to the explicit phase separation of Harper, Mitchell, and Moggi [HMM90] is to treat the module calculus as ordinary type theory, extended by a judgment for *static equivalence*. Then, two modules are considered statically equivalent when they have the same static part — though the projection of static parts is defined metatheoretically rather than intrinsically. This approach is represented by Dreyer, Crary, and Harper [DCH03].

1.2.3 This paper: synthetic phase distinction

Taking inspiration from both the explicit and implicit accounts of phase separation, we note that the detour through indexed categories was strictly unnecessary, and the object of real interest is the category $\mathscr{G}$ and the corresponding fibration $\mathscr{G} \longrightarrow \mathscr{B}$ that projects the static language from the full language. We obtain further leverage by reconstructing $\mathscr{B}$ as a slice $\mathscr{G}_{/\faUnlock_{\mathrm{st}}}$ for a special object $\faUnlock_{\mathrm{st}} : \mathscr{G}$.

The view of $\mathscr{B}$ as a slice of $\mathscr{G}$ is inspired by Artin gluing [AGV72], a mathematical version of logical predicates in which the syntactic category of a theory is reconstructed as a slice of a topos of logical predicates: there is a very precise sense in which the notion of "signature over a kind" can be identified with "logical predicate on a kind". The connection between phase separation and gluing/logical predicates is, to our knowledge, a novel contribution of this paper.

Put syntactically, the language corresponding to $\mathscr{G}$ possesses a new context-former $(\Gamma, \faUnlock_{\mathrm{st}})$ called the "static open";[1] when $\faUnlock_{\mathrm{st}}$ is in the context, everything except the static part of an object is ignored by the judgmental equality relation $A \equiv B$. For instance, module computations and terms of program type are rendered purely dynamic / statically inert by means of

[1] The terminology of "opens" is inspired by topos theory, in which proof irrelevant propositions correspond to partitions into open and closed subtopoi. Indeed, such a partition is the geometrical prototype of the phase distinction.

special rules of static connectivity under the assumption of 🔓_{st}:

$$\textsc{static open}\quad \frac{\Gamma\ \mathit{ctx}}{\Gamma, \text{🔓}_{\mathrm{st}}\ \mathit{ctx}} \qquad \textsc{static connectivity (1)}\quad \frac{\Gamma \vdash t : \mathsf{type} \qquad \Gamma \vdash \text{🔓}_{\mathrm{st}}}{\Gamma \vdash * : t} \qquad \textsc{static connectivity (2)}\quad \frac{\Gamma \vdash t : \mathsf{type} \qquad \Gamma \vdash e : t \qquad \Gamma \vdash \text{🔓}_{\mathrm{st}}}{\Gamma \vdash e \equiv * : t}$$

$$\textsc{static connectivity (3)}\quad \frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma \vdash \text{🔓}_{\mathrm{st}}}{\Gamma \vdash * \div \sigma} \qquad \textsc{static connectivity (4)}\quad \frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma \vdash \mathrm{M} \div \sigma \qquad \Gamma \vdash \text{🔓}_{\mathrm{st}}}{\Gamma \vdash \mathrm{M} \equiv * \div \sigma}$$

Signatures, kinds, and static equivalence In our account, the phase distinction between signatures/modules and kinds/constructors is expressed by a universal property: a signature $\Gamma \vdash \sigma\ \mathit{sig}$ is called a kind iff the weakening of sets of equivalence classes from $\{[\mathrm{V}] \mid \Gamma \vdash \mathrm{V} : \sigma\}$ to $\{[\mathrm{V}] \mid \Gamma, \text{🔓}_{\mathrm{st}} \vdash \mathrm{V} : \sigma\}$ is an isomorphism natural in Γ. In other words, the exponentiation by 🔓_{st} defines an *open modality* $\mathrm{St} = (\text{🔓}_{\mathrm{st}} \to -)$ in the sense of topos theory, and a kind is nothing more than an St-modal signature.

Because the modality St is idempotent, we may define (internally!) the static part of any signature σ as $\mathrm{St}(\sigma)$; the modal unit $\eta_{\mathrm{St}} : \sigma \longrightarrow \mathrm{St}(\sigma)$ abstractly implements the projection of constructors from module values. Because the modality St is defined by exponentiation with a subterminal (i.e. a proof-irrelevant sort), it is easy to show internally that the usual equations of static projection hold (naturally, up to isomorphism): for instance, we have $\mathrm{St}(\sigma \Rightarrow \tau) \cong \mathrm{St}(\sigma) \Rightarrow \mathrm{St}(\tau)$, etc.

The notion of static equivalence from Dreyer, Crary, and Harper [DCH03] is then reconstructed as ordinary judgmental equality in the context of 🔓_{st}; the view of phase separation as a projection functor from Moggi [Mog89] is reconstructed by the weakening $\mathscr{G} \longrightarrow \mathscr{G}_{/\text{🔓}_{\mathrm{st}}}$.

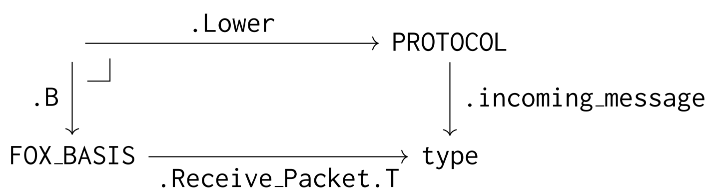
1.3 Sharing constraints, singletons, and the *static extent* connective

An important practical aspect of module languages is the ability to constrain the identity of a substructure; for instance, the implementation of IP in the FoxNet protocol stack [Bia+94] is given as a functor taking two structures as arguments *under the additional constraint* that the structures have compatible type components:

```
functor Ip
  (structure Lower : PROTOCOL
   structure B : FOX_BASIS
     where type Receive_Packet.T = Lower.incoming_message
   ...)
```

1.3.1 Sharing as pullback

The above fragment of the input to the `Ip` functor can be viewed as a pullback of two signatures along type projections, rather than a product of two signatures:



The view of sharing in terms of pullback or equalizers, proposed by Mitchell and Harper [MH88], is perfectly appropriate from a semantic perspective; however, it unfortunately renders type checking undecidable [CCD17]. Because types in ML-style languages are meant to provide *lightweight* verification, it is essential that the type checking problem be tractable: therefore, something weaker than general pullbacks is required. Semantically speaking, what one needs is roughly pullback along display maps only, i.e. equations that can be oriented as definitions.

1.3.2 Type sharing via singletons

A strategy more well-adapted to implementation is to elaborate type sharing in a way that involves a new singleton type signature $\mathcal{S}(t)$ *sig* for each $t : \texttt{type}$, as pioneered by Harper and Stone [HS00]. There is up to judgmental equality exactly one module of signature $\mathcal{S}(t)$, namely t itself; in contrast to general pullbacks, the singleton signature does not disrupt the decidability of type equivalence [SH06; ACP09].

The truly difficult part of singleton types, dealt with by Stone and Harper [SH06], is their subtyping and re-typing principles: not only should it be possible to pass from a more specific type to a less specific type, it must also be possible to pass from a less specific type to a more specific type when the identity of the value is known. Because of the dependency involved in the latter transition, ordinary subtyping is not enough to account for the full expressivity of singletons, hence the extensional retyping principles of earlier work on singleton calculi [DCH03; Cra19].

As a basic principle, we do not treat subtyping or retyping directly in the core type theory: we intend to give an *algebraic* account of program modules, so both subtyping and retyping become a matter of elaborating coercions. We propose to account for both the subtyping and retyping principles via an elaboration algorithm guided by the η-laws of each connective, including the η-laws of the singleton type connective. Early evidence that our proposal is tractable can be found in the implementation of the `cooltt` proof assistant for cubical type theory, which treats a generalization of singleton types via such an algorithm [Red20].[2]

1.3.3 General sharing via the *static extent*

It is useful to express the compatibility of components of modules other than types: families of types (e.g. the polymorphic type of lists) are one example, but arguably one should be able to express a sharing constraint on an entire substructure. Type theoretically, it is trivial to generalize the type singletons in this direction, but we risk incurring static dependencies on dynamic components of signatures, violating the spirit of the phase distinction.

One of the design constraints for module systems, embodied in the phase distinction, is that dependency should only involve static constructs; the decidable fragment of the dynamic algebra of programs is unfortunately too fine to act as more than an obstruction to the composition of program components. From our synthetic view of the phase distinction, it is most natural to rather generalize the type singletons to a signature connective $\{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}$ that classifies the "static extent" of a module $V : \sigma$ for an arbitrary signature σ, summarized

[2] An example of the application of `cooltt`'s elaboration algorithm to the subtyping and retyping of singletons can be found here: `https://github.com/RedPRL/cooltt/blob/7be1bb32f8b0eaae75c5a11f1c1c5b0ff1086c94/test/selfification.cooltt`.

in the following rules of inference:[3]

$$\textsc{formation}\quad \frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash \mathrm{V} : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow \mathrm{V}\}\ \mathit{sig}} \qquad \textsc{introduction}\quad \frac{\Gamma \vdash \mathrm{U} : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash \mathrm{U} \equiv \mathrm{V} : \sigma}{\Gamma \vdash \mathrm{U} : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow \mathrm{V}\}}$$

$$\textsc{elimination}\quad \frac{\Gamma \vdash \mathrm{U} : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow \mathrm{V}\}}{\Gamma \vdash \mathrm{U} : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash \mathrm{U} \equiv \mathrm{V} : \sigma}$$

In ModTT, the elements of the *static extent* of a module $\mathrm{V} : \sigma$ are all the modules whose static part is judgmentally equal to V; therefore $\{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow \mathrm{V}\}$ is not a singleton in general, but it is a singleton when σ is purely static. Our approach is equivalent to (but arguably more convenient than) the use of singleton kinds: the static extent is admissible under the explicit phase distinction.

Extension types in cubical type theory Our static extent connective is inspired by the *extension types* of Riehl and Shulman [RS17], already available in a few implementations of cubical type theory [Red18; Red20]. Whereas in cubical type theory one extends along a cofibrant subobject $\phi \rightarrowtail \mathbb{I}^n$ of a cube, in a phase separated module calculus one extends along the open domain $\text{🔓}_{\mathsf{st}} \rightarrowtail \mathbf{1}$. The static extent connective is also closely related to the formal disk bundle of Wellen [Wel17], which classifies the "infinitesimal extent" of a given point in synthetic differential (higher) geometry.

Strong structure sharing à la SML '90 Another account of the sharing of structures is argued for in earlier versions of Standard ML [MTH90], in which each structure is in essence tagged with a static identity [MHR20]; this "strong" structure sharing was replaced in SML '97 by the current "weak" structure sharing, which has force only on the static components of the signature [Mil+97]. Our static extents capture exactly the semantics of weak structure sharing; we note that the strong sharing of SML '90 can be simulated by adding a dummy abstract type to each signature during elaboration.

1.4 Proof-relevant parametricity: the objective metatheory of ML modules

We outline an approach to the definition and metatheory of a calculus for program modules, together with a modernized take on logical relations / Tait computability that enables succinct proofs of representation independence and parametricity results.

1.4.1 Algebraic metatheory in an equational logical framework

Many existing calculi for program modules are formulated using raw terms, and animated via a mixture of judgmental equality (for the module layer) and structural operational semantics (for the program layer). In contrast, we formulate ModTT entirely in an equational logical framework,[4] eschewing raw terms entirely and *only* considering terms up to typed judgmental equality. Because we have adopted a modal separation of effects (Section 1.1), there is no obstacle to accounting for genuine computational effects in the program layer, even in the purely equational setting [Sta13].

The mechanization of Standard ML [LCH07; CH09] in the Edinburgh Logical Framework [HHP93] is an obvious precursor to our design; whereas in the cited work, the LF's

[3]For simplicity, we present these rules in a style that violates uniqueness of types; the actual encoding in the logical framework is achieved using explicit introduction and elimination forms.

[4]Though we present it using standard notations for readability.

function space was used to encode the binding structure of raw terms and derivations, we employ a semantic logical framework due to Uemura [Uem19] to account for both typing and judgmental equality of abstract terms. The idea of dependently typed equational logical frameworks goes back to Cartmell [Car78] (for theories without binding), and was further developed by Martin-Löf for theories with binding of arbitrary order [NPS90]. Because we work only with typed terms up to judgmental equality, we may use *semantic* methods such as Artin gluing to succinctly prove syntactic results as in several recent works [AK16; Coq19; KHS19; CHS19; SAG19; SAG20; SA20].

The effectiveness of algebraic methods relies on the existence of initial algebras for theories defined in a logical framework. The existence of initial algebras is not hard to prove and usually follows from standard results in category theory. That an initial algebra can be *presented* by a quotient of raw syntax is more laborious to prove for a given logical framework (see Streicher [Str91] for a valiant effort); such a result is the combination of soundness and completeness.

It comes as a pleasant surprise, then, that the syntactic presentation of the core language is not in practice germane to the study of real type theories and programming languages: the only raw syntax one need be concerned with is that of the surface language, but the surface language is almost never expected to be complete for the core language, or even to have meaning independently of its elaboration into the core language. The fulfillment of any such expectation is immediately obstructed by the myriad non-compositional aspects of the elaboration of surface languages, including not only the use of unification to resolve implicit arguments and coercions, but also even the complex name resolution scopes induced by ML's open construct.

1.4.2 Artin gluing and logical relations

Logical relations, or Tait computability [Tai67], is a method by which a relation on terms of base type is equipped with a canonical *hereditary* action on type constructors. The hereditary action can be seen as a generalization of the induction hypothesis that allows a non-trivial property of base types to be proved, a perspective summarized in Harper's tutorial note [Har19]. For instance, let $\mathrm{R}_{\mathsf{bool}} \subseteq \mathsf{ClosedTerms}(\mathsf{bool})$ be the property of being either #t or #f; one shows that $\mathrm{R}_{\mathsf{bool}}$ holds of every closed boolean by lifting it to each connective in a compositional way:

$$f \in \mathrm{R}_{\sigma\to\tau} \Leftrightarrow \forall x \in \mathrm{R}_{\sigma}. f(x) \in \mathrm{R}_{\tau}$$

Other properties (like parametricity) lift to the other connectives in a similar way. The main obstruction to replacing this method by a general theorem is the fact that programming languages are traditionally defined in terms of hand-coded raw terms and operational semantics; for languages defined in this way, there is *a priori* no way to factor out the common aspects of logical relations.

In an algebraic setting, however, the syntax of a programming language is embodied in a particular category equipped with various structures characterized by universal properties (as detailed in Section 1.4.1). Here, it is possible to replace the *method* of logical relations with a *general theory* of logical relations, namely the theory of Artin gluing. First developed in the 1970s by the Grothendieck school for the purposes of algebraic geometry [AGV72], Artin gluing can be viewed as a tool to "stitch together" a type theory's syntactic category with a category of semantic things, leading to a category of "families of semantic things indexed in syntactic things". Logical relations are then the proof-irrelevant special case of gluing, where families are restricted to have subsingleton fibers.

Example 1.1 (Canonicity by global sections)**.** For instance, let $\mathscr{C}$ be the category of contexts and substitutions for a given language; the global sections functor $[1,-] : \mathscr{C} \longrightarrow \mathbf{Set}$ takes each context $\Gamma : \mathscr{C}$ to the set $[1,\Gamma]$ of closed substitutions for Γ. Then, the gluing of $\mathscr{C}$ along $[1,-]$ is the category $\mathscr{G}$ of pairs $(\Gamma, \tilde{\Gamma})$ where $\tilde{\Gamma}$ is a family of sets indexed in closing substitutions for Γ; given a closing substitution $\gamma \in [1,\Gamma]$, an element of the fiber $\tilde{\Gamma}_\gamma$ should be thought of as evidence that γ is "computable". An object of $\mathscr{G}$ is called a *computability structure* or a *logical family.*

The fundamental lemma of logical relations is located in the proof that $\mathscr{G}$ admits the structure of a model of the given type theory, and that the projection functor $\mathscr{G} \longrightarrow \mathscr{C}$ is a homomorphism of models. In particular, one may choose to define the $\mathscr{G}$-structure of the booleans to be the following, letting $q : 2 \to [1, \mathsf{bool}]$ be the function determined by the pair of closed terms (#t, #f):

$$\left(\mathsf{bool}, \{i : 2 \mid q(i) = b\}_{(b \in [1,\mathsf{bool}])}\right)$$

Then, by the fundamental lemma, every closed boolean is either #t or #f.

Example 1.2 (Binary logical relations on closed terms)**.** Rather than gluing along the global sections functor $[1,-]$, one may glue along $[1,-] \times [1,-]$: then a computability structure over context Γ is a family of sets $\tilde{\Gamma}$ indexed in pairs of closing substitutions for Γ. An ordinary binary logical relation is, then, a computability structure $\tilde{\Gamma}$ such that each fiber $\tilde{\Gamma}_{\gamma,\gamma'}$ is subsingleton.

Because traditional logical relations are defined on raw terms rather than judgmental equivalence classes thereof, their substantiation requires a great deal of syntactical bureaucracy and technical lemmas. By working abstractly over judgmental equivalence classes of typed terms, Artin gluing sweeps away these inessential details completely, but this is only possible by virtue of the fact that Artin gluing treats *families* (proof-relevant relations) in general, rather than only proof-irrelevant relations: the computability of a given term is a structure with evidence, rather than just a property of the term.

The proof relevance is important for many applications: for instance, a redex and its contractum lie in the same judgmental equivalence class, so it would seem at first that there is no way to treat normalization in a super-equational way. The insight of Fiore [Fio02] and Altenkirch, Hofmann, and Streicher [AHS95] from the 1990s is that normal forms can be presented as a *structure* over equivalence classes of typed terms, rather than as a *property* of raw terms. In many cases, the structures end up being fiberwise subsingleton, but this usually cannot be seen until after the fundamental lemma is proved.

An even more striking use of proof relevance, explained by Shulman [Shu13; Shu15] and Coquand [Coq19], is the computability interpretation of universes. A universe is a special type $\mathscr{U}$ whose elements $A : \mathscr{U}$ may be regarded as types $\mathsf{El}(A)$ *type*; in order to substantiate the part of the fundamental lemma that expresses closure under $\mathsf{El}(-)$, we must have a way to extract a logical relation over $\mathsf{El}(A)$ from each computable element $A : \mathscr{U}$. This would seem to require a "relation of relations", but there can be no such thing: the fibers of relations are subsingleton.

In the past, type theorists have accounted for the logical relations of universes by parameterizing the construction in the graph of an assignment of logical relations to type codes [All87], or by using induction-recursion; either approach, however, forces the universe to be closed and inductively defined — disrupting certain applications of logical relations, including parametricity. The proof relevance accorded by Artin gluing offers a more direct solution to the problem: one can always have a "family of small families".

Propositions:	$\unlock_{\mathsf{syn/l}}, \unlock_{\mathsf{syn/r}}, \unlock_{\mathsf{syn}}, \unlock_{\mathsf{st}} : \Omega$	$\unlock_{\mathsf{syn}} = \unlock_{\mathsf{syn/l}} \vee \unlock_{\mathsf{syn/r}}$	$\unlock_{\mathsf{syn/l}} \wedge \unlock_{\mathsf{syn/r}} = \bot$
Open modalities:	$\mathsf{Syn}, \mathsf{St} : \mathscr{U} \to \mathscr{U}$	$\mathsf{Syn}(A) = \unlock_{\mathsf{syn}} \to A$	$\mathsf{St}(A) = \unlock_{\mathsf{st}} \to A$
Closed modalities:	$\mathsf{Sem}, \mathsf{Dyn} : \mathscr{U} \to \mathscr{U}$	$\mathsf{Sem}(A) = A \sqcup_{A \times \unlock_{\mathsf{syn}}} \unlock_{\mathsf{syn}}$	$\mathsf{Dyn}(A) = A \sqcup_{A \times \unlock_{\mathsf{st}}} \unlock_{\mathsf{st}}$

Figure 2: A summary of the structure available in the internal language of a topos of *synthetic phase separated parametricity structures.* Above, $A \sqcup_{A\times\varphi} \varphi$ is the pushout along the product projections of $A \times \varphi$. In Section 3.1.2 and Lemma 3.6, we prove the closed modalities are complementary to the open modalities in the sense that $\mathsf{Syn}(\mathsf{Sem}(A)) \cong \mathbf{1}$ and $\mathsf{St}(\mathsf{Dyn}(A)) = \mathbf{1}$.

1.4.3 Synthetic Tait computability for phase separated parametricity

For a specific type theory, the explicit construction of the gluing category and the substantiation of the fundamental lemma can be quite complicated. A major contribution of this paper is a synthetic version of type-theoretic gluing that situates type theories and their logical relations in the language of topoi, where we have a wealth of classical results to draw on [AGV72; Joh02]: surprisingly, these classical results suffice to eliminate the explicit and technical constructions of logical relations and their fundamental lemma, replacing them with elementary type-theoretic arguments (Section 3.4.1).

Following the methodology pioneered (in another context) by Orton and Pitts [OP16], we axiomatize the structure required to work synthetically with phase separated proof-relevant logical relations ("parametricity structures"): in Section 3, we specify a dependent type theory ParamTT in which every type can be thought of as a parametricity structure.[5] To substantiate the view of *logical relations as types* we extend ParamTT with the following constructs:

1) A proof-irrelevant proposition $\unlock_{\mathsf{syn}}$ called the *syntactic open*; then, given a synthetic parametricity structure A, we may project the *syntactic part* of A as $\mathsf{Syn}(A) = (\unlock_{\mathsf{syn}} \to A)$. It is easy to see that Syn defines an lex (finite limit preserving) idempotent monad, and furthermore commutes with dependent products; a modality defined in this way is called an *open modality.* Then, a parametricity structure A is called *purely syntactic* if the unit $A \longrightarrow \mathsf{Syn}(A)$ is an isomorphism.
2) A proof-irrelevant proposition $\unlock_{\mathsf{st}}$ called the *static open*; then, given a synthetic parametricity structure A, the *static part* of A is projected by $\mathsf{St}(A) = (\unlock_{\mathsf{st}} \to A)$, and a *purely static* parametricity structure A is one for which $A \longrightarrow \mathsf{St}(A)$ is an isomorphism.
3) An embedding $\lfloor - \rfloor$ of ModTT's syntax as a collection of purely syntactic types and functions, such that for any sort T of ModTT, the static projection commutes with the embedding: $\lfloor \unlock_{\mathsf{st}} \to T \rfloor \cong \unlock_{\mathsf{st}} \to \lfloor T \rfloor$.

We may then form complementary *closed modalities* Sem, Dyn to the open modalities Syn, St that allow one to project the semantic and dynamic parts respectively of a synthetic parametricity structure, as summarized in Figure 2. The explanation of their meaning will have to wait, but we simply note that the "semantic modality" Sem is the universal way to trivialize the syntactic part of a parametricity structure, and the "dynamic modality" Dyn is the universal way to trivialize the static part of a parametricity structure.

[5] The type theory of synthetic parametricity structures will turn out to be the internal language of a certain topos $\mathcal{X}$, to be defined in Section 5.

Synthetic vs. analytic Tait computability Traditional analytic accounts of Tait computability proceed by defining exactly how to *construct* a logical relation out of more primitive things like sets of terms. In contrast, our synthetic viewpoint emphasizes what can be *done* with a logical relation: the syntactic and semantic parts can be extracted and pieced together again. The former primitives, such as sets of terms, then arise as logical relations A such that $A \cong \mathsf{Syn}(A)$.

Just as Euclidean geometry takes lines and circles as primitives rather than point-sets, the synthetic account of Tait computability takes the notion of logical relation as a primitive, characterized by what can be done with it. Perhaps surprisingly, we have found that all aspects of standard computability models can be reconstructed in the synthetic setting in a less technical way.

1.5 Discussion of related work

1.5.1 1ML and F-ing Modules

Most similar in spirit to our module calculus is that of 1ML [Ros18], which, as here, uses a universe to represent a signature of "small" types, which classify run-time values. Although ModTT does not have first class modules, there is no obstacle to supporting the packaging of modules of small signature into a type. 1ML also features a module connective analogous to the static extent, though the universal property of this connective is not explicated — in fact, the rules of equality of modules themselves are not even stated in Rossberg, Russo, and Dreyer [RRD14] and Rossberg [Ros18]. Consequently, the most substantial difference between ModTT and 1ML (aside from the lack of an abstraction theorem) is that the latter is defined by its translation into System Fω, whereas ModTT is given intrinsically as an algebraic theory that expresses equality of modules, with a modality to confine attention to their static parts. To be sure, it is elegant and practical to consider the compilation of modules by a phase-separating translation, as was done for example by Petersen [Pet05]. Nevertheless, it is also important to give a direct type-theoretic account of program modules *as they are to be used and reasoned about.*

1.5.2 Modules, Abstraction, and Parametric Polymorphism

In a pair of recent papers [Cra17; Cra19], Crary develops (1) the relational metatheory of a calculus of ML modules and (2) a fully abstract compilation procedure into a version of System Fω. Although our two calculi have similar expressivity, the rules of ModTT are simpler and more direct; in part, this is because subtyping and retyping are shifted into elaboration for us, but we also remark that Crary has placed side conditions on the rules for dependent sums to ensure they only apply in the non-dependent case, which are unnecessary in ModTT. Crary, however, treats general recursion at the value level, which we have not attempted in this paper. In more recent work Crary [Cra20] joins us in advocating that module projectibility be reconstructed in terms of a lax modality.

Crary's account of parametricity, the first to rigorously substantiate an abstraction theorem for modules, achieves a similar goal to our work, but is much more technically involved. In particular we have gained much leverage from working over equivalence classes of typed terms, rather than using operational semantics on untyped terms — in fact, our entire development proceeds without introducing any technical lemmas whatsoever. Another advantage of our approach is the use of proof relevance to account directly for strong sums over the collection of types; working in a proof-irrelevant setting, Crary must resort to an ingenious staging trick in which classes of precandidates are first defined for every kind, and then the candidates for module signatures are relations between a pair of module values *and* a precandidate. This can be seen as a defunctionalization of the proof-relevant interpretation, and is not likely to scale to more universes.

1.5.3 Internal parametricity

Traditionally (and in this work), parametricity is in essence about counting how many programs of a given type can be *defined*. This purely external notion allows unconditional theorems to be proved about the execution behavior of a program in all definable contexts. In contrast, there has recently been a great deal of interest in *internal parametricity*, which is the extension of type theory by parametricity theorems that previously only held externally [BM12; CH20]. Internal parametricity has many commonalities with our synthetic approach to external parametricity, because the substantiation of parametricity theorems can be carried out in type-theoretic language.

1.5.4 Applicative functor semantics in OCaml

The interaction between effects and module functors lies at the heart of nearly all previous work on modules. Leroy proposed an applicative semantics for module functors [Ler95], later used in OCaml's module system [Ler+20]: whereas generative functors can be thought of as functions $\sigma \Rightarrow \bigcirc\tau$, applicative functors correspond roughly to $\bigcirc(\sigma \Rightarrow \tau)$ as noted by Shao [Sha99], but subtleties abound. The subtleties of applicative and generative functor semantics (studied by Dreyer, Crary, and Harper [DCH03] as *weak* and *strong* sealing) are mostly located in the view of sealing as a computational effect: how can a structure be "pure" if a substructure is sealed? In contrast, we view sealing in the sense of static information loss as a (clearly pure) projection function inserted during typechecking, using the user's signature annotations as a guide. By decoupling sealing from the effect of generating a fresh abstract type, we obtain a simpler and more type-theoretic account of generativity embodied in the lax modality.

1.5.5 Proof-relevant logical relations

We are not the first to consider proof-relevant versions of parametricity; Sojakova and Johann [SJ18] define a general framework for parametric models of System F, which can be instantiated to give rise to a proof-relevant version of parametricity. Benton, Hofmann, and Nigam [BHN13; BHN14] use proof-relevant logical relations to work around the fact that logical relations involving an existential quantifier rarely satisfy an important closure condition known as admissibility, a problem also faced by [Cra17]. In the proof-irrelevant setting this can be resolved either by using continuations explicitly or by imposing a biorthogonal closure condition that amounts to much the same thing.

1.5.6 Computational effects and the Fire Triangle

Lax modalities do not interact cleanly with dependent type structure, unlike the idempotent lex and open modalities of Rijke, Shulman, and Spitters [RSS20]. A promising approach to the integration of real (non-idempotent) effects into dependent type theory is represented by the ∂CBPV calculus of Pédrot and Tabareau [PT19], a dependently typed version of Levy's Call-By-Push-Value [Lev04] that treats a hierarchy of universes of algebras for a given theory in parallel to the ordinary universes of unstructured types. We are optimistic about the potential of ∂CBPV as an improved account of effects in dependent type theory; ∂CBPV's design is motivated by deep syntactical and operational concerns, and we hope in future work to reconcile these with our admittedly category theoretic and semantical viewpoint.

1.5.7 Doubling the syntax

In Section 5 we consider the copower $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ of a topos $\widehat{\mathscr{C}_{\mathbb{T}}}$ representing the syntax of ModTT; this "doubled topos" serves as a suitable index to a gluing construction, yielding a topos $\mathcal{X} = \left(\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S}\right) \sqcup_{2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}} \mathbb{S}$ of phase separated parametricity structures. The fact that doubling the syntax of a suitable type theory preserves all of its structure was noticed and used effectively by Wadler [Wad07]. This same observation lies at the heart of our convenient Notation 3.3 for working synthetically with the left- and right-hand sides of parametricity structures.

1.5.8 Parametricity translations

Related to our synthetic account of logical relations, in which the relatedness of two programs is substantiated by a third program, is the tradition of parametricity translations exemplified by Bernardy, Jansson, and Paterson [BJP12], Pédrot, Tabareau, Fehrmann, and Tanter [Péd+19], and Tabareau, Tanter, and Sozeau [TTS18], also taken up by Per Martin-Löf in his Ernest Nagel Lecture in 2013 [Mar13]. The essential difference is that the parametricity translations are analytic, explicitly transforming types into (proof-relevant) logical relations, whereas our theory of parametricity structures is synthetic: we assume that everything in sight is a logical relation, and then identify the ones that are degenerate in either the syntactic or semantic direction via a modality.

2 ModTT: a type theory for program modules

We introduce ModTT, a type-theoretic core language for program modules based on the considerations discussed in Section 1. We first give an informal presentation of the language using familiar notations in Section 2.1; in Section 2.2, we discuss the formal definition of ModTT in Uemura's logical framework [Uem19].

2.1 Informal presentation of ModTT

2.1.1 Judgmental structure

ModTT is arranged around three basic syntactic classes: contexts $\boxed{\Gamma\ \mathit{ctx}}$, signatures $\boxed{\Gamma \vdash \sigma\ \mathit{sig}}$, modules values $\boxed{\Gamma \vdash V : \sigma}$, and module computations $\boxed{\Gamma \vdash M \div \sigma}$. All judgments presuppose the well-formedness of their constituents; for readability, we omit many annotations that *in fact* appear in a formal presentation of ModTT; furthemore, module signatures, values, and computations are all subject to judgmental equality, and we assume that derivability of all judgments is closed under judgmental equality. These informal assumptions are substantiated by the use of a logical framework to give the "true" definition of ModTT in Section 2.2.

2.1.2 Types and dynamic modules

The simplest module signature is 'type', the signature classifying the object-level types of the programming language, like bool or $s \rightharpoonup t$. Given a module t : type, there is a signature

$\llbracket t \rrbracket$ classifying the values of the type t.

$$\text{TYPE}\;\frac{}{\Gamma \vdash \mathsf{type}\ \mathit{sig}} \qquad \text{DYNAMIC}\;\frac{\Gamma \vdash t : \mathsf{type}}{\Gamma \vdash \llbracket t \rrbracket\ \mathit{sig}}$$

In this section, we do not axiomatize any specific types, though our examples will require them. This choice reflects our (perhaps heterodox) perspective that a programming language is a dynamic extension of a theory of modules, not the other way around.

2.1.3 Computations via lax modality

To reconstruct generativity (Section 2.1.4) in a type theoretic way, we employ a modal separation of effects and distinguish computations of modules from values. This is achieved by means of a strong monad, presented judgmentally as a lax modality $\bigcirc$ mediating between the $\boxed{\Gamma \vdash V : \sigma}$ and $\boxed{\Gamma \vdash M \div \sigma}$ judgments.[6]

$$\text{FORMATION}\;\frac{\Gamma \vdash \sigma\ \mathit{sig}}{\Gamma \vdash \bigcirc\sigma\ \mathit{sig}} \qquad \text{INTRODUCTION}\;\frac{\Gamma \vdash M \div \sigma}{\Gamma \vdash \{M\} : \bigcirc\sigma} \qquad \text{RETURN}\;\frac{\Gamma \vdash V : \sigma}{\Gamma \vdash \mathsf{ret}(V) \div \sigma} \qquad \text{BIND}\;\frac{\Gamma \vdash V : \bigcirc\sigma \qquad \Gamma, X : \sigma \vdash M \div \sigma'}{\Gamma \vdash (X \leftarrow V; M) \div \sigma'}$$

We also include a reduction rule and a commuting conversion corresponding to the monad laws.

2.1.4 Module hierarchies and functors

Signatures in ModTT are closed under dependent sum (module hierarchy) and dependent product (functor), using the standard type-theoretic rules. We display only the formation rules for brevity:

$$\text{DEPENDENT SUM}\;\frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma, X : \sigma \vdash \sigma'\ \mathit{sig}}{\Gamma \vdash [X : \sigma; \sigma']\ \mathit{sig}} \qquad \text{DEPENDENT PRODUCT}\;\frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma, X : \sigma \vdash \sigma'\ \mathit{sig}}{\Gamma \vdash (X : \sigma) \Rightarrow \sigma'\ \mathit{sig}}$$

Generative functors are defined as a mode of use of the dependent product combined with the lax modality, taking $((X : \sigma) \Rightarrow^{\mathsf{gen}} \sigma') := (X : \sigma) \Rightarrow \bigcirc\sigma'$ as in Crary [Cra20].

2.1.5 Contexts and the static open

The usual rules for contexts in Martin-Löf type theories apply, but we have an additional context former $\Gamma, \unlock_{\mathsf{st}}$ called the *static open* context:

$$\frac{}{\cdot\ \mathit{ctx}} \qquad \frac{\Gamma\ \mathit{ctx} \qquad \Gamma \vdash \sigma\ \mathit{sig}}{\Gamma, X : \sigma\ \mathit{ctx}} \qquad \frac{\Gamma\ \mathit{ctx}}{\Gamma, \unlock_{\mathsf{st}}\ \mathit{ctx}}$$

Remark 2.1. The notation is suggestive of the accounts of modal type theory based on dependent right adjoints [Clo+18]; indeed, the context extension $(-, \unlock_{\mathsf{st}})$ can be seen as a modality on contexts left adjoint to a modality on signatures that projects out their static parts.

[6] Semantically, a lax modality is exactly the same thing as a strong monad; at this level, the judgmental distinction between a "value of $\bigcirc\sigma$" and a "computation of σ" is blurred, because one conventionally works up to isomorphism.

The purpose of the static open is to facilitate a context-sensitive version of judgmental equality in which the dynamic parts of different objects are identified when $\Gamma \vdash \text{🔓}_{\mathsf{st}}$. Specifically, we add rules to ensure that programs of a type as well as computations of modules are *statically connected* in the sense of having exactly one element under 🔓_{st}, as in Section 1.2.3.

2.1.6 The static extent

The static open is a tool to ensure that dependency is only incurred on the static parts of objects in ModTT; consequently, we do not include an equality connective or even a general singleton signature (which would incur a dynamic dependency). Instead, we introduce the *static extent* of a static element $\Gamma, \text{🔓}_{\mathsf{st}} \vdash V : \sigma$ as the signature $\{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}$ of modules $U : \sigma$ whose static part restricts to V; because our results depend on the algebraic character of ModTT, we provide explicit introduction and elimination forms for the static extent, which are trivial to elaborate from an implicit notation.

$$\textsc{extent/formation}\quad \frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}\ \mathit{sig}} \qquad \textsc{extent/intro}\quad \frac{\Gamma \vdash W : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash W \equiv V : \sigma}{\Gamma \vdash \mathsf{in}_V(W) : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}}$$

$$\textsc{extent/elim}\quad \frac{\Gamma \vdash V : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow W\}}{\Gamma \vdash \mathsf{out}_W(V) : \sigma} \qquad \textsc{extent/inversion}\quad \frac{\Gamma \vdash \text{🔓}_{\mathsf{st}} \qquad \Gamma \vdash V : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow W\}}{\Gamma \vdash \mathsf{out}_W(V) \equiv W : \sigma}$$

$$\textsc{extent}/\beta\quad \frac{\Gamma \vdash W : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash V : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash W \equiv V : \sigma}{\Gamma \vdash \mathsf{out}_V(\mathsf{in}_V(W)) \equiv W : \sigma}$$

$$\textsc{extent}/\eta\quad \frac{\Gamma, \text{🔓}_{\mathsf{st}} \vdash W : \sigma \qquad \Gamma \vdash V : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow W\}}{\Gamma \vdash V \equiv \mathsf{in}_W(\mathsf{out}_W(V)) : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow W\}}$$

The static extent reconstructs both type sharing and weak structure sharing, which appear in SML '97 [Mil+97] and OCaml [Ler+20].

Example 2.2. The SML module signature (`SHOW where type t = bool`) is rendered in terms of the static extent as $\{\mathtt{SHOW} \mid \text{🔓}_{\mathsf{st}} \hookrightarrow [\mathsf{bool}, *]\}$, using the STATIC CONNECTIVITY (1) rule from Section 1.2.3:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \text{🔓}_{\mathsf{st}} \vdash \mathsf{bool} : \mathsf{type}} \qquad \dfrac{\dfrac{}{\Gamma, \text{🔓}_{\mathsf{st}} \vdash \text{🔓}_{\mathsf{st}}}}{\Gamma, \text{🔓}_{\mathsf{st}} \vdash * : \langle\!|\mathsf{bool} \rightharpoonup \mathtt{string}|\!\rangle}}{\Gamma, \text{🔓}_{\mathsf{st}} \vdash [\mathsf{bool}, *] : \mathtt{SHOW}}}{\Gamma \vdash \{\mathtt{SHOW} \mid \text{🔓}_{\mathsf{st}} \hookrightarrow [\mathsf{bool}, *]\}\ \mathit{sig}}$$

We have (intentionally) made no effort to restrict the families of signatures to depend only on variables of a static nature, in contrast to previous works on modules. We conjecture, but do not prove here, the admissibility of a principle that extends any signature to one that is defined over a purely static context. This should follow, roughly, from the fact that genuine dependencies are all introduced ultimately via the static extent and that there is no signature of signatures. We note that none of the results of this paper depend on the validity of this conjecture.

2.1.7 Further extensions: observables and partial function types

For brevity, we do not extend ModTT with all the features one would expect from a programming language. However, our examples will require a type of observables $\mathsf{bool} : \mathsf{type}$ with $\#\mathsf{t}, \#\mathsf{f} : \langle\!|\mathsf{bool}|\!\rangle$, as well as a partial function type $s \rightharpoonup t$ such that $\langle\!|s \rightharpoonup t|\!\rangle \cong \langle\!|s|\!\rangle \Rightarrow \bigcirc\langle\!|t|\!\rangle$.

2.1.8 External language and elaboration

We do not present here a surface language; such a language would include many features not present in the core language ModTT: for instance, named fields and paths are elaborated to iterated dependent sum projections, and SML-style sharing constraints and '`where type`' clauses are elaborated to uses of the static extent. Elaboration is essential to support the implicit dropping and reordering of fields in module signature matching; furthermore, the crucial subtyping and extensional retyping principles of Lee, Crary, and Harper [LCH07] are re-cast as an elaboration strategy guided by η-laws, as in the elaboration of extension types in the `cooltt` proof assistant [Red20]. The status of subtyping and retyping in ModTT is a significant divergence from previous work, which treated them within the core language (an untenable position for an algebraic account of modules).

2.2 Algebraic presentation in a logical framework

Uemura has defined a dependently sorted equational logical framework with support for one level of variable binding, which may be used to define almost any kind of type theory whose contexts enjoy all the structural rules. We defer to Uemura [Uem19] for a full explication of the details, but we may briefly summarize Uemura's LF as follows:

1) There is a universe **Jdg** of *judgments*, and a subuniverse $\mathbf{Ctx} \subseteq \mathbf{Jdg}$ of *representable judgments*, or judgments that can be represented by (object-level) contexts. For example, the term typing judgment $a : \mathrm{A}$ is usually representable because we have context extensions $\Gamma, x : \mathrm{A}$; on the other hand, the typehood judgment A *type* is not usually representable, because we do not have context extensions Γ, X *type*.
2) The judgments are closed under dependent products (hypothetical judgments) whose base is representable: so if $\mathrm{X} : \mathbf{Ctx}$ and $\mathrm{Y} : \mathrm{X} \to \mathbf{Jdg}$, then $(x : \mathrm{X}) \to \mathrm{Y}(x) : \mathbf{Jdg}$. Both **Jdg**, **Ctx** are closed under arbitrary pullback/substitution and dependent sum.[7]
3) The judgments are closed under extensional equality: $(a =_{\mathrm{X}} b)$ is a judgment for each $a, b : \mathrm{X}$.

A *signature* in the LF is given a dependent telescope of constants whose sorts are given by formal dependent products of judgments along judgments; the LF signature of ModTT is presented in Figure 3.

Definition 2.3 (Algebras for a signature)**.** Let Σ be a signature in the LF; the signature Σ can be viewed as a "dependent record type" in any sufficiently structured category $\mathscr{E}$. In particular, if $\mathscr{U}$ is a universe in $\mathscr{E}$ closed under dependent sum, product, and extensional equality, we have a type $\mathbf{Alg}_\Sigma(\mathscr{U})$ in $\mathscr{E}$ defined as the dependent sum of all of the components of Σ where **Ctx**, **Jdg** are interpreted as $\mathscr{U}$; an element of $\mathbf{Alg}_\Sigma(\mathscr{U})$ is then a model of the theory presented by Σ, in which judgments and contexts are $\mathscr{U}$-small.

[7] The dependent sum condition is reflected in Uemura's presentation by the use of telescopes for the parameters to declarations; however, a first-class dependent sum connective is easily accomodated by Uemura's categorical semantics in representable map categories.

$$
\begin{aligned}
\text{🔓}_{\mathsf{st}} &: \mathbf{Ctx} \\
\_ &: (x, y : \text{🔓}_{\mathsf{st}}) \to x =_{\text{🔓}_{\mathsf{st}}} y \\
\mathsf{Sig} &: \mathbf{Jdg} \\
\mathsf{Val} &: \mathsf{Sig} \to \mathbf{Ctx} \\
\mathsf{type} &: \mathsf{Sig} \\
\langle\!|{-}|\!\rangle &: \mathsf{Val}(\mathsf{type}) \to \mathsf{Sig} \\
\Pi, \Sigma &: (\sigma : \mathsf{Sig}) \to (\mathsf{Val}(\sigma) \to \mathsf{Sig}) \to \mathsf{Sig} \\
\mathsf{Ext} &: (\sigma : \mathsf{Sig}) \to (\text{🔓}_{\mathsf{st}} \to \mathsf{Val}(\sigma)) \to \mathsf{Sig} \\
\bigcirc &: \mathsf{Sig} \to \mathsf{Sig} \\
\Pi/\mathsf{val} &: \big((x : \mathsf{Val}(\sigma)) \to \mathsf{Val}(\tau(x))\big) \cong \mathsf{Val}(\Pi(\sigma, \tau)) \\
\Sigma/\mathsf{val} &: \big((x : \mathsf{Val}(\sigma)) \times \mathsf{Val}(\tau(x))\big) \cong \mathsf{Val}(\Sigma(\sigma, \tau)) \\
\mathsf{Ext}/\mathsf{val} &: \big((\mathrm{U} : \mathsf{Val}(\sigma)) \times ((z : \text{🔓}_{\mathsf{st}}) \to \mathrm{U} =_{\mathsf{Val}(\sigma)} \mathrm{V}(z))\big) \cong \mathsf{Val}(\mathsf{Ext}(\sigma, \mathrm{V})) \\
\mathsf{Cmp} &: \mathsf{Sig} \to \mathbf{Ctx} \\
\mathsf{Cmp} &:= \lambda\sigma.\mathsf{Val}(\bigcirc\sigma) \\
\mathsf{conn}/\mathsf{dyn} &: \text{🔓}_{\mathsf{st}} \to \mathsf{Val}(\langle\!| t |\!\rangle) \cong \mathbf{1} \\
\mathsf{conn}/\mathsf{cmp} &: \text{🔓}_{\mathsf{st}} \to \mathsf{Cmp}(\sigma) \cong \mathbf{1} \\
\mathsf{ret} &: \mathsf{Val}(\sigma) \to \mathsf{Cmp}(\sigma) \\
\mathsf{bind} &: (\mathsf{Val}(\sigma) \to \mathsf{Cmp}(\tau)) \to \mathsf{Cmp}(\sigma) \to \mathsf{Cmp}(\tau) \\
\_ &: \mathsf{bind}(\mathrm{F}, \mathsf{ret}(\mathrm{V})) =_{\mathsf{Cmp}(\tau)} \mathrm{F}(\mathrm{V}) \\
\_ &: \mathsf{bind}(\mathrm{F}, \mathsf{bind}(\mathrm{G}, \mathrm{V})) =_{\mathsf{Cmp}(\rho)} \mathsf{bind}(\lambda x.\mathsf{bind}(\mathrm{F}, \mathrm{G}(x)), \mathrm{V}) \\
(\rightharpoonup) &: \mathsf{Val}(\mathsf{type}) \to \mathsf{Val}(\mathsf{type}) \to \mathsf{Val}(\mathsf{type}) \\
\rightharpoonup/\mathsf{val} &: \big(\mathsf{Val}(\langle\!| s |\!\rangle) \to \mathsf{Cmp}(\langle\!| t |\!\rangle)\big) \cong \mathsf{Val}(\langle\!| s \rightharpoonup t |\!\rangle) \\
\mathsf{bool} &: \mathsf{Val}(\mathsf{type}) \\
\#\mathsf{t}, \#\mathsf{f} &: \mathsf{Val}(\langle\!| \mathsf{bool} |\!\rangle) \\
\mathsf{if} &: \mathsf{Val}(\langle\!| \mathsf{bool} |\!\rangle) \to \mathsf{Cmp}(\langle\!| t |\!\rangle) \to \mathsf{Cmp}(\langle\!| t |\!\rangle) \to \mathsf{Cmp}(\langle\!| t |\!\rangle) \\
\_ &: \mathsf{if}(\#\mathsf{t}, \mathrm{M}, \mathrm{N}) =_{\mathsf{Cmp}(\langle\!| t |\!\rangle)} \mathrm{M} \\
\_ &: \mathsf{if}(\#\mathsf{f}, \mathrm{M}, \mathrm{N}) =_{\mathsf{Cmp}(\langle\!| t |\!\rangle)} \mathrm{N}
\end{aligned}
$$

Figure 3: The explicit presentation of ModTT as a signature $\mathbb{T}$ in the logical framework; for readability, we omit quantification over certain metavariables. The introduction, elimination, computation, and uniqueness rules of the static extent are captured in a *single* rule declaring an isomorphism; declarations of this form are a definitional extension of Uemura's LF, because they always boil down to four elementary declarations.

Syntactic category of an LF signature A signature Σ in the LF *presents* a certain category $\mathscr{C}_\Sigma$ equipped with all finite limits and some dependent products — in the sense that there is a bijection between equivalence classes of LF terms and morphisms in the category. The objects of $\mathscr{C}_\Sigma$ are equivalence classes of judgments over Σ, and the morphisms are equivalence classes of deductions.

The notion of an algebra (Definition 2.3) is good for concrete constructions, but the higher-altitude structure of a development is best served by *functorial semantics* in the spirit of Lawvere [Law63]. A model of Σ in a sufficiently structured category $\mathscr{E}$ can be viewed in two ways:

1) A model is an element of $\mathbf{Alg}_\Sigma(\mathscr{U})$ for some universe $\mathscr{U}$ in $\mathscr{E}$.
2) A model is a structure preserving functor $\mathscr{C}_\Sigma \longrightarrow \mathscr{E}$.

We will use both perspectives in this paper. The induction principle or universal property of the syntax states that $\mathscr{C}_\Sigma$ is the smallest model of Σ; this universal property is the main ingredient for proving syntactic metatheorems by semantic means, as we advocate and apply in this paper.

Remark 2.4. We make no use of Uemura's more sophisticated notion of a *model of a type theory* [Uem19], nor of his notion of a *theory over a type theory*; we find it easier to work directly with the universal property of the type theory generated by a given signature in the LF.

Notation 2.5. We will write $\mathbb{T}$ for the signature presenting ModTT in Figure 3, and $\mathscr{C}_\mathbb{T}$ for the syntactic category of ModTT.

Equational presentation of specific effects It is important that our use of an equational logical framework does not prevent the extension of ModTT with non-trivial computational effects; although the effect of having a fixed collection of references cells or exceptions is clearly algebraic (see e.g. Plotkin and Power [PP02]), an equational and structural account of fresh names or nominal restriction is needed in order to account for languages that feature allocation.

An equational presentation of allocation may be achieved along the lines of Staton [Sta13] — as Staton's work shows, there is no obstacle to the equational presentation of *any* reasonable form of effect, but semantics are another story. We do not currently make any claim about the extension of our representation independence results to the setting of higher-order store, for instance.

3 A type theory for synthetic parametricity

Our goal is to define a "type theory of parametricity structures" ParamTT, in which the analytic view of logical relations (as a pair of a syntactic object together with a relation defined on its elements) is replaced by a streamlined synthetic perspective, captured under the slogan **logical relations as types**. Combined with a model construction detailed in Section 5, the results of this section will imply a generalized version of the Reynolds abstraction theorem [Rey83] for ModTT stated in Corollary 5.17.

ParamTT is an extension of the internal dependent type theory of a presheaf topos with modal features corresponding to phase separated parametricity: therefore, ParamTT has dependent products, dependent sums, extensional equality types, a strictly univalent universe Ω of proof irrelevant propositions, a strict hierarchy of universes $\mathcal{U}_\alpha$ of types, inductive types, subset types, and effective quotient types (consequently, strict pushouts). We first axiomatize ParamTT in the style of Orton and Pitts [OP16], and in Section 5 we construct a suitable model of ParamTT using topos theory. Referring to the types of ParamTT, we will often speak of "parametricity structures".

3.1 Modal structure of iterated phase separation

Using the insight that logical relations can be seen as a kind of phase distinction between the syntactic and the semantic, we iterate the use of the "static open" from ModTT and add to ParamTT a system of proof irrelevant propositions corresponding to the static part and the disjoint (left)-syntactic and (right)-syntactic parts of a parametricity structure.

$$\text{🔓}_{\mathsf{st}}, \text{🔓}_{\mathsf{syn/l}}, \text{🔓}_{\mathsf{syn/r}}, \text{🔓}_{\mathsf{syn}} : \Omega \qquad \text{🔓}_{\mathsf{syn/l}} \wedge \text{🔓}_{\mathsf{syn/r}} = \bot \qquad \text{🔓}_{\mathsf{syn}} := \text{🔓}_{\mathsf{syn/l}} \vee \text{🔓}_{\mathsf{syn/r}}$$

3.1.1 Static and syntactic open modalities

Using the propositions specified above, we may define *open modalities* that isolate the static and syntactic aspects of a given type.

Construction 3.1 (Open modality)**.** If $\phi : \Omega$ is a proposition, then the *open modality* corresponding to ϕ is $\mathsf{Open}_\phi(A) := \phi \to A$. One observes that the open modality has the following properties:

1) It is monadic: indeed, it is the "reader monad" for the proposition ϕ.
2) It is idempotent, in the sense that $\mathsf{Open}_\phi(\mathsf{Open}_\phi A) \cong \mathsf{Open}_\phi(A)$.
3) It is left exact ("lex" for short), in the sense that $\mathsf{Open}_\phi(a =_A b)$ is isomorphic to $(\lambda\_.a) =_{\mathsf{Open}_\phi(A)} (\lambda\_.b)$.
4) It commutes with exponentials, in the sense that $\mathsf{Open}_\phi(A \to B)$ is isomorphic to $\mathsf{Open}_\phi(A) \to \mathsf{Open}_\phi(B)$.

Definition 3.2. When M is an idempotent modality, we say that a type A is M-*modal* when the unit map $\eta : A \longrightarrow M(A)$ is an isomorphism; a type A is called M-*connected* when $M(A) \cong \mathbf{1}$.

We define the "static modality" to be $\mathsf{St} := \mathsf{Open}_{\text{🔓}_{\mathsf{st}}}$ and the "syntactic modality" to be $\mathsf{Syn} := \mathsf{Open}_{\text{🔓}_{\mathsf{syn}}}$; the notion of a Open_ϕ-modal type gives us an abstract way to speak of types that are purely syntactic or purely static (or both).

Our open modalities isolate the static and syntactic parts of a parametricity structure respectively; because $\text{🔓}_{\mathsf{syn/l}}, \text{🔓}_{\mathsf{syn/r}}$ have no overlap, we have an isomorphism $\mathsf{Syn}(A) \cong (\text{🔓}_{\mathsf{syn/l}} \to A) \times (\text{🔓}_{\mathsf{syn/r}} \to A)$. This isomorphism is captured more generally by the following *systems* notation of Cohen, Coquand, Huber, and Mörtberg [Coh+17] from cubical type theory for constructing maps out of disjunctions of propositions:

Notation 3.3 (Systems)**.** Following Cohen, Coquand, Huber, and Mörtberg [Coh+17], we employ the notation of *systems* for constructing elements of parametricity structures underneath the assumption of disjunction of propositions $\phi \vee \phi'$: when $\phi \wedge \phi'$ implies $a = a' : A$, we may write $[\phi \hookrightarrow a \mid \phi' \hookrightarrow a']$ for the unique element of A that restricts to a, a' on ϕ, ϕ' respectively.

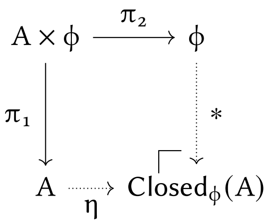
Notation 3.4 (Extension)**.** As foreshadowed by the static extents of ModTT, every proposition $\phi : \Omega$ gives rise to an *extension type* connective [RS17]: if A is a parametricity structure and a is an element of A assuming ϕ is true, then $\{A \mid \phi \hookrightarrow a\}$ is the parametricity structure of elements $a' : A$ such that $a = a'$ when ϕ is true.

3.1.2 Dynamic and semantic closed modalities

The static modality forgets the dynamic part of a parametricity structure (in both syntax and semantics), and the syntactic modality forgets the semantic part of a parametricity structure. We will require complementary modalities to do the opposite, e.g. form a parametricity structure with no syntactic force.

Construction 3.5 (Closed modality)**.** If $\phi : \Omega$ is a proposition, then the *closed* modality Closed_ϕ complementing the open modality $\mathsf{Open}_\phi = (\phi \to -)$ can be defined as a quotient of the product $A \times \phi$ or as a pushout. We define Closed_ϕ in both type theoretic and categorical notation below:

```
data Closedϕ (A : 𝒰) where
  η : A → Closedϕ(A)
  * : ϕ → Closedϕ(A)
  _ : ∏a:A ∏z:ϕ (η(a) = *(z))
```



The modality Closed_ϕ is lex, idempotent, and monadic, but it does not usually commute with exponentials.

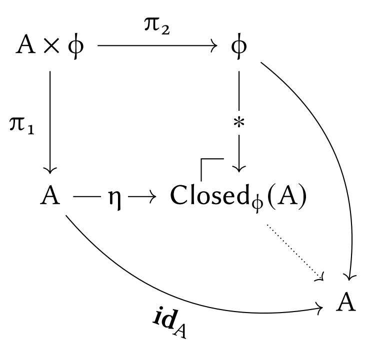
Using Construction 3.5, we define the "purely semantic" and "purely dynamic" modalities respectively:

$$\mathsf{Sem} := \mathsf{Closed}_{\text{🔓}_{\mathsf{syn}}}$$
$$\mathsf{Dyn} := \mathsf{Closed}_{\text{🔓}_{\mathsf{st}}}$$

Lemma 3.6. For any $\phi : \Omega$, then a type A is Closed_ϕ-modal if and only if it is Open_ϕ-connected.

Proof. Suppose that A is Closed_ϕ-modal; to show that A is Open_ϕ-connected, it therefore suffices to show that $\mathsf{Open}_\phi(\mathsf{Closed}_\phi(A)) \cong \mathbf{1}$, which is to say that there is a unique morphism $\phi \longrightarrow \mathsf{Closed}_\phi(A)$ given by the constructor $* : \phi \longrightarrow \mathsf{Closed}_\phi(A)$. This is clear using the induction principle of $\mathsf{Closed}_\phi(A)$, since the quotienting ensures that $\eta(a) = *(z)$ for any $a : A, z : \phi$.

In the other direction, suppose that A is Open_ϕ-connected; we must check that the unit constructor $A \longrightarrow \mathsf{Closed}_\phi(A)$ is an isomorphism. We construct the inverse, as follows, noting that the Open_ϕ-connectedness of A immediately induces a *unique* morphism $\phi \longrightarrow A$:



We see that $\mathsf{Closed}_\phi(A) \longrightarrow A$ is a retraction of the unit, and it remains to check that it is a section; this follows immediately from the universal property (i.e. the η-law) of the pushout. ◻

Instantiating Lemma 3.6, we see the sense in which the pairs of modalities Syn/Sem and St/Dyn are each complementary: in particular, we have $\mathsf{Syn}(\mathsf{Sem}(A)) \cong \mathbf{1}$ and $\mathsf{St}(\mathsf{Dyn}(A)) \cong \mathbf{1}$. Put more crudely, a "dynamic thing has no static component" and a "semantic thing has no syntactic component".

3.2 Universes of modal types

Each universe $\mathscr{U}_\alpha$ of ParamTT may be restricted to a universe consisting of *modal* types for each modality described above, e.g. a universe of purely syntactic types or purely dynamic types. Fixing a lex idempotent modality **M**, thought to be ranging over {Syn, Sem, St, Dyn}, we might naïvely consider defining the universe $\mathscr{U}^\alpha_{\mathbf{M}}$ of **M**-modal types as a subtype:

$$\mathscr{U}^\alpha_{\mathbf{M}} := \{A : \mathscr{U}_\alpha \mid A \cong \mathbf{M}(A)\} \qquad \text{(bad)}$$

Unfortunately, such a universe will not itself be **M**-modal, i.e. we do not have $\mathbf{M}(\mathscr{U}^\alpha_{\mathbf{M}}) \cong \mathscr{U}^\alpha_{\mathbf{M}}$, hence there is no hope of closing the **M**-modal fragment of ParamTT under a hierarchy of universes with such a definition.[8] An idea pioneered in a different context by Streicher [Str05] is to apply the modality directly to the universe:

$$\mathscr{U}^\alpha_{\mathbf{M}} := \mathbf{M}(\mathscr{U}_\alpha) \qquad \text{(good)}$$

With such a definition, we immediately have $\mathbf{M}(\mathscr{U}^\alpha_{\mathbf{M}}) \cong \mathscr{U}^\alpha_{\mathbf{M}}$, etc.; but we still have to specify the decodings of these new universes, which is to explain what the type of elements of modal universe is. This can be done systematically for any modality **M**, so long as **M** preserves the universe level of types. Categorically, one views the universe $\mathscr{U}_\alpha$ as a *generic family* $\pi : \sum_{A:\mathscr{U}_\alpha} A \longrightarrow \mathscr{U}_\alpha$ that expresses the indexing of elements over types. The insight of Streicher [Str05] was to apply the modality **M** to the entire generic family yielding $\mathbf{M}(\pi) : \mathbf{M}(\sum_{A:\mathscr{U}_\alpha} A) \longrightarrow \mathscr{U}^\alpha_{\mathbf{M}}$, and then obtain the collection of elements of a given $A : \mathscr{U}^\alpha_{\mathbf{M}}$ by pullback.

[8] The "naïve" definition considered here *does* work in homotopy type theories in the presence of the univalence principle, as shown by Rijke, Shulman, and Spitters [RSS20]; because we are working strictly in ordinary 1-dimensional mathematics, we must choose a different (but homotopically equivalent) definition of the universe of modal types.

In more type theoretic language, the collection of elements of $A : \mathscr{U}^{\alpha}_{\mathbf{M}}$ is given by the following decoding map:

$$\mathsf{El}_{\mathbf{M}} : \mathscr{U}^{\alpha}_{\mathbf{M}} \to \mathscr{U}_{\alpha}$$
$$\mathsf{El}_{\mathbf{M}}(A) := \left\{x : \mathbf{M}\left(\textstyle\sum_{X:\mathscr{U}_{\alpha}} X\right) \,\middle|\, \mathbf{M}(\pi)(x) = A\right\}$$

We note that each modal universe $\mathscr{U}^{\alpha}_{\mathbf{M}}$ is closed under all the connectives of ParamTT, a general fact about lex idempotent modalities in topos theory [MM92] and type theory [RSS20].

Lemma 3.7. If $A : \mathscr{U}_{\alpha}$, then $\mathsf{El}_{\mathbf{M}}(\eta_{\mathbf{M}}(A)) \cong \mathbf{M}(A)$

Lemma 3.8. In the case of the open modality for a proposition $\phi : \Omega$, there is a simpler computation of the decoding of the universe $\mathscr{U}^{\alpha}_{\mathsf{Open}_{\phi}}$,

$$\mathsf{El}_{\mathsf{Open}_{\phi}} : \mathscr{U}^{\alpha}_{\mathsf{Open}_{\phi}} \to \mathscr{U}_{\alpha}$$
$$\mathsf{El}_{\mathsf{Open}_{\phi}}(A) \cong \textstyle\prod_{z:\phi} A(z)$$

Notation 3.9. From Lemma 3.7, we are inspired to adopt a slight abuse of notation: when $A : \mathscr{U}_{\alpha}$, we will often write $\mathbf{M}(A) : \mathscr{U}^{\alpha}_{\mathbf{M}}$ to mean $\eta_{\mathbf{M}}(A)$; we will also leave $\mathsf{El}_{\mathbf{M}}$ implicit, since we have already indulged the notational fiction of universes à la Russell.

3.2.1 Strictification and syntactic realignment

We assert that the universe hierarchies of ParamTT moreover satisfy the following *strictification* axiom of Orton and Pitts [OP16], which we will justify by a model construction in Section 5.

Axiom 3.10 (Strictification). Let $\phi : \Omega$ be a proposition, and let $A : \phi \to \mathscr{U}_{\alpha}$ be a partial type defined on the extent of ϕ, and let $B : \mathscr{U}_{\alpha}$ be a total type. Now suppose we have a partial isomorphism $f : \prod_{x:\phi}(A(x) \cong B)$; then there exists a total type B' with $g : B' \cong B$, such that both $\forall x : \phi.B' = A(x)$ and $\forall x : \phi.f(x) = g$ strictly.

Axiom 3.10 above plays a critical role in the constructions of Section 3.4, letting $\phi := \blacksquare_{\mathsf{syn}}$.

Corollary 3.11 (Realignment). Let $A : \mathscr{U}^{\alpha}_{\mathsf{Syn}}$ be a syntactic type, and fix $\tilde{A} : \mathscr{U}_{\alpha}$ whose syntactic part is isomorphic to A, i.e. we have $f : \mathsf{Syn}(A \cong \tilde{A})$. Then there exists a type $f^{*}\tilde{A} : \mathscr{U}_{\alpha}$ with $f^{\dagger}\tilde{A} : f^{*}\tilde{A} \cong \tilde{A}$, such that both $\mathsf{Syn}(f^{*}\tilde{A} = A)$ and $\mathsf{Syn}(f^{\dagger}\tilde{A} = f)$ strictly.

3.3 Doubled embedding of syntax

We need to embed the syntax of ModTT into the syntactic fragment of ParamTT. This is done by assuming a $\mathbb{T}$-algebra valued in a universe $\mathscr{U}_{\mathsf{Syn}}$ of purely syntactic types, i.e. an element $\mathcal{A}_{\mathsf{Syn}} : \mathbf{Alg}_{\mathbb{T}}(\mathscr{U}_{\mathsf{Syn}})$. Because we have specified $\blacksquare_{\mathsf{syn}} = \blacksquare_{\mathsf{syn/l}} \vee \blacksquare_{\mathsf{syn/r}}$, we also obtain "left-syntactic" and "right-syntactic" algebras $\mathcal{A}_{\mathsf{L}}, \mathcal{A}_{\mathsf{R}}$ respectively such that $\mathcal{A}_{\mathsf{Syn}} = [\blacksquare_{\mathsf{syn/l}} \hookrightarrow \mathcal{A}_{\mathsf{L}} \mid \blacksquare_{\mathsf{syn/r}} \hookrightarrow \mathcal{A}_{\mathsf{R}}]$.

Notation 3.12 (Syntactic embedding). The algebra $\mathcal{A}_{\mathsf{Syn}} : \mathbf{Alg}_{\mathbb{T}}(\mathscr{U}_{\mathsf{Syn}})$ determines, by projection, an object corresponding to each piece of syntax definable in ModTT. For instance, object of ModTT-signatures is obtained by the projection $\mathcal{A}_{\mathsf{Syn}}.\mathsf{Sig} : \mathscr{U}_{\mathsf{Syn}}$. To lighten the notation we will write these projections informally as $\lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$, etc., writing $\lfloor \mathsf{Sig} \rfloor_{\mathsf{L}}$, $\lfloor \mathsf{Sig} \rfloor_{\mathsf{R}}$ for the corresponding projections from the induced left-syntactic and right-syntactic algebras respectively.

To complete our axiomatization of the embedding of ModTT into ParamTT, we additionally require that under the assumption of 🔓_{syn}, we have $\text{🔓}_{\mathsf{st}} = \lfloor \text{🔓}_{\mathsf{st}} \rfloor_{\mathsf{Syn}}$; in other words, we require $\text{🔓}_{\mathsf{st}} : \{\Omega \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \text{🔓}_{\mathsf{st}} \rfloor_{\mathsf{Syn}}\}$.

3.4 A parametric model of ModTT in ParamTT

In this section, we exhibit a second algebra for ModTT in ParamTT that lies over the doubled embedding described in Section 3.3. To be precise, we will construct an algebra with the following "syntactic extent" type for some sufficiently large universe $\mathscr{U}$:

$$\mathcal{A} : \{\mathbf{Alg}_{\mathbb{T}}(\mathscr{U}) \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathcal{A}_{\mathsf{Syn}}\}$$

We do not show every part of the construction of this "parametric algebra", but instead give several representative cases to illustrate the comparative ease of our approach in contrast to prior work on proof relevant logical relations [SAG19; SA20; Coq19; KHS19] and conventional logical relations [Cra17; GSB19; Ang19] for dependent types.

3.4.1 Parametricity structure of judgments

We define a parametricity structure of signatures over the purely syntactic parametricity structure of syntactic signatures $\lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$. Letting $\alpha < \beta < \gamma$, we define $\mathsf{Sig} : \mathscr{U}_\beta$ with the following interface:

$$\begin{aligned} &\mathsf{Sig} : \{\mathscr{U}_\gamma \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}\} \\ &\mathsf{Sig} \cong \textstyle\sum_{\sigma : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}} \{\mathscr{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma)\} \end{aligned}$$

The construction of Sig proceeds in the following way. First, we define Sig' to be the dependent sum $\sum_{\sigma : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}} \{\mathscr{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma)\}$. We observe that there is a canonical partial isomorphism $f : \mathsf{Syn}(\mathsf{Sig}' \cong \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}})$; supposing $\text{🔓}_{\mathsf{syn}} = \top$, it suffices to construct an ordinary isomorphism:

$$\begin{aligned} \mathsf{Sig}' &= \textstyle\sum_{\sigma : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}} \{\mathscr{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma)\} && \text{def. of } \mathsf{Sig}' \\ &= \textstyle\sum_{\sigma : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}} \{\mathscr{U}_\beta \mid \top \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma)\} && \text{🔓}_{\mathsf{syn}} = \top \\ &\cong \textstyle\sum_{\sigma : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}} \mathbf{1} && \text{singleton} \\ &\cong \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}} && \text{trivial} \end{aligned}$$

Therefore, by Corollary 3.11 we obtain $\mathsf{Sig} \cong \mathsf{Sig}'$ strictly extending $\lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$ as desired. Next, we may define the collection of elements of a glued signature directly:

$$\begin{aligned} &\mathsf{Val} : \{\mathsf{Sig} \to \mathscr{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}\} \\ &\mathsf{Val}(\sigma, \tilde{\sigma}) = \tilde{\sigma} \end{aligned}$$

3.4.2 Parametricity structure of dependent products

We show that Sig is closed under dependent product (dependent sums are analogous); fixing $\sigma_0 : \mathsf{Sig}$ and $\sigma_1 : \mathsf{Val}(\sigma_0) \to \mathsf{Sig}$, we may define $\Pi_{\mathsf{Sig}}(\sigma_0, \sigma_1) : \mathsf{Sig}$ as follows. We desire the first component to be the syntactic dependent product type $\sigma_\Pi = \lfloor \Pi_{\mathsf{Sig}} \rfloor_{\mathsf{Syn}}(\sigma_0, \lambda x : \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma_0).\sigma_1(x))$.[9] For the second component, we note that the syntactic modality commutes with dependent products up to isomorphism, so (using Corollary 3.11) we may define

[9]We note that we always have $\text{🔓}_{\mathsf{syn}} = \top$ in scope when constructing an element of $\lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$.

the second component lying strictly over σ_Π:

$$\begin{aligned}\tilde{\sigma}_\Pi &: \{\mathscr{U}_\beta \mid \unlock_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma_\Pi)\} \\ \tilde{\sigma}_\Pi &\cong \Pi_{\mathscr{U}_\beta}(\mathsf{Val}(\sigma_0), \mathsf{Val} \circ \sigma_1)\end{aligned}$$

Because we used the dependent product $\Pi_{\mathscr{U}_\beta}$ of ParamTT, we automatically have an appropriate model of the λ-abstraction, application, computation, and uniqueness rules without further work.

Remark 3.13. The parametricity structure of the dependent product is the "proof" that our synthetic approach is a big step forward (e.g. compared to the explicit constructions of Kaposi, Huber, and Sattler [KHS19] and Sterling and Angiuli [SA20]). In those formulations one constantly uses the fact that the gluing functor preserves finite limits, and it is non-trivial to show that the resulting construction is in fact a dependent product (which is here made trivial). The work did not disappear: it is in fact located in several pages of SGA 4, in which certain comma categories are proved to satisfy the Giraud axioms of a category of sheaves [AGV72], a result that is easier to prove in generality than any specific type theoretic corollary.

3.4.3 Parametricity structure of types

From the syntax of ModTT, we have the signature of types $\lfloor \mathsf{type} \rfloor_{\mathsf{Syn}} : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$ and its decoding $\llparenthesis - \rrparenthesis_{\mathsf{Syn}} : \lfloor \mathsf{Val}(\mathsf{type}) \to \mathsf{Sig} \rfloor_{\mathsf{Syn}}$; we must provide parametricity structures for both. First, we may define a collection of small statically connected parametricity structures for types, using Corollary 3.11:

$$\begin{aligned}\mathbf{Type} &: \{\mathscr{U}_\beta \mid \unlock_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val}(\mathsf{type}) \rfloor_{\mathsf{Syn}}\} \\ \mathbf{Type} &\cong \textstyle\sum_{t : \lfloor \mathsf{Val}(\mathsf{type}) \rfloor_{\mathsf{Syn}}} \{\mathscr{U}^{\alpha}_{\mathsf{Dyn}} \mid \unlock_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}} \llparenthesis t \rrparenthesis_{\mathsf{Syn}}\}\end{aligned}$$

We may therefore construct the parametricity structure of the signature of types:

$$\begin{aligned}&\mathsf{type} : \{\mathsf{Sig} \mid \unlock_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{type} \rfloor_{\mathsf{Syn}}\} && \llparenthesis - \rrparenthesis : \{\mathsf{Val}(\mathsf{type}) \to \mathsf{Sig} \mid \unlock_{\mathsf{syn}} \hookrightarrow \llparenthesis - \rrparenthesis_{\mathsf{Syn}}\} \\ &\mathsf{type} = (\lfloor \mathsf{type} \rfloor_{\mathsf{Syn}}, \mathbf{Type}) && \llparenthesis (t, \tilde{t}) \rrparenthesis = (\llparenthesis t \rrparenthesis_{\mathsf{Syn}}, \tilde{t})\end{aligned}$$

3.4.4 Parametricity structure of observables

We have a type $\mathsf{bool} : \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\lfloor \mathsf{type} \rfloor_{\mathsf{Syn}})$ and two constants $\#\mathsf{t}, \#\mathsf{f} : \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\llparenthesis \lfloor \mathsf{bool} \rfloor_{\mathsf{Syn}} \rrparenthesis)$; we must construct parametricity structures for all these. First, we define the collection of computable booleans as follows, using Corollary 3.11 as usual:[10]

$$\begin{aligned}\mathbf{bool} &: \{\mathscr{U}_\alpha \mid \lfloor \mathsf{Val}(\llparenthesis \mathsf{bool} \rrparenthesis) \rfloor_{\mathsf{Syn}}\} \\ \mathbf{bool} &\cong \textstyle\sum_{b : \lfloor \mathsf{Val}(\llparenthesis \mathsf{bool} \rrparenthesis) \rfloor_{\mathsf{Syn}}} \mathsf{Dyn}(\mathsf{Sem}(\{\tilde{b} : 2 \mid b = \mathsf{case}[\tilde{b}](\lfloor \#\mathsf{t} \rfloor_{\mathsf{Syn}}, \lfloor \#\mathsf{f} \rfloor_{\mathsf{Syn}})\}))\end{aligned}$$

The application of the closed modality Dyn ensures that the values of observable type have no static part (they are "statically connected"). We may therefore define the type of booleans:

$$\mathsf{bool} : \{\mathsf{Val}(\mathsf{type}) \mid \unlock_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{bool} \rfloor_{\mathsf{Syn}}\}$$

[10] Observe that the second component of the dependent sum is a singleton when $\unlock_{\mathsf{syn}} = \top$.

$$\mathsf{bool} = (\lfloor \mathsf{bool} \rfloor_{\mathsf{Syn}}, \mathbf{bool})$$

The parametricity structures for the observable values are defined as follows:

$$\#\mathsf{t}, \#\mathsf{f} : \{\mathsf{Val}(\llparenthesis \mathsf{bool} \rrparenthesis) \mid 🔓_{\mathsf{syn}} \hookrightarrow \lfloor \#\mathsf{t} \rfloor_{\mathsf{Syn}}, \lfloor \#\mathsf{f} \rfloor_{\mathsf{Syn}}\}$$
$$\#\mathsf{t} = \big(\lfloor \#\mathsf{t} \rfloor_{\mathsf{Syn}}, \eta_{\mathsf{Dyn}}(\eta_{\mathsf{Sem}}(0))\big) \qquad \#\mathsf{f} = \big(\lfloor \#\mathsf{f} \rfloor_{\mathsf{Syn}}, \eta_{\mathsf{Dyn}}(\eta_{\mathsf{Sem}}(1))\big)$$

3.4.5 Parametricity structure of computational effects

In this section, we show how to construct a monad on parametricity structures corresponding to the lax modality of ModTT, following an internal version of the recipe of Goubault-Larrecq, Lasota, and Nowak [GLN08] for gluing together two monads along a monad morphism. Emanating from the syntax is an internal monad $\lfloor \bigcirc \rfloor_{\mathsf{Syn}} : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}} \to \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}}$ on the internal category of syntactic signatures; here we describe how to glue this monad together with a monad on the internal category of purely semantic parametricity structures. Let $\mathrm{T} : \mathscr{U}^{\beta}_{\mathsf{Sem}} \to \mathscr{U}^{\beta}_{\mathsf{Sem}}$ be such a monad; we furthermore have an internal functor $\mathrm{F} : \lfloor \mathsf{Sig} \rfloor_{\mathsf{Syn}} \to \mathscr{U}^{\beta}_{\mathsf{Sem}}$ defined by taking the purely semantic part of the collection of modules of every syntactic signature:

$$\mathrm{F}(\sigma) = \mathsf{Sem}(\lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\sigma))$$

We parameterize the constructions of this section in a monad morphism $\mathsf{run} : \lfloor \bigcirc \rfloor_{\mathsf{Syn}} \to \mathrm{T}$ over F in the sense of Street [Str72], i.e. an internal natural transformation $\mathsf{run} : \mathrm{T} \circ \mathrm{F} \to \mathrm{F} \circ \lfloor \bigcirc \rfloor_{\mathsf{Syn}}$ satisfying a number of coherence conditions. Following Goubault-Larrecq, Lasota, and Nowak [GLN08], we may glue the two monads together along this morphism to define a monad on Sig, i.e. the internal category of glued signatures and glued modules determined by the constructions in Section 3.4.1. Fixing $\sigma : \mathsf{Sig}$, we may define a type $\mathrm{T}_{\mathcal{X}}(\sigma) : \mathscr{U}_{\beta}$ as follows, writing $\pi_{\sigma} : \mathsf{Sem}(\mathsf{Val}(\sigma)) \to \mathrm{T}(\mathrm{F}(\sigma))$ for the induced projection in $\mathscr{U}^{\beta}_{\mathsf{Sem}}$:

$$\mathrm{T}_{\mathcal{X}} : \textstyle\prod_{\sigma:\mathsf{Sig}} \{\mathscr{U}_{\beta} \mid 🔓_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\lfloor \bigcirc \rfloor_{\mathsf{Syn}}\sigma)\}$$
$$\mathrm{T}_{\mathcal{X}}(\sigma) \cong \textstyle\sum_{x^{\circ}:\lfloor \bigcirc \rfloor_{\mathsf{Syn}}\sigma} \big\{x^{\bullet} : \mathrm{T}(\mathsf{Sem}(\mathsf{Val}(\sigma))) \bigm| \mathsf{run}_{\sigma}(\mathrm{T}(\pi_{\sigma})(x^{\bullet})) = \eta_{\mathsf{Sem}}(x^{\circ})\big\}$$

Therefore, we may define the monad on parametricity structures for signatures as follows:

$$\bigcirc : \mathsf{Sig} \to \mathsf{Sig}$$
$$\bigcirc\sigma = (\lfloor \bigcirc \rfloor_{\mathsf{Syn}}\sigma, \mathrm{T}_{\mathcal{X}}(\sigma))$$

If ModTT is suitably extended by monadic operations (such as those corresponding to exceptions, printing, a global reference cell, etc.), then the assumptions of this section are readily substantiated by the corresponding monad on purely semantic objects. Some computational effects may require the constructions of Section 5 to be relativized from **Set** to a suitable presheaf category— for instance, partiality / general recursion might be modeled by replacing **Set** with the topos of trees as in Birkedal, Møgelberg, Schwinghammer, and Stovring [Bir+11] and Paviotti [Pav16] (but we do not make any claims in this direction).

Example 3.14. Suppose that ModTT were extended with an operation $\mathsf{throw} : \bigcirc\sigma$ for each signature σ, such that $\bigcirc$ corresponds to the exception monad. We may glue this together with the internal monad $\mathrm{T}(\mathrm{X}) = \mathsf{Dyn}(1 + \mathrm{X})$ on the internal category of purely semantic parametricity structures. We must define a family of functions $\mathsf{run}_{\sigma} : \mathrm{T}(\mathrm{F}(\sigma)) \to \mathrm{F}(\lfloor \bigcirc \rfloor_{\mathsf{Syn}}\sigma)$. Because $\mathrm{F}(\lfloor \bigcirc \rfloor_{\mathsf{Syn}}\sigma)$ is purely dynamic and Dyn is a lex idempotent modality, any such function run_σ is uniquely determined by a map $1 + F(\sigma) \longrightarrow F(\sigma)$, which we may choose as follows:

$$inl(*) \mapsto \eta_{\mathsf{Sem}}(\lfloor \mathtt{throw} \rfloor_{\mathsf{Syn}}) \qquad\qquad inr(x) \mapsto \eta_{\mathsf{Sem}}(\lfloor \mathtt{ret} \rfloor_{\mathsf{Syn}}(x))$$

Then, the monad $T_{\mathcal{X}}(\sigma)$ on a parametricity structure $\sigma : \mathsf{Sig}$ associates to each syntactic computation $M : \lfloor \bigcirc \rfloor_{\mathsf{Syn}} \sigma$ either a proof that M throws the exception or a proof that M returns a computable value.

4 Case study: representation independence for queues

In this section, we consider an extension of ModTT by an inductive type of lists, as well as the `throw` effect of Example 3.14. For the purpose of readability, we adopt a high-level notation for modules and their signatures where components are identified by name rather than by position.

4.1 A simulation structure between two queues

We may define an abstract type of queues $\lfloor \mathtt{QUEUE} \rfloor_{\mathsf{Syn}}$ together with two implementations as in Harper [Har16], depicted in Figure 4. We will observe that the semantic part of `QUEUE` is the collection of proof-relevant phase separated simulation relations between two given closed syntactic queues. First, we note the meaning of `QUEUE` in the glued algebra:

$$\mathtt{QUEUE} \cong \textstyle\sum_{t:\mathsf{type}} \langle\!| t |\!\rangle \times \langle\!| \mathsf{bool} * t \rightharpoonup t |\!\rangle \times \langle\!| t \rightharpoonup \mathsf{bool} * t |\!\rangle$$

The two implementations internalize as elements $\mathtt{Q}_0 : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{L}}, \mathtt{Q}_1 : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{R}}$; these can be combined into $\mathtt{Q}_{01} : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{Syn}}$ by splitting, $\mathtt{Q}_{01} = [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \mathtt{Q}_0 \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow \mathtt{Q}_1]$. We may define a purely dynamic type that represents the invariant structure on a pair of queues using Corollary 3.11, writing $\mathsf{bits} = 2^\star$ for $\mathsf{ParamTT}$-type of finite lists of bits and $\lceil - \rceil$ for the obvious projection of a syntactic element of ModTT-type $\mathsf{list}(\mathsf{bool})$ from finite list of bits.

$$\mathsf{invariant} : \{\mathscr{U}^{\alpha}_{\mathsf{Dyn}} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Dyn}(\lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\mathtt{Q}_{01}.\mathtt{t}))\}$$
$$\mathsf{invariant} \cong \textstyle\sum_{q:\lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\langle\!| \mathtt{Q}_{01}.\mathtt{t} |\!\rangle)} \mathsf{Sem}(\{\vec{x}, \vec{y}, \vec{z} : \mathsf{Dyn}(\mathsf{bits}) \mid \vec{x} = (\vec{y} + rev(\vec{z})) \wedge q = [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \lceil \vec{x} \rceil \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]\})$$

We may then define a single parametricity structure to unite the two implementations under the invariant above, depicted in Figure 5; it is now possible to prove the central result of our case study, the representation independence theorem for queues.

Theorem 4.1. Let $f : \lfloor \mathtt{QUEUE} \rightarrow \langle\!| \mathsf{bool} |\!\rangle \rfloor_{\mathsf{Syn}}$; then we have $f(\mathtt{Q}_0) = f(\mathtt{Q}_1)$.

Proof. This can be seen by considering the image of f under the parametricity interpretation of ModTT into $\mathsf{ParamTT}$, $\tilde{f} : \mathtt{QUEUE} \rightarrow \langle\!| \mathsf{bool} |\!\rangle$. Applying $\tilde{f}$ to the simulation queue defined in Figure 5, we have a single element of $\langle\!| \mathsf{bool} |\!\rangle$ relating two syntactic booleans:

$$b : \{\langle\!| \mathsf{bool} |\!\rangle \mid \text{🔓}_{\mathsf{syn/l}} \hookrightarrow \lfloor f(\mathtt{Q}_0) \rfloor_{\mathsf{L}} \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow \lfloor f(\mathtt{Q}_1) \rfloor_{\mathsf{R}}\}$$

But we have defined bool along the diagonal (Section 3.4.4), so this actually proves that either $f(\mathtt{Q}_0) = f(\mathtt{Q}_1) = \#\mathsf{t}$ or $f(\mathtt{Q}_0) = f(\mathtt{Q}_1) = \#\mathsf{f}$. □

```
signature QUEUE = sig
  type t
  val emp : t
  val ins : bool * t ⇀ t
  val rem : t ⇀ bool * t
end
```

```
structure Q0 : QUEUE = struct
  type t = bool list
  val emp = nil
  fun ins (x, q) = ret (x :: q)
  fun rem q =
    bind val rev_q ← rev q in
    case rev_q of
    | nil ⇒ throw
    | x :: xs ⇒
      bind val rev_xs ← rev xs in
      ret (f, rev_xs)
end
```

```
structure Q1 : QUEUE = struct
  type t = bool list * bool list
  val emp = (nil, nil)
  fun ins (x, (fs, rs)) = ret (fs, x :: rs)
  fun rem (fs, rs) =
    case fs of
    | nil ⇒
      bind val rev_rs ← rev rs in
      (case rev_rs of
       | nil ⇒ throw
       | x::rs' ⇒ ret (x, rs', nil))
    | x::fs' ⇒ ret (x, fs', rs)
end
```

Figure 4: Two implementations of a queue in an extended version of ModTT, written in SML-style notation.

A simulation over $\mathsf{Q}_{01} = [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \mathsf{Q}_0 \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow \mathsf{Q}_1]$ consists of the following data:

$$\begin{aligned} t &: \{\mathsf{Val}(\mathsf{type}) \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Q}_{01}.\mathtt{t}\} \\ emp &: \{\mathsf{Val}(\llparenthesis t \rrparenthesis) \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Q}_{01}.\mathsf{emp}\} \\ ins &: \{\mathsf{Val}(\llparenthesis \mathsf{bool} * t \rightharpoonup t \rrparenthesis) \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Q}_{01}.\mathtt{ins}\} \\ rem &: \{\mathsf{Val}(\llparenthesis t \rightharpoonup \mathsf{bool} * t \rrparenthesis) \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Q}_{01}.\mathtt{rem}\} \end{aligned}$$

These operations are implemented in ParamTT as follows.

$$\begin{aligned} t &= (\mathsf{Q}_{01}.\mathsf{t}, \mathsf{invariant}) \\ emp &= (\mathsf{Q}_{01}.\mathsf{emp}, (\langle\rangle, \langle\rangle, \langle\rangle)) \\ ins((b, x), (q, (\vec{x}, \vec{y}, \vec{z}))) &= (\mathsf{Q}_{01}.\mathtt{ins}(b, q), \eta_{\mathrm{T}}([\text{🔓}_{\mathsf{syn/l}} \hookrightarrow b :: q \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lfloor \mathtt{fst} \rfloor_{\mathsf{R}}(q), b :: \lfloor \mathsf{snd} \rfloor_{\mathsf{R}}(q))], (x :: \vec{x}, x :: \vec{y}, \vec{z}))) \\ rem(q, (\vec{x}, \vec{y}, \vec{z})).1 &= \mathsf{Q}_{01}.\mathtt{rem}(q) \\ rem(q, (\langle\rangle, \langle\rangle, \langle\rangle)).2 &= throw_{\mathrm{T}} \\ rem(q, (\vec{x} \ldots x), x :: \vec{y}, \vec{z}).2 &= \eta_{\mathrm{T}}((\lceil x \rceil, [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \lceil \vec{x} \rceil \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]), (x, (\vec{x}, \vec{y}, \vec{z}))) \\ rem(q, ((\vec{x} \ldots x), \langle\rangle, \vec{z} \ldots x)).2 &= \eta_{\mathrm{T}}((\lceil x \rceil, [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \lceil \vec{x} \rceil \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lceil rev(\vec{z}) \rceil, \lceil \vec{y} \rceil)]), (x, (\vec{x}, rev(\vec{z}), \vec{y}))) \end{aligned}$$

where

$$\mathsf{invariant} : \{\mathscr{U}^{\alpha}_{\mathsf{Dyn}} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathsf{Dyn}(\lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\mathsf{Q}_{01}.\mathsf{t}))\}$$

$$\mathsf{invariant} \cong \textstyle\sum_{q : \lfloor \mathsf{Val} \rfloor_{\mathsf{Syn}}(\llparenthesis \mathsf{Q}_{01}.\mathsf{t} \rrparenthesis)} \mathsf{Sem}(\{\vec{x}, \vec{y}, \vec{z} : \mathsf{Dyn}(\mathsf{bits}) \mid \vec{x} = (\vec{y} + rev(\vec{z})) \wedge q = [\text{🔓}_{\mathsf{syn/l}} \hookrightarrow \lceil \vec{x} \rceil \mid \text{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]\})$$

Figure 5: Constructing a simulation between the two queue implementations becomes a straightforward *programming problem* in ParamTT.

5 The topos of phase separated parametricity structures

The simplest way to substantiate the type theory ParamTT of Section 3 is to use the existing infrastructure of Grothendieck topoi and Artin gluing [AGV72]; every topos possesses an extremely rich *internal type theory*, so our strategy will be roughly as follows:

1) Embed the syntax of ModTT into a topos $\widehat{\mathscr{C}_{\mathbb{T}}}$; this will be the topos corresponding to the free cocompletion of the syntactic category $\mathscr{C}_{\mathbb{T}}$ (see Notation 2.5). The copower $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ will then serve as a suitable index for binary parametricity.
2) Identify a topos $\mathbb{S}$ that captures the notion of phase distinction: a type in the internal language of $\mathbb{S}$ should be a set that has both a static part and a dynamic part depending on it.
3) Glue the topos of (doubled) syntax $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ and the topos of semantics $\mathbb{S}$ together to form a topos $\mathcal{X}$ of *phase separated parametricity structures*: a type in the internal language of $\mathcal{X}$ will have several aspects corresponding to the orthogonal distinctions ((left syntax, right syntax), semantics) and (static, dynamic). The topos $\mathcal{X}$ then has enough structure to model all of ParamTT.

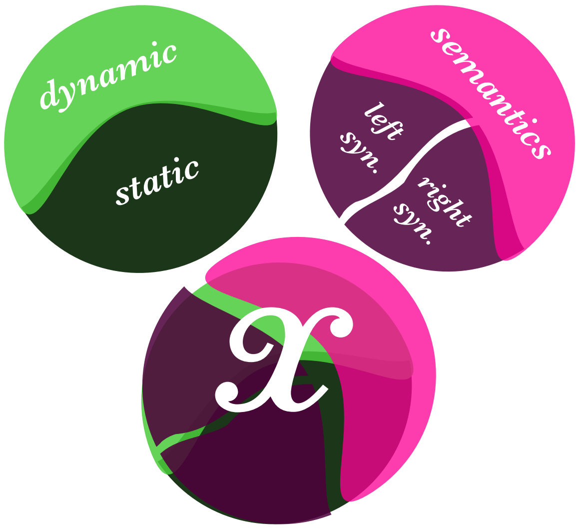


Figure 6: A geometrical depiction of the topos $\mathcal{X}$ of parametricity structures: dark and light regions in the same color-range indicate complementary open and closed subtopoi corresponding to the static–dynamic and syntactic–semantic distinctions.

5.1 Topo-logical metatheory of programming languages

To prove a property of a logical system, it has been common practice since the famous work of McKinsey and Tarski [MT46] and Kripke [Kri65] to interpret the logic into the preorder $\mathcal{O}_{\mathrm{X}}$ of opens of a carefully chosen topological space X. In this way, one may study a given axiom by finding a space whose logic of opens either verifies or refutes it. One quickly runs up against the limitations of this "topo-logical" approach, however: it is not appropriate to interpret the terms $\Gamma \vdash a : A$ of a programming language as morphisms in a preorder, because there exist non-equal $a, b : A$!

From opens to sheaves The problem identified above can be partly resolved by generalizing the concept of an *open* of a topological space to a *sheaf* on a topological space. While an open $U : \mathcal{O}_X$ can be thought of as a continuous mapping from X to the space of truth values, a sheaf $E : \mathrm{Sh}(X)$ can be thought of as a continuous mapping from X to the space of all sets. From this characterization, it is clear that sheaves generalize opens, and one might hope this would make enough room for the investigation of most type theoretic problems.

A category of points? Although the generalization to sheaves solves many problems for the study of logic qua type theory, it is not enough. In programming languages one considers semantics in functor categories $\mathscr{E} = [\mathscr{C}, \mathbf{Set}]$ as in the work of Reynolds [Rey95] and Oles [Ole86], but $\mathscr{E}$ is not likely to be of the form Sh(X) for a topological space X unless $\mathscr{C}$ is a preorder. The geometric way to view this problem is as follows: if $[\mathscr{C}, \mathbf{Set}]$ were the category of sheaves on a topological space, the collection of points of this space would have to form a category and not a preorder.

The preponderance of useful categories that behave *as if they were* the category of sheaves on a space led algebraic geometers under the leadership of Grothendieck in the early 1970s to consider a new kind of generalized space called a **topos** defined in terms of such categories, in which the refinement relation between two points might be witnessed by non-trivial evidence rather than being at most true [AGV72]. The importance of this "proof-relevance" in geometry is as follows: while there cannot be a topological space whose collection of points is the category of local algebras for a given ring, there is a category that behaves *as if it were* the category of sheaves on such a space, if it could exist.

Logoi and topoi What does it mean to behave like a category of sheaves on a space? The behavioral properties of such a category, called a *logos* by Anel and Joyal [AJ19], were concentrated by Giraud into a several simple axioms.

Definition 5.1 (Logos)**.** A *logos*, or category of sheaves, is a category closed under finite limits and small colimits, such that colimits commute with finite limits, sums are disjoint, and quotients are effective;[11] for technical reasons one also requires that a logos be presentable by generators and relations. A morphism between logoi is just a functor that preserves this structure, i.e. finite limits and small colimits.

Grothendieck's important idea was to take the (very large) category of logoi and then define a new kind of space in terms of these, which he called the *topos*.

Definition 5.2 (Topos)**.** A topos $\mathcal{X}$ is defined by specifying a logos conventionally called $\mathrm{Sh}(\mathcal{X})$, the category of "sheaves on $\mathcal{X}$"; a continuous map of topoi $f : \mathcal{X} \longrightarrow \mathcal{Y}$ is defined by specifying a morphism of logoi $f^* : \mathrm{Sh}(\mathcal{Y}) \longrightarrow \mathrm{Sh}(\mathcal{X})$ called the *inverse image* of f, i.e. a functor that is left exact (preserves finite limits) and cocontinuous (preserves colimits). In this way, by definition, one has a contravariant equivalence $\mathrm{Sh}(-) : \mathbf{Topos}^{\mathrm{op}} \longrightarrow \mathbf{Logos}$.

Remark 5.3. The left exactness and cocontinuity of morphisms of logoi generalizes the way that the inverse image of a continuous map between topological spaces preserves all joins and finite meets, as a morphism between frames of open sets.

[11] The condition that colimits commute with finite limits is analogous to the way that finite meets distribute over joins in $\mathcal{O}_X$ for a topological space X.

The style of Definition 5.2 is analogous to how a topological space is defined by specifying what its open sets are! In the case of topoi sheaves play the role that opens play in topological spaces. A topological space X gives rise to a topos $\bar{X}$, setting $\mathrm{Sh}(\bar{X})$ to be the classic category of sheaves on X; but the language of topoi is more practical than the language of topological spaces, because it contains more of the objects that we need in order to solve type theoretic and logical problems.

Example 5.4. The domain interpretation of programming languages can be seen to be an instance of this generalized "topo-logical" approach: while we are not aware of any topological space whose category of sheaves embeds the ω-CPOs, it is possible to find a topos with this property, making the Scott semantics of programming languages a special case of sheaf semantics [FR97].

5.2 The language of topoi

Definition 5.5 (Points of a topos)**.** The logos of sets **Set** is, classically, the category of sheaves on the one-point space. Therefore, we define the topos $(\mathrm{pt}) : \mathbf{Topos}$ to be the unique topos such that $\mathrm{Sh}\big((\mathrm{pt})\big) = \mathbf{Set}$. A morphism of topoi $(\mathrm{pt}) \longrightarrow \mathcal{X}$ is called a *point* of $\mathcal{X}$; from the perspective of sheaves, a point is therefore a left exact and cocontinuous functor $\mathrm{Sh}(\mathcal{X}) \longrightarrow \mathbf{Set}$; an arbitrary morphism $\mathcal{Y} \longrightarrow \mathcal{X}$ can be called a *generalized point* of $\mathcal{X}$, thinking of $\mathcal{Y}$ as the stage of definition.

In addition to points, the language of opens generalizes from topological spaces to topoi.

Definition 5.6 (Opens of a topos)**.** An *open* of a topos $\mathcal{X}$ is defined to be a subterminal object in $\mathrm{Sh}(\mathcal{X})$, i.e. a proof-irrelevant proposition in the internal type theory of $\mathcal{X}$. We will write $\mathcal{O}_{\mathcal{X}}$ for the frame of opens of the topos $\mathcal{X}$. An open $\mathrm{U} : \mathcal{O}_{\mathcal{X}}$ gives rise to a subtopos $\mathcal{X}_{\mathrm{U}} \hookrightarrow \mathcal{X}$: we define $\mathcal{X}_{\mathrm{U}}$ to be the topos corresponding to the slice logos $\mathrm{Sh}(\mathcal{X})_{/\mathrm{U}}$.

The definition of an open is given in algebraic/logical terms; however, we may also speak of them using purely geometrical language by finding a *classifying topos* of opens, i.e. a topos whose generalized points at stage $\mathcal{X}$ are all the opens of $\mathcal{X}$.

Example 5.7 (Sierpiński topos, the classifier of opens)**.** There is a topos $\mathbb{S}$ equipped with two points $\circ, \bullet : (\mathrm{pt}) \longrightarrow \mathbb{S}$ with the following property: every open subtopos $\mathcal{X}_{\mathrm{U}} \hookrightarrow \mathcal{X}$ arises in a unique way by pullback along the "open point" $\circ : (\mathrm{pt}) \hookrightarrow \mathbb{S}$:

$$\begin{array}{ccc} \mathcal{X}_{\mathrm{U}} & \longrightarrow & (\mathrm{pt}) \\ \big\downarrow & \lrcorner & \big\downarrow \circ \\ \mathcal{X} & \xrightarrow[\lfloor \mathrm{U} \rfloor]{} & \mathbb{S} \end{array}$$

The intuition is that the characteristic map $\lfloor \mathrm{U} \rfloor : \mathcal{X} \longrightarrow \mathbb{S}$ sends a point $x \in \mathcal{X}$ to the open point $\circ \in \mathbb{S}$ if $x \in \mathcal{X}_{\mathrm{U}}$, and sends it to the closed point $\bullet$ if $x \notin \mathcal{X}_{\mathrm{U}}$.

The view of opens via their characteristic maps will become important for us in Section 5.3.1, where we shall use it to obtain a phase separated version of the global sections functor.

Example 5.8 (Presheaves). Let $\mathscr{C}$ be a small category; then $\mathrm{Pr}(\mathscr{C})$ is the category of *presheaves* on $\mathscr{C}$, i.e. functors $\mathscr{C}^{\mathrm{op}} \longrightarrow \mathbf{Set}$. We write $\widehat{\mathscr{C}}$ for the topos whose sheaves are the presheaves on $\mathscr{C}$, i.e. $\mathrm{Sh}(\widehat{\mathscr{C}}) = \mathrm{Pr}(\mathscr{C})$. Suppose that $\mathscr{C}$ has finite limits; then a *generalized point* $\mathcal{Y} \longrightarrow \widehat{\mathscr{C}}$ corresponds to a left exact functor $\mathscr{C} \longrightarrow \mathrm{Sh}(\mathcal{Y})$.

5.3 Phase separation and the Sierpiński topos

We intend to use the Sierpiński topos $\mathbb{S}$ to capture the notion of phase separation: in essence, a sheaf on $\mathbb{S}$ will be a kind of "phase separated set". To substantiate this intuition, we must consider an explicit construction of $\mathbb{S}$ that allows us to characterize its sheaves in terms of something familiar.

Computation 5.9. The Sierpiński topos may be constructed in terms of presheaves (Example 5.8): letting Δ^1 be the category containing two objects and an arrow between them, we define $\mathbb{S}$ to be the presheaf topos $\widehat{\Delta^1}$. It follows from unfolding definitions that $\mathrm{Sh}(\mathbb{S}) = \mathrm{Pr}(\Delta^1) = \mathbf{Set}^{\rightarrow}$, i.e. the category of families of sets.

If a sheaf on $\mathbb{S}$ is just a family of sets, then we may profitably view the downstairs part of such a family as its "static component", the upstairs part as its "dynamic component"; the projection expresses the dependency of dynamic on static. The inverse image of the open point $\circ : (\mathrm{pt}) \hookrightarrow \mathbb{S}$ is the codomain functor $\mathrm{cod} : \mathbf{Set}^{\rightarrow} \longrightarrow \mathbf{Set}$, and the inverse image of the closed point $\bullet : (\mathrm{pt}) \hookrightarrow \mathbb{S}$ is the domain functor $\mathrm{dom} : \mathbf{Set}^{\rightarrow} \longrightarrow \mathbf{Set}$.

Of course, we might equally well replace the (static, dynamic) intuition with (syntactic, semantic), reflecting the fact that splitting a logical relation into syntactic and semantic parts is *itself* a kind of phase distinction in the language of logical relations. For this reason logical relations for a calculus that admits a phase distinction can be thought of as an iteration of logical relations: the underlying calculus ModTT is already a language of (proof-relevant) synthetic logical relations over the sublanguage of purely static kinds and constructors.

5.3.1 Phase separated global sections

Let $\mathscr{C}_{\mathbb{T}}$ be the syntactic category of ModTT; we may manipulate $\mathscr{C}_{\mathbb{T}}$ in the language of topoi by enlarging it to $\widehat{\mathscr{C}_{\mathbb{T}}}$, the topos of presheaves on $\mathscr{C}_{\mathbb{T}}$ (see Example 5.8). $\widehat{\mathscr{C}_{\mathbb{T}}}$ can be thought of as a topos of generalized syntax.

By the universal property of the Sierpiński topos (Example 5.7), the open $🔓_{\mathrm{st}} : \mathcal{O}_{\widehat{\mathscr{C}_{\mathbb{T}}}}$ corresponds to a unique continuous map $\boldsymbol{\gamma} : \widehat{\mathscr{C}_{\mathbb{T}}} \longrightarrow \mathbb{S}$ of topoi; it is appropriate to think of the direct image $\boldsymbol{\gamma}_* : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}}) \longrightarrow \mathrm{Sh}(\mathbb{S})$ as a phase separated version of the global sections functor, sending each object to the canonical projection map from its collection of global elements to their static parts.

Computation 5.10. To see that we have correctly understood the action of the direct image, we first note that the inverse image $\boldsymbol{\gamma}^* : \mathrm{Sh}(\mathbb{S}) \longrightarrow \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})$ is completely determined under the Yoneda embedding $\mathsf{y} : \Delta^1 \hookrightarrow \mathrm{Sh}(\mathbb{S})$ by the diagram $\Delta^1 \longrightarrow \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})$ corresponding to the open $🔓_{\mathrm{st}} \rightarrowtail \mathbf{1}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}$. Therefore, we may compute the direct image $\boldsymbol{\gamma}^* \dashv \boldsymbol{\gamma}_* : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}}) \longrightarrow \mathrm{Sh}(\mathbb{S})$ by adjointness:

$$\boldsymbol{\gamma}_* X = \mathrm{Hom}_{\mathrm{Sh}(\mathbb{S})}(\mathsf{y}(-), \boldsymbol{\gamma}_* X) = \mathrm{Hom}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}(\boldsymbol{\gamma}^* \mathsf{y}(-), X) \tag{1}$$

From the perspective of $\mathrm{Sh}(\mathbb{S})$ as the logos of families of sets, the direct image $\boldsymbol{\gamma}_* X$ is therefore just the function $\mathrm{Hom}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}(\mathbf{1}, X) \longrightarrow \mathrm{Hom}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}(🔓_{\mathrm{st}}, X)$ that projects the static part of a closed term of sort X, considering the functorial action of the interval $\mathsf{i} : 0 \longrightarrow 1$ on Equation (1).

5.4 Topos of parametricity structures

We will construct a topos whose sheaves will model the parametricity structures of ParamTT, as proof-relevant relations between two potentially different syntactic objects. Let E be a finite cardinal and $\mathcal{Y}$ a topos. The copower $\mathrm{E} \cdot \mathcal{Y} = \coprod_{e \in \mathrm{E}} \mathcal{Y}$ is a topos, whose corresponding logos may be computed as follows: $\mathrm{Sh}(\mathrm{E} \cdot \mathcal{Y}) = \mathrm{Sh}\left(\coprod_{e \in \mathrm{E}} \mathcal{Y}\right) = \prod_{e \in \mathrm{E}} \mathrm{Sh}(\mathcal{Y}) = \mathrm{Sh}(\mathcal{Y})^{\mathrm{E}}$.

The codiagonal morphism of topoi $\nabla : \mathrm{E} \cdot \mathcal{Y} \longrightarrow \mathcal{Y}$ corresponds under inverse image to the diagonal morphism of logoi $\nabla^* : \mathrm{Sh}(\mathcal{Y}) \longrightarrow \mathrm{Sh}(\mathcal{Y})^{\mathrm{E}}$; indeed, the diagonal map is lex as it is right adjoint to the colimit functor $\mathrm{colim}_{\mathrm{Y}} : \mathrm{Sh}(\mathcal{Y})^{\mathrm{E}} \longrightarrow \mathrm{Sh}(\mathcal{Y})$, and it is cocontinuous because it is left adjoint to the limit functor, i.e. the direct image $\nabla^* \dashv \nabla_*$. Because we are considering binary parametricity, we will set $\mathrm{E} := 2$ and define a topos whose sheaves correspond to parametricity structures by gluing. We may consider the following morphism $\boldsymbol{\rho} : 2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} \longrightarrow \mathbb{S}$ of topoi:

$$2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} \xrightarrow{\;\nabla\;} \widehat{\mathscr{C}_{\mathbb{T}}} \xrightarrow{\;\boldsymbol{\gamma}\;} \mathbb{S}, \qquad \boldsymbol{\rho} = \boldsymbol{\gamma} \circ \nabla$$

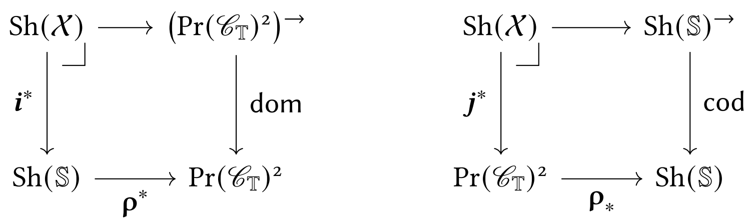
Computation 5.11. The direct image $\boldsymbol{\rho}_* : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ takes a pair $(\mathrm{X}, \mathrm{Y}) : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})^2$ of (generalized) syntactic objects to $\boldsymbol{\gamma}_*\mathrm{X} \times \boldsymbol{\gamma}_*\mathrm{Y}$, the product of their phase separated global sections.

Construction 5.12 (Topos of parametricity structures)**.** We then obtain a topos $\mathcal{X}$ whose sheaves correspond to parametricity structures by gluing, specifically via a phase separated version of the Sierpiński cone construction: we first form the Sierpiński cylinder $\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S}$ and then *pinch* the end corresponding to the closed point $\bullet \in \mathbb{S}$ along $\boldsymbol{\rho}$ as follows:

$$\begin{array}{ccc} 2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} & \xrightarrow{\;\boldsymbol{\rho}\;} & \mathbb{S} \\ {\scriptstyle (\mathbf{id}, \bullet)} \big\downarrow & & \big\downarrow {\scriptstyle \boldsymbol{i}} \\ \left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S} & \longrightarrow & \mathcal{X} \end{array}$$

Remark 5.13. The Sierpiński topos $\mathbb{S}$ plays two distinct roles in Construction 5.12: first, we use $\mathbb{S}$ to form a cylinder on $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ (which is always done in gluing), and secondly $\mathbb{S}$ is the codomain of the functor we are gluing along. This second use corresponds to the fact that we are constructing *phase separated* parametricity structures rather than ordinary parametricity structures, in which case we would be gluing into the punctual topos (pt).

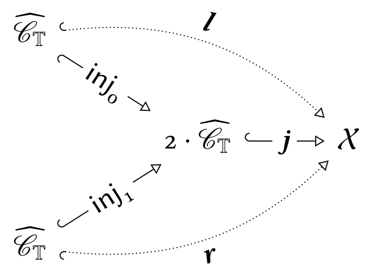
Computation 5.14. The logos $\mathrm{Sh}(\mathcal{X})$ corresponding to $\mathcal{X}$ may be computed by dualizing the diagram of Construction 5.12, noting that $\mathrm{Sh}(\mathcal{Y} \times \mathbb{S}) = \mathrm{Sh}(\mathcal{Y})^{\rightarrow}$ and recalling that under this identification, the inverse image of the (closed, open) point is the (domain, codomain) functor:



Above, $\boldsymbol{i}^*$ is the inverse image part of the closed immersion $\boldsymbol{i} : \mathbb{S} \hookrightarrow \mathcal{X}$, and $\boldsymbol{j}^*$ is the inverse image part of the open immersion $\boldsymbol{j} : 2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} \hookrightarrow \mathcal{X}$. Consequently, we arrive at a concrete description of parametricity structures (i.e. sheaves on $\mathcal{X}$):

1) A pair of generalized syntactic objects $X^{\circ}_{L}, X^{\circ}_{R} : \Pr(\mathscr{C}_{\mathbb{T}})$.
2) A family of phase separated sets $X^{\bullet} \longrightarrow \boldsymbol{\gamma}_* X^{\circ}_{L} \times \boldsymbol{\gamma}_* X^{\circ}_{R} : \mathrm{Sh}(\mathbb{S})$, i.e. a proof-relevant relation between the (phase separated) closed terms of X°_{L} and X°_{R}.

The open immersion $\boldsymbol{j} : 2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} \hookrightarrow \mathcal{X}$ corresponds (by definition) to an open $\text{🔓}_{\mathsf{syn}} : O_{\mathcal{X}}$, i.e. the subterminal parametricity structure $\text{🔓}_{\mathsf{syn}} = \left(\emptyset_{\mathrm{Sh}(\mathbb{S})} \longrightarrow \boldsymbol{\rho}_*(\mathbf{1}_{\Pr(\mathscr{C}_{\mathbb{T}})^2})\right)$. Let $\mathcal{Y}$ be a topos and E a finite cardinal; the injections $\mathsf{inj}_e : \mathcal{Y} \hookrightarrow \mathrm{E} \cdot \mathcal{Y}$ into the coproduct are in fact open immersions [Joh02, Lemma B.3.4.1]. By composition, we may therefore reconstruct $\widehat{\mathscr{C}_{\mathbb{T}}}$ as two different open subtopoi of $\mathcal{X}$:



We associate to each open subtopos of $\mathcal{X}$ a subterminal object and a corresponding open modality in Sh($\mathcal{X}$). In particular, we have opens $\text{🔓}_{\mathsf{syn}}, \text{🔓}_{\mathsf{syn/l}}, \text{🔓}_{\mathsf{syn/r}} \rightarrowtail \mathbf{1}_{\mathrm{Sh}(\mathcal{X})}$ reconstructing $\Pr(\mathscr{C}_{\mathbb{T}})^2$ as $\mathrm{Sh}(\mathcal{X})_{/\text{🔓}_{\mathsf{syn}}}$, and $\Pr(\mathscr{C}_{\mathbb{T}})$ twice as $\mathrm{Sh}(\mathcal{X})_{/\text{🔓}_{\mathsf{syn/l}}}$ and $\mathrm{Sh}(\mathcal{X})_{/\text{🔓}_{\mathsf{syn/r}}}$ respectively, corresponding to the symmetry of swapping the left and right syntactic components of a parametricity structure. Moreover, $\text{🔓}_{\mathsf{syn}} = \text{🔓}_{\mathsf{syn/l}} \vee \text{🔓}_{\mathsf{syn/r}}$ and $\text{🔓}_{\mathsf{syn/l}} \wedge \text{🔓}_{\mathsf{syn/r}} = \bot$.

Working synthetically, we may use the modalities L, R in the internal language of Sh($\mathcal{X}$) to isolate the (left, right) syntactic parts of a parametricity structure — or to *construct* parametricity structures that are degenerate everywhere except for in their (left, right) syntactic parts. The modality Syn isolates the left and right parts of the syntax together, and its *closed complement* Sem is used to trivialize the syntactic parts and isolate the semantic part: in particular, we have $\mathsf{Syn}(\mathsf{Sem}(\mathrm{X})) = \mathbf{1}$. The closed complement to an open modality is not in general open, but it is always a *lex idempotent modality* in the sense of Rijke, Shulman, and Spitters [RSS20].

The parametricity structure of phase separation is also expressed as an open modality. Recalling that we already have an open $\text{🔓}_{\mathsf{st}} : O_{2\cdot\widehat{\mathscr{C}_{\mathbb{T}}}}$ that isolates the static part of the syntax, we note that we also have an analogous open $\{\circ\} : O_{\mathbb{S}}$ of the Sierpiński topos corresponding to the open point $\circ \in \mathbb{S}$; by intersection, we may therefore define an open of $\mathcal{X}$ to isolate the static part of a general parametricity structure all at once: $\text{🔓}_{\mathsf{st}} := \boldsymbol{j}_* \text{🔓}_{\mathsf{st}} \wedge \boldsymbol{i}_* \{\circ\}$.

Lemma 5.15. The logos of parametricity structures $\mathrm{Sh}(\mathcal{X})$ is a category of presheaves, i.e. there exists a category $\mathscr{D}$ such that $\mathrm{Sh}(\mathcal{X}) \simeq \mathrm{Pr}(\mathscr{D})$.

Proof. First, we note that $\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})^2$ is $\mathrm{Pr}(2\cdot\mathscr{C}_{\mathbb{T}})$ and $\mathrm{Sh}(\mathbb{S})$ is $\mathrm{Pr}(\Delta^1)$. Moreover, the direct image $\rho_* : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ is continuous, being a right adjoint; but this is one of the equivalent conditions for the stability of presheaf topoi under gluing identified by the Grothendieck school in SGA 4, Tome 1, Exposé iv, Exercise 9.5.10 (and worked out by Carboni and Johnstone [CJ95]). □

Consequently, we may construct $\mathrm{Sh}(\mathcal{X})$ such that its internal dependent type theory contains a *strict* hierarchy of universes $\mathscr{U}_\alpha$ à la Hofmann and Streicher [HS97] and moreover enjoys the *strictification* axiom of Orton and Pitts [OP16], restated here as Axiom 3.10. This is of course only possible because the high-altitude structure of our work respects the principle of equivalence.

The central theorem of this section is an immediate consequence of the forgoing discussion, combined with standard results in the presheaf semantics of dependent type theory [HS97; Hof97; Str05].

Theorem 5.16. The category of sheaves $\mathrm{Sh}(\mathcal{X})$ admits the structure of a model of ParamTT.

Combined with the internal constructions in Section 3, we may simply unfold definitions until we reach a proof-relevant and phase separated version of Reynolds' abstraction theorem [Rey83] in the context of ModTT.

Corollary 5.17 (Generalized abstraction theorem)**.** Fix two families of signatures σ, τ : $\mathsf{Val}(\mathsf{type}) \to \mathsf{Sig}$, and a closed module functor $\mathrm{V} : \mathsf{Val}\big(\prod_{x:\mathsf{type}}\prod_{\_:\sigma(x)}\tau(x)\big)$, together with a pair of closed modules values $\mathrm{U}_i : \mathsf{Val}(\sigma(\mathrm{T}_i))$ for a pair of closed types $\mathrm{T}_0, \mathrm{T}_1 : \mathsf{Val}(\mathsf{type})$. Now, fix a family of α-small sets $\tilde{\mathrm{T}}$ indexed in the closed values of type $\mathrm{T}_0 \times \mathrm{T}_1$; the interpretations of σ, τ induce a pair of families of phase separated sets $[\![\sigma]\!](\tilde{\mathrm{T}}), [\![\tau]\!](\tilde{\mathrm{T}})$ indexed in the closed values of $\sigma(\mathrm{T}_0) \times \sigma(t_1)$ and $\tau(\mathrm{T}_0) \times \tau(\mathrm{T}_1)$ respectively. The **generalized abstraction theorem** states that we have a function of phase separated sets from $[\![\sigma]\!](\tilde{\mathrm{T}})[\mathrm{U}_0, \mathrm{U}_1]$ to $[\![\tau]\!](\tilde{\mathrm{T}})[\mathrm{V}(\mathrm{T}_0, \mathrm{U}_0), \mathrm{V}(\mathrm{T}_1, \mathrm{U}_1)]$, tracked by a function between the static components.

A further consequence of our abstraction theorem is that the static behavior of a module functor on closed modules does not depend on its dynamic behavior.

6 Conclusions and future work

What is the relationship between programming languages and their module systems? Often seen as a useful feature by which to extend a programming language, we contrarily view a language of modules as the "basis theory" that any given programming language ought to extend. To put it bluntly, a programming language is a universe $\mathcal{L}$ in the module type theory, and specific aspects (such as evaluation order) are mediated by the decoding function $t : \mathcal{L} \vdash \langle\!|t|\!\rangle\ \mathit{sig}$ of the universe.

In the present version of ModTT we chose to force all "object language" types to be purely dynamic, in the sense that $\langle\!|t|\!\rangle$ always has a trivial static component. This design, inspired by the actual behavior of ML languages with weak structure sharing (SML '97, OCaml, and 1ML), is by no means forced: by allowing types to classify values with non-trivial static components, we could reconstruct the "half-spectrum" dependent types available in current versions of Haskell [Eis16]. Taking Reynolds's dictum[12] seriously, we believe that the phase distinction is the prototype for any number of *levels of abstraction*, each corresponding to a different open modality.

Our approach is firmly rooted within the tradition of logical frameworks and categorical algebra, which has enabled us to reduce the highly technical (and very syntactic) logical relations arguments of prior work on modules to some trivial type theoretic arguments that are amenable to formalization à la Orton and Pitts [OP16]. Actually formalizing the axioms of ParamTT in a proof assistant like Agda, Coq, or Lean is within reach, thanks to the work of Gilbert, Cockx, Sozeau, and Tabareau [Gil+19].

The lax modality as an account of effects is natural, but admittedly does some violence to the dependent type structure: there can be very few useful laws governing the commutation of (non-degenerate) effects and dependent types. We plan to investigate whether the ∂CBPV calculus of Pédrot and Tabareau [PT19] can provide a better way forward, replacing the standard "dependent product of a family of types" with the more refined "dependent product of a family of algebras".

Another area for future work is to instantiate ModTT with non-trivial effects, such as recursive types or higher-order store. These features, often accounted for using step-indexing, will likely require relativizing the construction of ParamTT (Section 5) from **Set** to a logos in which domain equations can be solved.

Acknowledgments

Thanks to Mathieu Anel, Carlo Angiuli, Steve Awodey, Karl Crary, Daniel Gratzer, Gordon Plotkin, and Michael Shulman for their advice and comments, and to Tristan Nguyen at AFOSR for support.

This work was supported in part by AFOSR under grants MURI FA9550-15-1-0053 and FA9550-19-1-0216. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

[ACP09] Andreas Abel, Thierry Coquand, and Miguel Pagano. "A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance". In: *Typed Lambda Calculi and Applications*. Ed. by Pierre-Louis Curien. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 5–19. ISBN: 978-3-642-02273-9 (cit. on p. 7).

[AGV72] Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. *Théorie des topos et cohomologie étale des schémas*. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, 270, 305. Berlin: Springer-Verlag, 1972 (cit. on pp. 5, 9, 11, 25, 29, 30).

[12] "Type structure is a syntactic discipline for enforcing levels of abstraction" [Rey83].

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. “Categorical reconstruction of a reduction free normalization proof”. In: *Category Theory and Computer Science*. Ed. by David Pitt, David E. Rydeheard, and Peter Johnstone. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 182–199. ISBN: 978-3-540-44661-3 (cit. on p. 10).

[AJ19] Mathieu Anel and André Joyal. *Topo-logie*. Preprint. Mar. 2019. URL: http://mathieu.anel.free.fr/mat/doc/Anel-Joyal-Topo-logie.pdf (cit. on p. 30).

[AK16] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by Evaluation for Dependent Types”. In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 6:1–6:16. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.6. URL: http://drops.dagstuhl.de/opus/volltexte/2016/5972 (cit. on p. 9).

[All87] Stuart Frazier Allen. “A Non-Type-Theoretic Definition of Martin-Löf’s Types”. In: *Proceedings of the Symposium on Logic in Computer Science (LICS ’87)*. Ithaca, NY: IEEE, June 1987, pp. 215–221 (cit. on p. 10).

[Ang19] Carlo Angiuli. “Computational Semantics of Cartesian Cubical Type Theory”. PhD thesis. Carnegie Mellon University, 2019 (cit. on p. 24).

[Asp95] David Aspinall. “Subtyping with singleton types”. In: *Computer Science Logic*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–15. ISBN: 978-3-540-49404-1.

[BHN13] Nick Benton, Martin Hofmann, and Vivek Nigam. “Proof-Relevant Logical Relations for Name Generation”. In: *Typed Lambda Calculi and Applications*. Ed. by Masahito Hasegawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 48–60. ISBN: 978-3-642-38946-7 (cit. on p. 13).

[BHN14] Nick Benton, Martin Hofmann, and Vivek Nigam. “Abstract Effects and Proof-Relevant Logical Relations”. In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 619–631. ISSN: 0362-1340. DOI: 10.1145/2578855.2535869 (cit. on p. 13).

[Bia+94] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. “Signatures for a Network Stack: A Systems Application of Standard ML”. In: *Proceedings of the ACM Conference on LISP and Functional Programming*. Orlando, Florida: ACM Press, June 1994, pp. 55–64 (cit. on p. 6).

[Bir+11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Stovring. “First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees”. In: *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 55–64. ISBN: 978-0-7695-4412-0 (cit. on p. 26).

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. “Proofs for Free: Parametricity for Dependent Types”. In: *J. Funct. Program.* 22.2 (Mar. 2012), pp. 107–152. ISSN: 0956-7968. DOI: 10.1017/S0956796812000056 (cit. on p. 14).

[BM12] Jean-Philippe Bernardy and Guilhem Moulin. "A Computational Interpretation of Parametricity". In: *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. New Orleans, Louisiana: IEEE Computer Society, 2012, pp. 135–144. ISBN: 978-0-7695-4769-5. DOI: 10.1109/LICS.2012.25 (cit. on p. 13).

[BMW12] Clemens Berger, Paul-André Melliès, and Mark Weber. "Monads with arities and their associated theories". In: *Journal of Pure and Applied Algebra* 216.8 (2012). Special Issue devoted to the International Conference in Category Theory 'CT2010', pp. 2029–2048. ISSN: 0022-4049. DOI: 10.1016/j.jpaa.2012.02.039.

[Car78] John Cartmell. "Generalised Algebraic Theories and Contextual Categories". PhD thesis. Oxford University, Jan. 1978 (cit. on p. 9).

[CCD17] Simon Castellan, Pierre Clairambault, and Peter Dybjer. "Undecidability of Equality in the Free Locally Cartesian Closed Category (Extended version)". In: *Logical Methods in Computer Science* 13.4 (2017) (cit. on p. 7).

[CH07] Karl Crary and Robert Harper. "Syntactic Logical Relations for Polymorphic and Recursive Types". In: *Electronic Notes in Theoretical Computer Science* 172 (2007). Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin, pp. 259–299. ISSN: 1571-0661. DOI: 0.1016/j.entcs.2007.02.010.

[CH09] Karl Crary and Robert Harper. *Mechanized Definition of Standard ML (alpha release)*. 2009. URL: https://www.cs.cmu.edu/~crary/papers/2009/mldef-alpha.tar.gz (cit. on p. 8).

[CH20] Evan Cavallo and Robert Harper. "Internal Parametricity for Cubical Type Theory". In: *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*. Ed. by Maribel Fernández and Anca Muscholl. Vol. 152. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 13:1–13:17. ISBN: 978-3-95977-132-0. DOI: 10.4230/LIPIcs.CSL.2020.13. URL: https://drops.dagstuhl.de/opus/volltexte/2020/11656 (cit. on p. 13).

[CHS19] Thierry Coquand, Simon Huber, and Christian Sattler. "Homotopy canonicity for cubical type theory". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-107-8 (cit. on p. 9).

[CJ95] Aurelio Carboni and Peter Johnstone. "Connected limits, familial representability and Artin glueing". In: *Mathematical Structures in Computer Science* 5.4 (1995), pp. 441–459. DOI: 10.1017/S0960129500001183 (cit. on p. 35).

[CL90] Luca Cardelli and Xavier Leroy. "Abstract types and the dot notation". In: *Proceedings IFIP TC2 working conference on programming concepts and methods*. North-Holland, 1990, pp. 479–504.

[Clo+18] Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. *Modal Dependent Type Theory and Dependent Right Adjoints*. 2018. arXiv: 1804.05236. URL: https://arxiv.org/abs/1804.05236 (cit. on p. 15).

[Coh+17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: *IfCoLog Journal of Logics and their Applications* 4.10 (Nov. 2017), pp. 3127–3169. URL: http://www.collegepublications.co.uk/journals/ifcolog/?00019 (cit. on p. 20).

[Coq19] Thierry Coquand. "Canonicity and normalization for dependent type theory". In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015. arXiv: 1810.09367 (cit. on pp. 9, 10, 24).

[Cra17] Karl Crary. "Modules, Abstraction, and Parametric Polymorphism". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 100–113. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009892 (cit. on pp. 12, 13, 24).

[Cra19] Karl Crary. "Fully Abstract Module Compilation". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290323 (cit. on pp. 7, 12).

[Cra20] Karl Crary. "A Focused Solution to the Avoidance Problem". In: *Journal of Functional Programming* (2020). Special Issue: Essays in Honor of Robert Harper, to appear (cit. on pp. 4, 12, 15).

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. "A Type System for Higher-Order Modules". In: *SIGPLAN Not.* 38.1 (Jan. 2003), pp. 236–249. ISSN: 0362-1340. DOI: 10.1145/640128.604151 (cit. on pp. 3, 5–7, 13).

[DHC05] Derek Dreyer, Robert Harper, and Karl Crary. "Understanding and Evolving the ML Module System". PhD thesis. USA, 2005. ISBN: 0-542-01550-1.

[Dre+07] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. "Modular Type Classes". In: *SIGPLAN Not.* 42.1 (Jan. 2007), pp. 63–70. ISSN: 0362-1340. DOI: 10.1145/1190215.1190229.

[Dre07] Derek Dreyer. "A Type System for Recursive Modules". In: *SIGPLAN Not.* 42.9 (Oct. 2007), pp. 289–302. ISSN: 0362-1340. DOI: 10.1145/1291220.1291196.

[Eis16] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". PhD thesis. University of Pennsylvania, 2016 (cit. on p. 36).

[Fio02] Marcelo Fiore. "Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus". In: *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 26–37. ISBN: 1-58113-528-9. DOI: 10.1145/571157.571161. URL: http://doi.acm.org/10.1145/571157.571161 (cit. on p. 10).

[FM97] Matt Fairtlough and Michael Mendler. "Propositional Lax Logic". In: *Information and Computation* 137.1 (1997), pp. 1–33. ISSN: 0890-5401. DOI: 10.1006/inco.1997.2627 (cit. on p. 4).

[FR97] Marcelo P. Fiore and Giuseppe Rosolini. "Two models of synthetic domain theory". In: *Journal of Pure and Applied Algebra* 116.1 (1997), pp. 151–162. ISSN: 0022-4049. DOI: 10.1016/S0022-4049(96)00164-8 (cit. on p. 31).

[Gil+19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. "Definitional Proof-Irrelevance without K". In: *Proceedings of the ACM on Programming Languages*. POPL'19 (Jan. 2019), pp. 1–28. DOI: 10.1145/329031610.1145/3290316. URL: https://hal.inria.fr/hal-01859964 (cit. on p. 36).

[GLN08] Jean Goubault-Larrecq, Sławomir Lasota, and David Nowak. "Logical relations for monadic types". In: *Mathematical Structures in Computer Science* 18.6 (2008), pp. 1169–1217. DOI: 10.1017/S0960129508007172 (cit. on p. 26).

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. "Implementing a Modal Dependent Type Theory". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (July 2019), 107:1–107:29. ISSN: 2475-1421. DOI: 10.1145/3341711. URL: http://doi.acm.org/10.1145/3341711 (cit. on p. 24).

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second. New York, NY, USA: Cambridge University Press, 2016 (cit. on pp. 3, 27).

[Har19] Robert Harper. ***PFPL** Supplement: How to (Re)Invent Tait's Method*. Nov. 29, 2019. URL: http://www.cs.cmu.edu/~rwh/pfpl/supplements/tait.pdf (cit. on p. 9).

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A Framework for Defining Logics". In: *J. ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060. URL: http://doi.acm.org/10.1145/138027.138060 (cit. on p. 8).

[HL94] Robert Harper and Mark Lillibridge. "A Type-Theoretic Approach to Higher-Order Modules with Sharing". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 123–137. ISBN: 0-89791-636-0. DOI: 10.1145/174675.176927.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. "Higher-Order Modules and the Phase Distinction". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, 1990, pp. 341–354. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96744 (cit. on pp. 4, 5).

[Hof97] Martin Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation*. Cambridge University Press, 1997, pp. 79–130 (cit. on p. 35).

[HS00] Robert Harper and Christopher Stone. "A Type-theoretic Interpretation of Standard ML". In: *Proof, Language, and Interaction*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 341–387. ISBN: 0-262-16188-5. URL: http://dl.acm.org/citation.cfm?id=345868.345906 (cit. on p. 7).

[HS97] Martin Hofmann and Thomas Streicher. "Lifting Grothendieck Universes". Unpublished note. 1997. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf (cit. on p. 35).

[Joh02] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2*. Oxford Logical Guides 43. Oxford Science Publications, 2002 (cit. on pp. 11, 34).

[KHS19] Ambrus Kaposi, Simon Huber, and Christian Sattler. “Gluing for type theory”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-107-8 (cit. on pp. 9, 24, 25).

[Kri65] Saul A. Kripke. “Semantical Analysis of Intuitionistic Logic I”. In: *Formal Systems and Recursive Functions*. Ed. by J. N. Crossley and M. A. E. Dummett. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 92–130. DOI: 10.1016/S0049-237X(08)71685-9 (cit. on p. 29).

[Law63] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. PhD thesis. Columbia University, 1963 (cit. on p. 19).

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. “Towards a Mechanized Metatheory of Standard ML”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Nice, France: ACM, 2007, pp. 173–184. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190245. URL: http://doi.acm.org/10.1145/1190216.1190245 (cit. on pp. 8, 17).

[Ler+20] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system manual*. 2020 (cit. on pp. 13, 16).

[Ler94] Xavier Leroy. “Manifest Types, Modules, and Separate Compilation”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 109–122. ISBN: 0-89791-636-0. DOI: 10.1145/174675.176926.

[Ler95] Xavier Leroy. “Applicative Functors and Fully Transparent Higher-Order Modules”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 142–153. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199476 (cit. on p. 13).

[Ler96] Xavier Leroy. “A syntactic theory of type generativity and sharing”. In: *Journal of Functional Programming* 6.5 (1996), pp. 667–698. DOI: 10.1017/S0956796800001933.

[Lev04] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Norwell, MA, USA: Kluwer Academic Publishers, 2004. ISBN: 1-4020-1730-8 (cit. on p. 13).

[Mac86] David B. MacQueen. “Using Dependent Types to Express Modular Structure”. In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 277–286. ISBN: 978-1-4503-7347-0. DOI: 10.1145/512644.512670 (cit. on p. 1).

[Mar13] Per Martin-Löf. *Invariance Under Isomorphism and Definability*. Presented in the *Ernest Nagel Lectures in Philosophy & Science* at Carnegie Mellon University on March 18, March 20, and March 22 of 2013. 2013 (cit. on p. 14).

[MH88] J. C. Mitchell and R. Harper. “The Essence of ML”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 28–46. ISBN: 0-89791-252-7 (cit. on p. 7).

[MHR20] David MacQueen, Robert Harper, and John Reppy. "The History of Standard ML". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386336 (cit. on p. 8).

[Mil+97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997 (cit. on pp. 8, 16).

[MM92] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory.* Universitext. New York: Springer, 1992. ISBN: 0-387-97710-4 (cit. on p. 23).

[Mog89] Eugenio Moggi. "A Category-Theoretic Account of Program Modules". In: *Category Theory and Computer Science.* Berlin, Heidelberg: Springer-Verlag, 1989, pp. 101–117. ISBN: 3-540-51662-X (cit. on pp. 4–6).

[MS07] R. E. Møgelberg and A. Simpson. "Relational Parametricity for Computational Effects". In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007).* 2007, pp. 346–355.

[MT46] J. C. C. McKinsey and Alfred Tarski. "On Closed Elements in Closure Algebras". In: *Annals of Mathematics* 47.1 (1946), pp. 122–162. ISSN: 0003486X (cit. on p. 29).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990 (cit. on p. 8).

[NPS90] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Vol. 7. International Series of Monographs on Computer Science. NY: Oxford University Press, 1990 (cit. on p. 9).

[Ole86] F J Oles. "Type Algebras, Functor Categories and Block Structure". In: *Algebraic Methods in Semantics.* USA: Cambridge University Press, 1986, pp. 543–573 (cit. on p. 30).

[OP16] Ian Orton and Andrew M. Pitts. "Axioms for Modelling Cubical Type Theory in a Topos". In: *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France.* 2016, 24:1–24:19 (cit. on pp. 11, 20, 23, 35, 36).

[Pav16] Marco Paviotti. "Denotational semantics in Synthetic Guarded Domain Theory". English. PhD thesis. Denmark, 2016. ISBN: 978-87-7949-345-2 (cit. on p. 26).

[Péd+19] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. "A Reasonably Exceptional Type Theory". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341712 (cit. on p. 14).

[Pet05] Leaf Eames Petersen. "Certifying compilation for Standard ML in a type analysis framework". PhD thesis. Carnegie Mellon University, 2005 (cit. on p. 12).

[PP02] Gordon D. Plotkin and John Power. "Notions of Computation Determine Monads". In: *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures.* Berlin, Heidelberg: Springer-Verlag, 2002, pp. 342–356. ISBN: 3-540-43366-X (cit. on p. 19).

[PT19] Pierre-Marie Pédrot and Nicolas Tabareau. "The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371126 (cit. on pp. 13, 36).

[Red18] The RedPRL Development Team. redtt. 2018. URL: http://www.github.com/RedPRL/redtt (cit. on p. 8).

[Red20] The RedPRL Development Team. cooltt. 2020. URL: http://www.github.com/RedPRL/cooltt (cit. on pp. 7, 8, 17).

[Rey83] John C. Reynolds. "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing*. 1983 (cit. on pp. 2, 19, 35, 36).

[Rey95] John C. Reynolds. "Using Functor Categories to Generate Intermediate Code". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 25–36. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199452. URL: 10.1145/199448.199452 (cit. on p. 30).

[Ros18] Andreas Rossberg. "1ML – Core and modules united". In: *Journal of Functional Programming* 28 (2018), e22. DOI: 10.1017/S0956796818000205 (cit. on p. 12).

[RRD14] Andreas Rossberg, Claudio Russo, and Derek Dreyer. "F-ing modules". In: *Journal of Functional Programming* 24.5 (2014), pp. 529–607. DOI: 10.1017/S0956796814000264 (cit. on p. 12).

[RS17] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1 (2017) (cit. on pp. 8, 21).

[RSS20] Egbert Rijke, Michael Shulman, and Bas Spitters. "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). DOI: 10.23638/LMCS-16(1:2)2020. arXiv: 1706.07526. URL: https://lmcs.episciences.org/6015 (cit. on pp. 13, 22, 23, 34).

[SA20] Jonathan Sterling and Carlo Angiuli. "Gluing models of type theory along flat functors". Unpublished draft. 2020 (cit. on pp. 9, 24, 25).

[SAG19] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "Cubical Syntax for Reflection-Free Extensional Equality". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 31:1–31:25. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.31. arXiv: 1904.08562. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10538 (cit. on pp. 9, 24).

[SAG20] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "A cubical language for Bishop sets". Under review. 2020 (cit. on p. 9).

[SH00] Christopher A. Stone and Robert Harper. "Deciding Type Equivalence in a Language with Singleton Kinds". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Boston, MA, USA: Association for Computing Machinery, 2000, pp. 214–227. ISBN: 1-58113-125-9. DOI: 10.1145/325694.325724.

[SH06] Christopher A. Stone and Robert Harper. "Extensional equivalence and singleton types". In: *ACM Transactions on Computational Logic* (2006) (cit. on p. 7).

[Sha99] Zhong Shao. "Transparent Modules with Fully Syntatic Signatures". In: *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. Paris, France: Association for Computing Machinery, 1999, pp. 220–232. ISBN: 1-58113-111-9. DOI: 10.1145/317636.317801 (cit. on p. 13).

[Shu13] Michael Shulman. *Scones, Logical Relations, and Parametricity.* Blog post. 2013. URL: https://golem.ph.utexas.edu/category/2013/04/scones_logical_relations_and_p.html (cit. on p. 10).

[Shu15] Michael Shulman. "Univalence for inverse diagrams and homotopy canonicity". In: *Mathematical Structures in Computer Science* 25.5 (2015), pp. 1203–1277. DOI: 10.1017/S0960129514000565 (cit. on p. 10).

[SJ18] Kristina Sojakova and Patricia Johann. "A General Framework for Relational Parametricity". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science.* Oxford, United Kingdom: ACM, 2018, pp. 869–878. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209141 (cit. on p. 13).

[Sta13] Sam Staton. "Instances of Computational Effects: An Algebraic Perspective". In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science.* Washington, DC, USA: IEEE Computer Society, 2013, pp. 519–519. ISBN: 978-0-7695-5020-6 (cit. on pp. 8, 19).

[Str05] Thomas Streicher. "Universes in toposes". In: *From Sets and Types to Topology and Analysis: Towards practical foundations for constructive mathematics.* Ed. by Laura Crosilla and Peter Schuster. Vol. 48. Oxford Logical Guides. Oxford: Oxford University Press, 2005, pp. 78–90. ISBN: 978-0-19-856651-9. DOI: 10.1093/acprof:oso/9780198566519.001.0001 (cit. on pp. 22, 35).

[Str72] Ross Street. "The formal theory of monads". In: *Journal of Pure and Applied Algebra* 2.2 (1972), pp. 149–168. ISSN: 0022-4049. DOI: 10.1016/0022-4049(72)90019-9 (cit. on p. 26).

[Str91] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results.* Cambridge, MA, USA: Birkhauser Boston Inc., 1991. ISBN: 0-8176-3594-7 (cit. on p. 9).

[Tai67] W. W. Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. ISSN: 00224812. URL: http://www.jstor.org/stable/2271658 (cit. on p. 9).

[TTS18] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. "Equivalences for Free: Univalent Parametricity for Effective Transport". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236787 (cit. on p. 14).

[Uem19] Taichi Uemura. *A General Framework for the Semantics of Type Theory.* 2019. arXiv: 1904.04097. URL: https://arxiv.org/abs/1904.04097 (cit. on pp. 9, 14, 17, 19).

[Wad07] Philip Wadler. "The Girard–Reynolds isomorphism (second edition)". In: *Theoretical Computer Science* 375.1 (2007). Festschrift for John C. Reynolds's 70th birthday, pp. 201–226. ISSN: 0304-3975 (cit. on p. 14).

[Wel17] Felix Wellen. "Formalizing Cartan Geometry in Modal Homotopy Type theory". PhD thesis. Karlsruhe Institute of Technology, 2017 (cit. on p. 8).