Logical Relations as Types

Proof-Relevant Parametricity for Program Modules

JONATHAN STERLING and ROBERT HARPER, Carnegie Mellon University, USA

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.

CCS Concepts: • **Theory of computation** → *Denotational semantics*; *Categorical semantics*; **Type theory**; **Abstraction**; Type structures; • **Software and its engineering** → **Modules / packages**; *Polymorphism*; **Abstract data types**; *Functional languages*.

ACM Reference Format:
Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 1, 1 (May 2021), 47 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program modules are the application of dependent type theory with universes to the large-scale structuring of programs. As MacQueen [84] observed, the hierarchical structuring of programs is an instance of dependent sum; consider the example of a type together with a pretty printer:

Authors' address: Jonathan Sterling, jmsterli@cs.cmu.edu; Robert Harper, rwh@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
0004-5411/2021/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(* 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"* [111]; 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:

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

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:

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

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

```
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
```

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

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* [46, 58], 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 $\Diamond$; concurrent work of Crary supports the same conclusion [40]. ModTT distinguishes between commands $M \div \sigma$ and values $V : \sigma$, and mediates between them using the standard rules of the lax modality [49]:

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

In this style, one no longer needs the notion of projectibility: a generative functor is nothing more than a module-level function $\sigma \Rightarrow \Diamond\tau$, 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 Harper et al. [62], Moggi [95] 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 $\lambda u/x.M$, 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*

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

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 et al. [62] 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 et al. [46].

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 $\mathscr{G} \longrightarrow\!\triangleright \mathscr{B}$ that projects the static language from the full language. We obtain further leverage by additionally specifying $\mathscr{B}$ as a slice $\mathscr{G}_{/\text{🔓}_{\text{st}}}$ for a special object $\text{🔓}_{\text{st}} : \mathscr{G}$. In the phase-split setting, the object 🔓_{st} corresponds to a signature $(_ : \top.\bot)$ whose static part is terminal and whose dynamic part is initial; the intuition behind this definition is that the presence of $\bot$ at the dynamic level "zeroes out" any dynamic data to its right, whereas $\top$ at the static level has no effect.

The view of $\mathscr{B}$ as a slice of $\mathscr{G}$ is inspired by Artin gluing [10], 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 $(\Gamma, \text{🔓}_{\text{st}})$ called the "static open";[1] when 🔓_{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 commands and terms of program type are rendered purely dynamic / statically inert by means of special rules of static connectivity under the assumption of 🔓_{st}:

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

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

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 $\Gamma \vdash \sigma\ sig$ is called a kind iff the weakening of sets of equivalence classes from $\{[V] \mid \Gamma \vdash V : \sigma\}$ to $\{[V] \mid \Gamma, \text{🔓}_{\text{st}} \vdash V : \sigma\}$ is an isomorphism natural in Γ. In other words, the exponentiation by 🔓_{st} defines an *open modality* $\bigcirc_{\text{st}} = (\text{🔓}_{\text{st}} \to -)$ in the sense of topos theory.

Because the modality $\bigcirc_{\text{st}}$ is idempotent, we may define (internally!) the static part of any signature σ as $\bigcirc_{\text{st}}(\sigma)$; the modal unit $\eta_{\bigcirc_{\text{st}}} : \sigma \longrightarrow \bigcirc_{\text{st}}(\sigma)$ abstractly implements the projection of constructors from module values. Because the modality $\bigcirc_{\text{st}}$ 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 $\bigcirc_{\text{st}}(\sigma \Rightarrow \tau) \cong \bigcirc_{\text{st}}(\sigma) \Rightarrow \bigcirc_{\text{st}}(\tau)$, *etc.*

[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, an insight that informs the central tool of this paper.

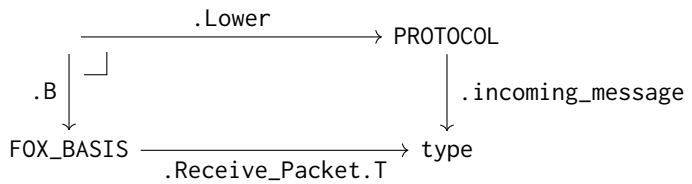
The notion of static equivalence from Dreyer et al. [46] is then reconstructed as ordinary judgmental equality in the context of $\mathbb{O}_{\mathrm{st}}$; the view of phase separation as a projection functor from Moggi [95] is reconstructed by the weakening $\mathscr{G} \longrightarrow \mathscr{G}_{/\mathbb{O}_{\mathrm{st}}}$.

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 [22] 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 [91], is perfectly appropriate from a semantic perspective; however, it unfortunately renders type checking undecidable [30]. 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 $t : \mathtt{type}$, as pioneered by Harper and Stone [63]. 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 [3, 130].

The truly difficult part of singleton types, dealt with by Stone and Harper [130], 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 [39, 46].

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 [110].[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 $\{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\}$ that classifies the "static extent" of a module $V : \sigma$ for an arbitrary signature σ, summarized in the following rules of inference:[3]

FORMATION

$$\frac{\Gamma \vdash \sigma \; \mathit{sig} \qquad \Gamma, \text{🔓}_{\text{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\} \; \mathit{sig}}$$

INTRODUCTION

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

ELIMINATION

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

In ModTT, the elements of the *static extent* of a module $V : \sigma$ are all the modules whose static part is judgmentally equal to V; therefore $\{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\}$ 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 [113], already available in a few implementations of cubical type theory [109, 110]. 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 $\text{🔓}_{\text{st}} \rightarrowtail \mathbf{1}$. The static extent connective is also closely related to the formal disk bundle of Wellen [143], 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 [89], in which each structure is in essence tagged with a static identity [83]; 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 [90]. 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

[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.

[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.

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 [123].

The mechanization of Standard ML [41, 75] in the Edinburgh Logical Framework [60] is an obvious precursor to our design; whereas in the cited work, the LF's function space was used to encode the binding structure of raw terms and derivations, we employ the internal language of locally Cartesian closed categories as a logical framework to account for both typing and judgmental equality of abstract terms, as explicated by Gratzer and Sterling [56]. The idea of dependently typed equational logical frameworks goes back to Cartmell [29] (for theories without binding), and was further developed by Martin-Löf for theories with binding of arbitrary order [97]. 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 [6, 36, 37, 71, 124, 126, 127].

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 [132] 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 [136], 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 [59]. For instance, let $R_{\mathsf{bool}} \subseteq \mathsf{ClosedTerms}(\mathsf{bool})$ be the property of being either `#t` or `#f`; one shows that R_{bool} holds of every closed boolean by lifting it to each connective in a compositional way:

$$f \in R_{\sigma \to \tau} \Leftrightarrow \forall x \in R_\sigma . f(x) \in R_\tau$$

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 [10], 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

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

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 $\mathscr{G} \longrightarrow\!\!\!\!\triangleright\ \mathscr{C}$ 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 $(\#\mathsf{t}, \#\mathsf{f})$:

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

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 Altenkirch et al. [5], Fiore [50] 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 [119, 120] and Coquand [36], 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, some 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 [4], 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". This insight is also employed in proof-relevant models of parametricity as discussed in Section 1.5.4.

Propositions:	$\blacksquare\!\!\!\!\square_{\mathsf{syn/l}}, \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}, \blacksquare\!\!\!\!\square_{\mathsf{syn}}, \blacksquare\!\!\!\!\square_{\mathsf{st}} : \Omega$	$\blacksquare\!\!\!\!\square_{\mathsf{syn}} = \blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \vee \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}$	$\blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \wedge \blacksquare\!\!\!\!\square_{\mathsf{syn/r}} = \bot$
Open modalities:	$\bigcirc_{\mathsf{syn}}, \bigcirc_{\mathsf{st}} : \mathscr{U} \to \mathscr{U}$	$\bigcirc_{\mathsf{syn}}(A) = \blacksquare\!\!\!\!\square_{\mathsf{syn}} \to A$	$\bigcirc_{\mathsf{st}}(A) = \blacksquare\!\!\!\!\square_{\mathsf{st}} \to A$
Closed modalities:	$\bullet_{\mathsf{syn}}, \bullet_{\mathsf{st}} : \mathscr{U} \to \mathscr{U}$	$\bullet_{\mathsf{syn}}(A) = A \sqcup_{A \times \blacksquare\!\!\!\!\square_{\mathsf{syn}}} \blacksquare\!\!\!\!\square_{\mathsf{syn}}$	$\bullet_{\mathsf{st}}(A) = A \sqcup_{A \times \blacksquare\!\!\!\!\square_{\mathsf{st}}} \blacksquare\!\!\!\!\square_{\mathsf{st}}$

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

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 [10, 70]: 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 [101], 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 $\blacksquare\!\!\!\!\square_{\mathsf{syn}}$ called the *syntactic open* that splits into two disjoint parts $\blacksquare\!\!\!\!\square_{\mathsf{syn}} = \blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \vee \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}$ corresponding to the left and right components of binary parametricity. Then, given a synthetic parametricity structure A, we may project the *syntactic part* of A as $\bigcirc_{\mathsf{syn}}(A) = (\blacksquare\!\!\!\!\square_{\mathsf{syn}} \to A)$. It is easy to see that $\bigcirc_{\mathsf{syn}}$ defines a 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 $A \longrightarrow \bigcirc_{\mathsf{syn}}(A)$ is an isomorphism.
(2) A proof-irrelevant proposition $\blacksquare\!\!\!\!\square_{\mathsf{st}}$ called the *static open*; then, given a synthetic parametricity structure A, the *static part* of A is projected by $\bigcirc_{\mathsf{st}}(A) = (\blacksquare\!\!\!\!\square_{\mathsf{st}} \to A)$, and a *purely static* parametricity structure A is one for which $A \longrightarrow \bigcirc_{\mathsf{st}}(A)$ 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: $\lfloor \blacksquare\!\!\!\!\square_{\mathsf{st}} \to T \rfloor \cong \blacksquare\!\!\!\!\square_{\mathsf{st}} \to \lfloor T \rfloor$.

We may then form complementary *closed modalities* $\bullet_{\mathsf{syn}}, \bullet_{\mathsf{st}}$ to the open modalities $\bigcirc_{\mathsf{syn}}, \bigcirc_{\mathsf{st}}$ that allow one to project the semantic and dynamic parts respectively of a synthetic parametricity structure, as summarized in Fig. 2. The explanation of their meaning will have to wait, but we simply note that the "semantic modality" $\bullet_{\mathsf{syn}}$ is the universal way to trivialize the syntactic part of a parametricity structure, and the "dynamic modality" $\bullet_{\mathsf{st}}$ is the universal way to trivialize the static part of a parametricity structure.

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:

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

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 $A \cong \bigcirc_{\mathsf{syn}}(A)$.

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 [116], which, as here, uses a universe to represent a signature of "small" types of 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, declarative rules for neither typing nor equality of modules are specified by Rossberg [116], Rossberg et al. [117]. Consequently, the most substantial difference between ModTT and 1ML 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 [105]. Nevertheless, it is also important to give a direct type-theoretic account of program modules *as they are to be used and reasoned about.*

It is true that our language too would require elaboration to be usable in practice, but elaboration here is needed only to introduce subtyping. The transformation of source code into core language code therefore preserves the intuitive meanings of all modular constructs. For instance, the meaning of module hierarchy in our calculus is simply dependent sum; in contrast, the meaning of a module hierarchy in the F-ing calculi can only be understood by unraveling the somewhat complex relationship between module signatures and the n-ary iterations of existential types they denote.

A very elegant contribution of Rossberg et al. [117] is a more compositional reduction of the *dot notation* `M.t` to existential unpacking than that of Cardelli and Leroy [28]. In light of the fact that an existential encoding of dependent sums cannot satisfy the η-law, however, we find that MacQueen's intervention remains in force today: abstract types do *not* have existential type (*pace* Mitchell and Plotkin [92, 93]). We welcome further exploration of the F-ing interpretation's equational theory, whose relationship to the equational theory of modules themselves remains somewhat opaque.

1.5.2 Modules, Abstraction, and Parametric Polymorphism. In a pair of recent papers [38, 39], 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 [40] 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 of kinds by Atkey [12], and is not likely to scale to more universes.

1.5.3 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 [77], later used in OCaml's module system [80]: whereas generative functors can be thought of as functions $\sigma \Rightarrow \Diamond\tau$, applicative functors correspond roughly to $\Diamond(\sigma \Rightarrow \tau)$ as noted by Shao [118], but subtleties abound. The subtleties of applicative and generative functor semantics (studied by Dreyer et al. [46] 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.

1.5.4 Proof-relevant relational interpretation. We are not the first to consider proof-relevant relational interpretations, which are essential in the context of dependent type theory because they enable a compositional interpretation of the universe, an insight employed by Atkey et al. [13], Nuyts et al. [99]. Atkey [12] uses the same insight in his interpretation of *kinds* as reflexive graphs, with the kind of types given by the reflexive graph of set-theoretic relations. Sojakova and Johann [121] define a general framework for parametric models, which can be instantiated to give rise to a proof-relevant version of parametricity. Benton et al. [15, 16] 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 Crary [38]. 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.

1.5.5 Syntactic vs. *semantic parametricity.* Parametricity has historically been studied in two forms: the present work is about *syntactic parametricity* aims to establish identifications between definable terms within a theory, whereas *semantic parametricity* aims to establish the compatibility of a theory with certain identifications by means of a model. Syntactic parametricity can be construed somewhat crudely as being about *counting* the number of definable elements of a given type, whereas semantic parametricity has no bearing at all on the question of how many elements a given type has. The relationship between the semantic and syntactic parametericity is somewhat analogous to the difference between a model of typed lambda calculus that interpets `bool` as $\mathbf{1} + \mathbf{1}$, and a logical relations or gluing argument that establishes that there are exactly two distinct closed terms of type `bool`. The former establishes the compatiblity of the language with the standard booleans, whereas the latter establishes that the language actually has the standard booleans.

A particularly attractive model of semantic parametricity is given in *reflexive graphs* [12, 13, 115]. A reflexive graph is given by an object E of edges, an object N of nodes, two morphisms $\pi_L, \pi_R : E \longrightarrow N$, and a morphism $r : N \longrightarrow E$ that is a section of both π_L, π_R. Reflexive graphs can be seen to be a proof-relevant generalization of reflexive relations, because the two boundary projections can be viewed a single map $E \longrightarrow N \times N$ which need not be a monomorphism. As a category of diagrams, reflexive graphs give rise to a presheaf topos, and hence they have the advantage of being closed under universes as discussed above (Section 1.5.4).

1.5.6 Internal parametricity. Abstracting from the semantics of parametricity, it is possible to consider extensions of dependent type theory for *internal parametricity* that involve connectives for relatedness [18, 19, 21, 31, 32, 73, 98]. Semantic parametricity, especially as embodied in reflexive graphs, can be seen to be a truncation of a much higher-dimensional structure; going one level up, one can consider a reflexive graph enriched in reflexive graphs, but there is no need to stop there. Iterating the reflexive graph construction infinitely, one gains the ability to speak non-trivially of relatedness of proofs of relatedness, and so on. Abstracting from these semantics, one obtains higher-dimensional relatedness connectives as in the work of Bernardy et al. [18], Cavallo and Harper [32].

As we noted in Section 1.5.5, semantic parametricity differs from syntactic parametricity in that it does not prove any non-trivial property of a *language*. Internal parametricity can be seen as an extreme way to resolve this deficiency, by defining a new language that is inspired by the parametric model.

1.5.7 Representation independence via univalence. The *principle of invariance* is a law of structural mathematics stating that all definable constructs ought to be invariant under isomorphism [14]. The closure of formal languages for mathematics under this principle can be seen to be somewhat analogous to parametricity arguments in which one restricts attention to relations that are the graphs of isomorphisms, as pointed out by Martin-Löf in his Ernest Nagel Lecture [85]. Voevodsky's *univalence* principle [137, 139] is the internalization of this invariance into a new formal language for mathematics, Homotopy Type Theory / Univalent Foundations [137].

Univalence states that isomorphic types are interchangeable; as a programming tool, univalence allows one to replace any goal of the form $P(A)$ with one of the form $P(B)$ provided as one has an isomorphism $A \cong B$. While univalence is stated only for isomorphisms $A \cong B$ between types, the principle also applies to structure-preserving isomorphisms between implementations of *abstract types*, *i.e.* types equipped with operations. In this way, univalence is an internal parametricity principle for abstract types *vis-à-vis* isomorphisms rather than arbitrary relations.

Angiuli et al. [9] demonstrate that the limitation of univalence to isomorphisms is not a serious one in practice as far as representation independence is concerned; on the other hand, actual parametricity is still needed to obtain *free theorems* in the sense of Wadler [140]. Most practical examples of representation independence where the relation is not an isomorphism can be "upgraded" to a structure-preserving isomorphism by quotienting the representation types on either side. The contribution of Angiuli et al. [9] is to identify sufficient conditions on a relation required for such an upgrade to take place, and to develop a library of lemmas and proof tactics that facilitate the use of univalence to prove internal representation independence results.

1.5.8 The Plotkin–Abadi parametricity logic. Similar in spirit to our efforts is the Plotkin–Abadi logic for parametricity polymorphism [107], which overlays a logic over System F that includes not only the equational theory of System F but an additional non-logical axiom scheme for parametricity. The Plotkin–Abadi logic relies on a built-in parametricity translation: for each family of System F types $\alpha \vdash \sigma[\alpha]$ and relation $R \subseteq \tau_L \times \tau_R$, there is a relation $\sigma[\![R]\!] \subseteq \sigma[\tau_L] \times \sigma[\tau_R]$ defined inductively on the structure of $\sigma[\alpha]$. Then the parametricity axiom scheme asserts for for any polymorphic program $u : \forall\alpha.\sigma[\alpha]$ and any relation $R \subseteq \tau_L \times \tau_R$, we have $(u[\tau_L], u[\tau_R]) \in \sigma[\![R]\!]$. The resulting logic can be used to refine and prove theorems about System F programs that would follow from parametricity. Birkedal and Møgelberg [24] provide a category-theoretic notion of parametricity that is sound and complete for the Plotkin–Abadi logic.

Our work can be seen as a more systematic way to recover the consequences of parametricity. Plotkin and Abadi [107] need to *axiomatize* parametricity atop a built-in parametricity translation embodied in the relational instantiation $\sigma[\![R]\!]$. In contrast, we derive parametricity results from

more general considerations; in particular, the connection between types and relations that lies at the heart of the Plotkin–Abadi axiom is reflected in our setting by the classic *fracture theorem* for recollments [10, 114]. On the other hand, we do *not* deal with impredicative polymorphism and hence our approach does not account for System F; it is plausible that this could be resolved by replaying our constructions over a realizability topos as in the work of Hyland [67], Pitts [106], but this remains to be verified and is by no means obvious.

1.5.9 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 et al. [20], Pédrot et al. [104], Tabareau et al. [135], also taken up by Per Martin-Löf in his Ernest Nagel Lecture in 2013 [85]. In the case of unary parametricity (logical predicates), one has the *deliverables* translation described by McKinna and Burstall [86] and investigated fibrationally by Hermida [64]. Closely related to the binary parametricity translations is the System R calculus of *formal parametricity* by Abadi et al. [2]. To put it somewhat crudely, System R is a language for programming in the image of a parametricity translation.

One methodological difference between these and our work 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 *modally* isolate the ones that are degenerate in either the syntactic or semantic direction.

Another significant difference between our work and the parametricity translations (as well as internal parametricity) is that we account for the quite common situation in which the semantic parts of parametricity structures come from an entirely different category than their syntactic parts; this is important, because there *is* a difference between (e.g.) parametricity with respect to definable relations, and parametricity with respect to set-theoretic relations on closed terms. In fact, we make use of this flexibility in this paper by considering parametricity with respect to *phase-separated* relations on (phase-separated) closed terms.

1.5.10 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 $\mathbf{X} = \left(\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S}\right) \sqcup_{2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}} \mathbb{S}$ 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 [141]. 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.

1.5.11 Computational effects. Lax modalities do not interact cleanly with dependent type structure, unlike the idempotent lex and open modalities of Rijke et al. [114]. A potentially promising approach to the integration of real (non-idempotent) effects into dependent type theory is represented by the ∂CBPV calculus of Pédrot and Tabareau [103], a dependently typed version of Levy's Call-By-Push-Value [81] 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 *certain* effects in dependent type theory, especially those for which the collection of algebras is itself an algebra. Because not all effects that we wish to support can have this very strong property, we based our theory on the more traditional Moggi metalanguage [96].

At this time, we are unsure how best to account for the addition of general recursion and general store to our language. For general recursion, one possibility is to model the semantic parts of parametricity structures in a topos model of synthetic domain theory [51, 68] or synthetic guarded domain theory [25]; then the computational monad might be interpreted as a kind of lifting operation. Higher-order store is a thornier question whose solution likely involves an application of synthetic guarded domain theory, but remains elusive; there is a circularity involved in the

semantics of higher-order store that seems, even in the presence of solutions to the needed domain equations, to preclude a *proof-relevant* interpretation of the parametricity structures of types.

2 ModTT: A TYPE THEORY FOR PROGRAM MODULES

We introduce ModTT, a type-theoretic core language for modules based on the considerations discussed in Section 1. We first give an informal description of the language using familiar notations in Section 2.1; in Section 2.2, we present the formal presentation of ModTT in a logical framework.

2.1 Informal presentation of ModTT

2.1.1 Judgmental structure. ModTT is arranged around three basic syntactic classes: contexts $\boxed{\Gamma\ ctx}$, signatures $\boxed{\Gamma \vdash \sigma\ sig}$, module values $\boxed{\Gamma \vdash V : \sigma}$, and module commands $\boxed{\Gamma \vdash M \div \sigma}$. 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; furthermore, module signatures, values, and commands 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 $t : \mathsf{type}$, there is a signature $\langle\!|t|\!\rangle$ classifying the values of the type t.

$$\textsc{type}\ \frac{}{\Gamma \vdash \mathsf{type}\ sig} \qquad \textsc{dynamic}\ \frac{\Gamma \vdash t : \mathsf{type}}{\Gamma \vdash \langle\!|t|\!\rangle\ sig}$$

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

$$\textsc{formation}\ \frac{\Gamma \vdash \sigma\ sig}{\Gamma \vdash \diamond\sigma\ sig} \qquad \textsc{introduction}\ \frac{\Gamma \vdash M \div \sigma}{\Gamma \vdash \{M\} : \diamond\sigma} \qquad \textsc{return}\ \frac{\Gamma \vdash V : \sigma}{\Gamma \vdash \mathsf{ret}(V) \div \sigma} \qquad \textsc{bind}\ \frac{\Gamma \vdash V : \diamond\sigma \qquad \Gamma, X : \sigma \vdash M \div \sigma'}{\Gamma \vdash (X \leftarrow V; M) \div \sigma'}$$

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:

$$\textsc{dependent sum}\ \frac{\Gamma \vdash \sigma\ sig \qquad \Gamma, X : \sigma \vdash \sigma'\ sig}{\Gamma \vdash [X : \sigma; \sigma']\ sig} \qquad \textsc{dependent product}\ \frac{\Gamma \vdash \sigma\ sig \qquad \Gamma, X : \sigma \vdash \sigma'\ sig}{\Gamma \vdash (X : \sigma) \Rightarrow \sigma'\ sig}$$

Generative functors are defined as a mode of use of the dependent product combined with the lax modality, taking $((X : \sigma) \Rightarrow^{\mathsf{gen}} \sigma') := (X : \sigma) \Rightarrow \diamond\sigma'$ as in Crary [40].

[6] A lax modality is exactly the same thing as a strong monad; at this level, the judgmental distinction between a "value of signature $\diamond\sigma$" and a "command of signature σ" is blurred, because one conventionally works up to isomorphism. It would therefore be fine to omit the form of judgment $\Gamma \vdash M \div \sigma$ from our language, but we keep it for aesthetic reasons.

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 $\Gamma, \text{🔓}_{\text{st}}$ called the *static open* context:

$$\frac{}{\cdot\ ctx} \qquad \frac{\Gamma\ ctx \qquad \Gamma \vdash \sigma\ sig}{\Gamma, X : \sigma\ ctx} \qquad \frac{\Gamma\ ctx}{\Gamma, \text{🔓}_{\text{st}}\ ctx}$$

Remark 2.1. The notation is suggestive of the accounts of modal type theory based on dependent right adjoints [33]; indeed, the context extension $(-, \text{🔓}_{\text{st}})$ can be seen as a modality on contexts left adjoint to a modality on signatures that projects out their static parts.

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 $\Gamma \vdash \text{🔓}_{\text{st}}$. Specifically, we add rules to ensure that programs of a given type as well as commands of a given signature are *statically connected* in the sense of having exactly one element under 🔓_{st}, as in Section 1.2.3.

STATIC CONNECTIVITY (1)
$$\frac{\Gamma \vdash t : \mathsf{type} \qquad \Gamma \vdash \text{🔓}_{\text{st}}}{\Gamma \vdash * : t}$$

STATIC CONNECTIVITY (2)
$$\frac{\Gamma \vdash t : \mathsf{type} \qquad \Gamma \vdash e : t \qquad \Gamma \vdash \text{🔓}_{\text{st}}}{\Gamma \vdash e \equiv * : t}$$

STATIC CONNECTIVITY (3)
$$\frac{\Gamma \vdash \sigma\ sig \qquad \Gamma \vdash \text{🔓}_{\text{st}}}{\Gamma \vdash * \div \sigma}$$

STATIC CONNECTIVITY (4)
$$\frac{\Gamma \vdash \sigma\ sig \qquad \Gamma \vdash M \div \sigma \qquad \Gamma \vdash \text{🔓}_{\text{st}}}{\Gamma \vdash M \equiv * \div \sigma}$$

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 $\Gamma, \text{🔓}_{\text{st}} \vdash V : \sigma$ as the signature $\{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\}$ 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.

EXTENT/FORMATION
$$\frac{\Gamma \vdash \sigma\ sig \qquad \Gamma, \text{🔓}_{\text{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\}\ sig}$$

EXTENT/INTRO
$$\frac{\Gamma \vdash W : \sigma \qquad \Gamma, \text{🔓}_{\text{st}} \vdash W \equiv V : \sigma}{\Gamma \vdash \mathsf{in}_V(W) : \{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow V\}}$$

EXTENT/ELIM
$$\frac{\Gamma \vdash V : \{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow W\}}{\Gamma \vdash \mathsf{out}_W(V) : \sigma}$$

EXTENT/INVERSION
$$\frac{\Gamma \vdash \text{🔓}_{\text{st}} \qquad \Gamma \vdash V : \{\sigma \mid \text{🔓}_{\text{st}} \hookrightarrow W\}}{\Gamma \vdash \mathsf{out}_W(V) \equiv W : \sigma}$$

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

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

The static extent reconstructs both type sharing and weak structure sharing, which appear in SML '97 [90] and OCaml [80].

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

Section 1.2.3:

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

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 $\langle\!|s \rightharpoonup t|\!\rangle \cong \langle\!|s|\!\rangle \Rightarrow \Diamond\langle\!|t|\!\rangle$.

2.1.8 External language and elaboration. We do not present here a surface language, which 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 et al. [75] are re-cast as an elaboration strategy guided by η-laws, as in the elaboration of extension types in the `cooltt` proof assistant [110]. 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

Rather than studying directly the informal presentation of ModTT given in Section 2.1, we intend to study a mathematical version of this syntax that can be defined in a logical framework, namely the internal language of locally Cartesian closed categories. We use the logical framework to capture not only binding structure, but also well-typedness and judgmental equality. One important difference between the informal presentation and the logical framework presentation is that the latter does not distinguish contexts from other forms of judgment; such a distinction can be important for implementation as well as establishing various metatheorems (*e.g.* normalization), but it does not seem to play a role in the specification of the theory itself.

Theories are encoded in the LF (logical framework) as follows:

(1) Both parameters and hypothetical judgments are formulated using the dependent products $(x : X) \to Y(x)$ of the LF.
(2) The LF contains one universe **Jdg** of *judgments*; an object-level judgment/sort is defined by adding a constant whose type ends in **Jdg**. LF signatures must use **Jdg** in only strictly positive positions.
(3) Object-level equality is specified by adding constants whose types end in the logical framework's equality type $(a =_X b)$.

The LF signature of ModTT is presented in Fig. 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 $\mathscr{C}$. In particular, if $\mathscr{U}$ is

$$
\begin{aligned}
\blacksquare_{\mathsf{st}} &: \mathbf{Jdg} \\
_ &: (x, y : \blacksquare_{\mathsf{st}}) \to x =_{\blacksquare_{\mathsf{st}}} y \\
\mathsf{Sig} &: \mathbf{Jdg} \\
\mathsf{Val} &: \mathsf{Sig} \to \mathbf{Jdg} \\
\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 (\blacksquare_{\mathsf{st}} \to \mathsf{Val}(\sigma)) \to \mathsf{Sig} \\
\Diamond &: \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((U : \mathsf{Val}(\sigma)) \times ((z : \blacksquare_{\mathsf{st}}) \to U =_{\mathsf{Val}(\sigma)} V(z))\big) \cong \mathsf{Val}(\mathsf{Ext}(\sigma, V)) \\
\mathsf{Cmd} &: \mathsf{Sig} \to \mathbf{Jdg} \\
\mathsf{Cmd} &:= \lambda\sigma.\mathsf{Val}(\Diamond\sigma) \\
\mathsf{conn}/\mathsf{dyn} &: \blacksquare_{\mathsf{st}} \to \mathsf{Val}(\langle\!|t|\!\rangle) \cong \mathbf{1} \\
\mathsf{conn}/\mathsf{cmp} &: \blacksquare_{\mathsf{st}} \to \mathsf{Cmd}(\sigma) \cong \mathbf{1} \\
\mathsf{ret} &: \mathsf{Val}(\sigma) \to \mathsf{Cmd}(\sigma) \\
\mathsf{bind} &: (\mathsf{Val}(\sigma) \to \mathsf{Cmd}(\tau)) \to \mathsf{Cmd}(\sigma) \to \mathsf{Cmd}(\tau) \\
_ &: \mathsf{bind}(F, \mathsf{ret}(V)) =_{\mathsf{Cmd}(\tau)} F(V) \\
_ &: \mathsf{bind}(F, \mathsf{bind}(G, V)) =_{\mathsf{Cmd}(\rho)} \mathsf{bind}(\lambda x.\mathsf{bind}(F, G(x)), 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{Cmd}(\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{Cmd}(\langle\!|t|\!\rangle) \to \mathsf{Cmd}(\langle\!|t|\!\rangle) \to \mathsf{Cmd}(\langle\!|t|\!\rangle) \\
_ &: \mathsf{if}(\#\mathsf{t}, M, N) =_{\mathsf{Cmd}(\langle\!|t|\!\rangle)} M \\
_ &: \mathsf{if}(\#\mathsf{f}, M, N) =_{\mathsf{Cmd}(\langle\!|t|\!\rangle)} N
\end{aligned}
$$

Fig. 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 Ext/val declaring an isomorphism, which can be unfolded to a dependent sum.

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 **Jdg** is 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.

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.[7] 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 our development is best served by *functorial semantics* in the spirit of Lawvere [74]. A model of Σ in a sufficiently structured category $\mathscr{E}$ can be viewed in two ways:

(1) A model of Σ is an element of $\mathbf{Alg}_\Sigma(\mathscr{U})$ for some universe $\mathscr{U}$ in a locally Cartesian closed category $\mathscr{E}$.
(2) A model of Σ is a locally Cartesian closed functor 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.

Notation 2.4. We will write $\mathbb{T}$ for the signature presenting ModTT in Fig. 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 reference cells or exceptions is clearly algebraic (see *e.g.* Plotkin and Power [108]), 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 [123] — as Staton's work shows, there is no obstacle to the equational presentation of *any* reasonable form of deterministic 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 [111] for ModTT stated in Corollary 5.32.

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 $\mathscr{U}_\alpha$ of types, inductive types, subset types, and effective quotient types (consequently, strict pushouts). We first axiomatize ParamTT in the style of Orton and Pitts [101], 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

[7] Our application does not, however, rely on this bijection: instead, we treat the morphisms of the presented category as a *definition* of our language.

a system of proof irrelevant propositions corresponding to the static part and the disjoint (left)-syntactic and (right)-syntactic parts of a parametricity structure.

$$\blacksquare\!\!\!\!\square_{\mathsf{st}}, \blacksquare\!\!\!\!\square_{\mathsf{syn/l}}, \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}, \blacksquare\!\!\!\!\square_{\mathsf{syn}} : \Omega \qquad \blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \wedge \blacksquare\!\!\!\!\square_{\mathsf{syn/r}} = \bot \qquad \blacksquare\!\!\!\!\square_{\mathsf{syn}} := \blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \vee \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}$$

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 $\bigcirc_{\mathsf{st}} := \mathsf{Open}_{\blacksquare\!\!\!\!\square_{\mathsf{st}}}$ and the "syntactic modality" to be $\bigcirc_{\mathsf{syn}} := \mathsf{Open}_{\blacksquare\!\!\!\!\square_{\mathsf{syn}}}$; 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 $\blacksquare\!\!\!\!\square_{\mathsf{syn/l}}, \blacksquare\!\!\!\!\square_{\mathsf{syn/r}}$ have no overlap, we have an isomorphism $\bigcirc_{\mathsf{syn}}(A) \cong (\blacksquare\!\!\!\!\square_{\mathsf{syn/l}} \to A) \times (\blacksquare\!\!\!\!\square_{\mathsf{syn/r}} \to A)$. This isomorphism is captured more generally by the following *systems* notation of Cohen et al. [34] from cubical type theory for constructing maps out of disjunctions of propositions:

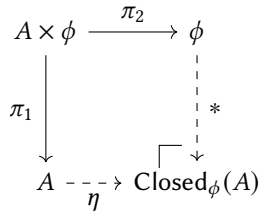
Notation 3.3 (Systems). Following Cohen et al. [34], we employ the notation of *systems* for constructing elements of parametricity structures underneath the assumption of disjunction of propositions $\phi \vee \phi'$: when $\phi \wedge \phi'$ implies $a = a' : A$, we may write $[\phi \hookrightarrow a, \phi' \hookrightarrow a']$ 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 [113]: 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 Closed_φ (A : 𝒰) where
  η : A → Closed_φ(A)
  * : φ → Closed_φ(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.

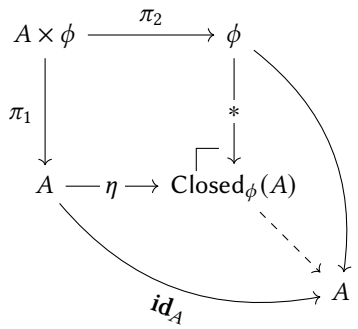
Using Construction 3.5, we define the "purely semantic" and "purely dynamic" modalities respectively:

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

Lemma 3.6. *For any $\phi : \Omega$, 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 $\bigcirc_{\mathsf{syn}}/\bullet_{\mathsf{syn}}$ and $\bigcirc_{\mathsf{st}}/\bullet_{\mathsf{st}}$ are each complementary: in particular, we have $\bigcirc_{\mathsf{syn}}(\bullet_{\mathsf{syn}}(A)) \cong \mathbf{1}$ and $\bigcirc_{\mathsf{st}}(\bullet_{\mathsf{st}}(A)) \cong \mathbf{1}$. Put more crudely, a "dynamic thing has no static component" and a "semantic thing has no syntactic component". This complementarity is not the one of boolean logic: the open/closed partition evinces an area of overlap that is sometimes called the *boundary* or *fringe*, depicted visually in Fig. 6. The geometrical boundary between complementary the open and closed subspaces is reflected in the modal presentation the fact that the semantic part of a syntactic thing is *not* trivial, *i.e.* we do not have $\bullet_{\mathsf{syn}}(\bigcirc_{\mathsf{syn}}(A)) \cong \mathbf{1}$.

3.2 Universes of modal types

Each universe $\mathcal{U}_\alpha$ 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 $\mathbf{M}$, thought to be ranging over $\{\bigcirc_{\mathsf{syn}}, \bullet_{\mathsf{syn}}, \bigcirc_{\mathsf{st}}, \bullet_{\mathsf{st}}\}$, we might naïvely consider defining the universe $\mathcal{U}^\alpha_{\mathbf{M}}$ of $\mathbf{M}$-modal types as a subtype:

$$\mathcal{U}^\alpha_{\mathbf{M}} \coloneqq \{A : \mathcal{U}_\alpha \mid A \cong \mathbf{M}(A)\} \tag{bad}$$

Unfortunately, such a universe will not itself be $\mathbf{M}$-modal, *i.e.* we do not have $\mathbf{M}(\mathcal{U}^\alpha_{\mathbf{M}}) \cong \mathcal{U}^\alpha_{\mathbf{M}}$, hence there is no hope of closing the $\mathbf{M}$-modal fragment of ParamTT under a hierarchy of universes with such a definition.[8] An idea pioneered in a different context by Streicher [133] is to apply the modality directly to the universe:

$$\mathcal{U}^\alpha_{\mathbf{M}} \coloneqq \mathbf{M}(\mathcal{U}_\alpha) \tag{good}$$

With such a definition, we immediately have $\mathbf{M}(\mathcal{U}^\alpha_{\mathbf{M}}) \cong \mathcal{U}^\alpha_{\mathbf{M}}$, *etc.*; but we still have to specify the decodings of these new universes, which is to explain what the type of elements of the modal universe is. This can be done systematically for any modality $\mathbf{M}$, so long as $\mathbf{M}$ preserves the universe level of types. Categorically, one views the universe $\mathcal{U}_\alpha$ as a *generic family* $\pi : \sum_{A:\mathcal{U}_\alpha} A \longrightarrow \mathcal{U}_\alpha$ that expresses the indexing of elements over types. The insight of Streicher [133] was to apply the modality $\mathbf{M}$ to the entire generic family yielding $\mathbf{M}(\pi) : \mathbf{M}\big(\sum_{A:\mathcal{U}_\alpha} A\big) \longrightarrow \mathcal{U}^\alpha_{\mathbf{M}}$, and then obtain the collection of elements of a given $A : \mathcal{U}^\alpha_{\mathbf{M}}$ by pullback.

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

$$\mathsf{El}_{\mathbf{M}} : \mathcal{U}^\alpha_{\mathbf{M}} \to \mathcal{U}_\alpha$$
$$\mathsf{El}_{\mathbf{M}}(A) \coloneqq \big\{x : \mathbf{M}\big(\textstyle\sum_{X:\mathcal{U}_\alpha} X\big) \;\big|\; \mathbf{M}(\pi)(x) = A\big\}$$

We note that each modal universe $\mathcal{U}^\alpha_{\mathbf{M}}$ is closed under all the connectives of ParamTT, a general fact about lex idempotent modalities in topos theory [82] and type theory [114].

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

In the case of the open modality for a proposition $\phi : \Omega$, there is a simpler computation of the decoding of the open subuniverse, namely $\mathsf{El}_{\mathsf{Open}_\phi}(A) \coloneqq \prod_{z:\phi} A(z)$.

Notation 3.8. From Lemma 3.7, we are inspired to adopt a slight abuse of notation: when $A : \mathcal{U}_\alpha$, we will often write $\mathbf{M}(A) : \mathcal{U}^\alpha_{\mathbf{M}}$ 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 Birkedal et al. [23], Orton and Pitts [101], which we will justify by a model construction in Section 5.

Axiom 3.9 (Strictification). *Let $\phi : \Omega$ be a proposition, and let $A : \phi \to \mathcal{U}_\alpha$ be a partial type defined on the extent of ϕ, and let $B : \mathcal{U}_\alpha$ 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 3.9 above plays a critical role in the constructions of Section 3.4, letting $\phi \coloneqq \text{🔒}_{\mathsf{syn}}$.

[8] The "naïve" definition considered here *does* work in homotopy type theories in the presence of the univalence principle, as shown by Rijke et al. [114]; 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.

Corollary 3.10 (Realignment). *Let $A : \mathcal{U}^{\alpha}_{\bigcirc_{\mathsf{syn}}}$ be a syntactic type, and fix $\tilde{A} : \mathcal{U}_\alpha$ whose syntactic part is isomorphic to A, i.e. we have $f : \bigcirc_{\mathsf{syn}}(A \cong \tilde{A})$. Then there exists a type $f^*\tilde{A} : \mathcal{U}_\alpha$ with $f^\dagger\tilde{A} : f^*\tilde{A} \cong \tilde{A}$, such that both $\bigcirc_{\mathsf{syn}}(f^*\tilde{A} = A)$ and $\bigcirc_{\mathsf{syn}}(f^\dagger\tilde{A} = f)$ 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 $\mathcal{U}_{\bigcirc_{\mathsf{syn}}}$ of purely syntactic types, *i.e.* an element $\mathcal{A}_{\mathsf{syn}} : \mathbf{Alg}_{\mathbb{T}}(\mathcal{U}_{\bigcirc_{\mathsf{syn}}})$. Because we have specified $\blacksquare\!\!\!

Therefore, by Corollary 3.10 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:

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

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 3.10) we may define the second component lying strictly over σ_Π:

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

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

Remark 3.12. 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 et al. [71], Sterling and Angiuli [124]). 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 [10], a result that is easier to prove in generality than any specific type theoretic corollary.[10]

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 $\langle\!| - |\!\rangle_{\mathsf{syn}} : \lfloor \mathsf{Val}(\mathsf{type}) \to \mathsf{Sig} \rfloor_{\mathsf{syn}}$; we must provide parametricity structures for both. First, we may define a collection of small statically connected parametricity structures for types, using Corollary 3.10:

$$\begin{aligned} \mathbf{Type} &: \{\mathcal{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val}(\mathsf{type}) \rfloor_{\mathsf{syn}}\} \\ \mathbf{Type} &\cong \textstyle\sum_{t : \lfloor \mathsf{Val}(\mathsf{type}) \rfloor_{\mathsf{syn}}} \{\mathcal{U}^\alpha_{\bullet_{\mathsf{st}}} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{syn}} \langle\!| t |\!\rangle_{\mathsf{syn}}\} \end{aligned}$$

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

$$\begin{aligned} &\mathsf{type} : \{\mathsf{Sig} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{type} \rfloor_{\mathsf{syn}}\} && \langle\!| - |\!\rangle : \{\mathsf{Val}(\mathsf{type}) \to \mathsf{Sig} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \langle\!| - |\!\rangle_{\mathsf{syn}}\} \\ &\mathsf{type} = (\lfloor \mathsf{type} \rfloor_{\mathsf{syn}}, \mathbf{Type}) && \langle\!| (t, \tilde{t}) |\!\rangle = (\langle\!| t |\!\rangle_{\mathsf{syn}}, \tilde{t}) \end{aligned}$$

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 $\#\mathsf{t}, \#\mathsf{f} : \lfloor \mathsf{Val} \rfloor_{\mathsf{syn}}(\langle\!| \lfloor \mathsf{bool} \rfloor_{\mathsf{syn}} |\!\rangle)$; we must construct parametricity structures for all these. First, we define the collection of computable booleans as follows, using Corollary 3.10 as usual:[11]

$$\begin{aligned} \mathbf{bool} &: \{\mathcal{U}_\alpha \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val}(\langle\!| \mathsf{bool} |\!\rangle) \rfloor_{\mathsf{syn}}\} \\ \mathbf{bool} &\cong \textstyle\sum_{b : \lfloor \mathsf{Val}(\langle\!| \mathsf{bool} |\!\rangle) \rfloor_{\mathsf{syn}}} \bullet_{\mathsf{st}} \bullet_{\mathsf{syn}} \{\tilde{b} : 2 \mid b = \mathsf{case}[\tilde{b}](\lfloor \#\mathsf{t} \rfloor_{\mathsf{syn}}, \lfloor \#\mathsf{f} \rfloor_{\mathsf{syn}})\} \end{aligned}$$

[9] We note that we always have $\text{🔓}_{\mathsf{syn}} = \top$ in scope when constructing an element of $\lfloor \mathsf{Sig} \rfloor_{\mathsf{syn}}$.
[10] Later on we simplify matters further by making use of the closure of presheaf topoi under gluing along continuous functors.
[11] Observe that the second component of the dependent sum is a singleton when $\text{🔓}_{\mathsf{syn}} = \top$.

The application of the closed modality $\bullet_{\mathsf{st}}$ ensures that the values of observable type have no static part (they are "statically connected"). We may therefore define the type of booleans:

$$\mathsf{bool} : \{\mathsf{Val}(\mathsf{type}) \mid \text{🔒}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{bool} \rfloor_{\mathsf{syn}}\}$$
$$\mathsf{bool} = (\lfloor \mathsf{bool} \rfloor_{\mathsf{syn}}, \mathbf{bool})$$

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

$$\#\mathsf{t}, \#\mathsf{f} : \{\mathsf{Val}(\langle\!|\mathsf{bool}|\!\rangle) \mid \text{🔒}_{\mathsf{syn}} \hookrightarrow \lfloor \#\mathsf{t} \rfloor_{\mathsf{syn}}, \lfloor \#\mathsf{f} \rfloor_{\mathsf{syn}}\}$$
$$\#\mathsf{t} = \left(\lfloor \#\mathsf{t} \rfloor_{\mathsf{syn}}, \eta_{\bullet_{\mathsf{st}}}(\eta_{\bullet_{\mathsf{syn}}}(0))\right) \qquad \#\mathsf{f} = \left(\lfloor \#\mathsf{f} \rfloor_{\mathsf{syn}}, \eta_{\bullet_{\mathsf{st}}}(\eta_{\bullet_{\mathsf{syn}}}(1))\right)$$

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 et al. [54] for gluing together two monads along a monad morphism. Emanating from the syntax is an internal monad $\lfloor \Diamond \rfloor_{\mathsf{syn}} : \lfloor \mathsf{Sig} \rfloor_{\mathsf{syn}} \to \lfloor \mathsf{Sig} \rfloor_{\mathsf{syn}}$ 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 $\mathrm{T} : \mathcal{U}^{\beta}_{\bullet_{\mathsf{syn}}} \to \mathcal{U}^{\beta}_{\bullet_{\mathsf{syn}}}$ be such a monad; we furthermore have an internal functor $F : \lfloor \mathsf{Sig} \rfloor_{\mathsf{syn}} \to \mathcal{U}^{\beta}_{\bullet_{\mathsf{syn}}}$ defined by taking the purely semantic part of the collection of modules of every syntactic signature:

$$F(\sigma) = \bullet_{\mathsf{syn}}(\lfloor \mathsf{Val} \rfloor_{\mathsf{syn}}(\sigma))$$

We parameterize the constructions of this section in a monad morphism $\mathsf{run} : \lfloor \Diamond \rfloor_{\mathsf{syn}} \to \mathrm{T}$ over F in the sense of Street [131], *i.e.* an internal natural transformation $\mathsf{run} : \mathrm{T} \circ F \to F \circ \lfloor \Diamond \rfloor_{\mathsf{syn}}$ satisfying a number of coherence conditions. Following Goubault-Larrecq et al. [54], 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 $\mathrm{T}_{\mathbf{X}}(\sigma) : \mathcal{U}_{\beta}$ as follows, writing $\pi_\sigma : \bullet_{\mathsf{syn}}(\mathsf{Val}(\sigma)) \to \mathrm{T}(F(\sigma))$ for the induced projection in $\mathcal{U}^{\beta}_{\bullet_{\mathsf{syn}}}$:

$$\mathrm{T}_{\mathbf{X}} : \textstyle\prod_{\sigma:\mathsf{Sig}} \{\mathcal{U}_\beta \mid \text{🔒}_{\mathsf{syn}} \hookrightarrow \lfloor \mathsf{Val} \rfloor_{\mathsf{syn}}(\lfloor \Diamond \rfloor_{\mathsf{syn}} \sigma)\}$$
$$\mathrm{T}_{\mathbf{X}}(\sigma) \cong \textstyle\sum_{x^\circ : \lfloor \Diamond \rfloor_{\mathsf{syn}} \sigma} \left\{ x^\bullet : \mathrm{T}(\bullet_{\mathsf{syn}}(\mathsf{Val}(\sigma))) \;\middle|\; \mathsf{run}_\sigma(\mathrm{T}(\pi_\sigma)(x^\bullet)) = \eta_{\bullet_{\mathsf{syn}}}(x^\circ) \right\}$$

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

$$\Diamond : \mathsf{Sig} \to \mathsf{Sig}$$
$$\Diamond\sigma = (\lfloor \Diamond \rfloor_{\mathsf{syn}} \sigma, \mathrm{T}_{\mathbf{X}}(\sigma))$$

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 et al. [25], Paviotti [102] (but we do not make any claims in this direction).

Example 3.13. Suppose that ModTT were extended with an operation $\mathsf{throw} : \Diamond\sigma$ for each signature σ, such that $\Diamond$ corresponds to the exception monad. We may glue this together with the internal monad $\mathrm{T}(X) = \bullet_{\mathsf{st}}(1 + X)$ on the internal category of purely semantic parametricity structures. We must define a family of functions $\mathsf{run}_\sigma : \mathrm{T}(F(\sigma)) \to F(\lfloor \Diamond \rfloor_{\mathsf{syn}} \sigma)$. Because $F(\lfloor \Diamond \rfloor_{\mathsf{syn}} \sigma)$ is purely dynamic and $\bullet_{\mathsf{st}}$ is a lex idempotent modality, any such function run_σ is uniquely determined by a map $1 + F(\sigma) \to F(\sigma)$, which we may choose as follows:

$$inl(*) \mapsto \eta_{\bullet_{\mathsf{syn}}}(\lfloor \mathsf{throw} \rfloor_{\mathsf{syn}}) \qquad inr(x) \mapsto \eta_{\bullet_{\mathsf{syn}}}(\lfloor \mathsf{ret} \rfloor_{\mathsf{syn}}(x))$$

Then, the monad $\mathrm{T}_{\mathbf{X}}(\sigma)$ on a parametricity structure $\sigma : \mathrm{Sig}$ associates to each syntactic computation $M : \lfloor \diamond \rfloor_{\mathsf{syn}} \sigma$ 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 3.13. 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 [58], depicted in Fig. 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 $\mathtt{ListQueue} : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{L}}, \mathtt{BatchedQueue} : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{R}}$; these can be combined into $\mathtt{Q} : \lfloor \mathsf{Val}(\mathtt{QUEUE}) \rfloor_{\mathsf{syn}}$ by splitting:

$$\mathtt{Q} = [\mathbin{🔓}_{\mathsf{syn/l}} \hookrightarrow \mathtt{ListQueue}, \mathbin{🔓}_{\mathsf{syn/r}} \hookrightarrow \mathtt{BatchedQueue}]$$

We may define a purely dynamic type that represents the invariant structure on a pair of queues using Corollary 3.10, writing $\mathsf{bits} = 2^\star$ for the 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.

$$\begin{aligned}
&\mathsf{invariant} : \{\mathscr{U}^{\alpha}_{\bullet_{\mathsf{st}}} \mid \mathbin{🔓}_{\mathsf{syn}} \hookrightarrow \bullet_{\mathsf{st}} \mathbin{\bigcirc}_{\mathsf{syn}} \mathsf{Val}(\mathtt{Q.t})\} \\
&\mathsf{invariant} \cong \\
&\quad \textstyle\sum_{q : \lfloor \mathsf{Val} \rfloor_{\mathsf{syn}}(\langle\!|\mathtt{Q.t}|\!\rangle)} \\
&\quad \bullet_{\mathsf{syn}} \{\vec{x}, \vec{y}, \vec{z} : \bullet_{\mathsf{st}} \mathsf{bits} \mid \vec{x} = (\vec{y} + rev(\vec{z})) \wedge q = [\mathbin{🔓}_{\mathsf{syn/l}} \hookrightarrow \lceil \vec{x} \rceil \mid \mathbin{🔓}_{\mathsf{syn/r}} \hookrightarrow (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]\}
\end{aligned}$$

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

Theorem 4.1. *Let $f : \lfloor \mathtt{QUEUE} \to \langle\!|\mathsf{bool}|\!\rangle \rfloor_{\mathsf{syn}}$; then we have $f(\mathtt{ListQueue}) = f(\mathtt{BatchedQueue})$.*

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

$$b : \{\langle\!|\mathsf{bool}|\!\rangle \mid \mathbin{🔓}_{\mathsf{syn/l}} \hookrightarrow \lfloor f(\mathtt{ListQueue}) \rfloor_{\mathsf{L}}, \mathbin{🔓}_{\mathsf{syn/r}} \hookrightarrow \lfloor f(\mathtt{BatchedQueue}) \rfloor_{\mathsf{R}}\}$$

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

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 [10]; 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 2.4). The copower $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ will then serve as a suitable index for binary parametricity.

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

structure ListQueue : 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 BatchedQueue : 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
```

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

(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 $\mathbf{X}$ of *phase separated parametricity structures*: a type in the internal language of $\mathbf{X}$ will have several aspects corresponding to the orthogonal distinctions ((left syntax, right syntax), semantics) and (static, dynamic). The topos $\mathbf{X}$ then has enough structure to model all of ParamTT.

A simulation over $\mathtt{Q} = [\mathbb{1}^{\mathsf{syn}}_{\mathsf{l}} \hookrightarrow \mathtt{ListQueue}, \mathbb{1}^{\mathsf{syn}}_{\mathsf{r}} \hookrightarrow \mathtt{BatchedQueue}]$ consists of the following data:

$$\begin{aligned} t &: \{\mathsf{Val}(\mathsf{type}) \mid \mathbb{1}^{\mathsf{syn}} \hookrightarrow \mathtt{Q.t}\} \\ \mathit{emp} &: \{\mathsf{Val}(\langle\!|t|\!\rangle) \mid \mathbb{1}^{\mathsf{syn}} \hookrightarrow \mathtt{Q.emp}\} \\ \mathit{ins} &: \{\mathsf{Val}(\langle\!|\mathsf{bool} * t \multimap t|\!\rangle) \mid \mathbb{1}^{\mathsf{syn}} \hookrightarrow \mathtt{Q.ins}\} \\ \mathit{rem} &: \{\mathsf{Val}(\langle\!|t \multimap \mathsf{bool} * t|\!\rangle) \mid \mathbb{1}^{\mathsf{syn}} \hookrightarrow \mathtt{Q.rem}\} \end{aligned}$$

These operations are implemented in ParamTT as follows.

$$\begin{aligned} t &= (\mathtt{Q.t}, \mathsf{invariant}) \\ \mathit{emp} &= (\mathtt{Q.emp}, (\langle\rangle, \langle\rangle, \langle\rangle)) \\ \mathit{ins}((b,x),(q,(\vec{x},\vec{y},\vec{z}))) &= (\mathtt{Q.ins}(b,q), \eta_{\mathsf{T}}([\mathbb{1}^{\mathsf{syn}}_{\mathsf{l}} \hookrightarrow b :: q, \mathbb{1}^{\mathsf{syn}}_{\mathsf{r}} \hookrightarrow (\mathtt{fst}(q), b :: \mathtt{snd}(q))], (x :: \vec{x}, x :: \vec{y}, \vec{z}))) \\ \mathit{rem}(q,(\vec{x},\vec{y},\vec{z})).1 &= \mathtt{Q.rem}(q) \\ \mathit{rem}(q,(\langle\rangle,\langle\rangle,\langle\rangle)).2 &= \mathit{throw}_{\mathsf{T}} \\ \mathit{rem}(q,(\vec{x}\ldots x),x :: \vec{y},\vec{z}).2 &= \eta_{\mathsf{T}}((\lceil x\rceil, [\mathbb{1}^{\mathsf{syn}}_{\mathsf{l}} \hookrightarrow \lceil\vec{x}\rceil \mid \mathbb{1}^{\mathsf{syn}}_{\mathsf{r}} \hookrightarrow (\lceil\vec{y}\rceil, \lceil\vec{z}\rceil)]), (x,(\vec{x},\vec{y},\vec{z}))) \\ \mathit{rem}(q,((\vec{x}\ldots x),\langle\rangle,\vec{z}\ldots x)).2 &= \eta_{\mathsf{T}}((\lceil x\rceil, [\mathbb{1}^{\mathsf{syn}}_{\mathsf{l}} \hookrightarrow \lceil\vec{x}\rceil \mid \mathbb{1}^{\mathsf{syn}}_{\mathsf{r}} \hookrightarrow (\lceil \mathit{rev}(\vec{z})\rceil, \lceil\vec{y}\rceil)]), (x,(\vec{x},\mathit{rev}(\vec{z}),\vec{y}))) \end{aligned}$$

where

$$\begin{aligned} &\mathsf{invariant} : \{\mathscr{U}^{\alpha}_{\bullet_{\mathsf{st}}} \mid \mathbb{1}^{\mathsf{syn}} \hookrightarrow \bullet_{\mathsf{st}} \bigcirc_{\mathsf{syn}} \mathsf{Val}(\mathtt{Q.t})\} \\ &\mathsf{invariant} \cong \textstyle\sum_{q : \bigcirc_{\mathsf{syn}} \mathsf{Val}(\langle\!|\mathtt{Q.t}|\!\rangle)} \bullet_{\mathsf{syn}} \{\vec{x},\vec{y},\vec{z} : \bullet_{\mathsf{st}}\mathsf{bits} \mid \vec{x} = (\vec{y} + \mathit{rev}(\vec{z})) \wedge q = [\mathbb{1}^{\mathsf{syn}}_{\mathsf{l}} \hookrightarrow \lceil\vec{x}\rceil \mid \mathbb{1}^{\mathsf{syn}}_{\mathsf{r}} \hookrightarrow (\lceil\vec{y}\rceil, \lceil\vec{z}\rceil)]\} \end{aligned}$$

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

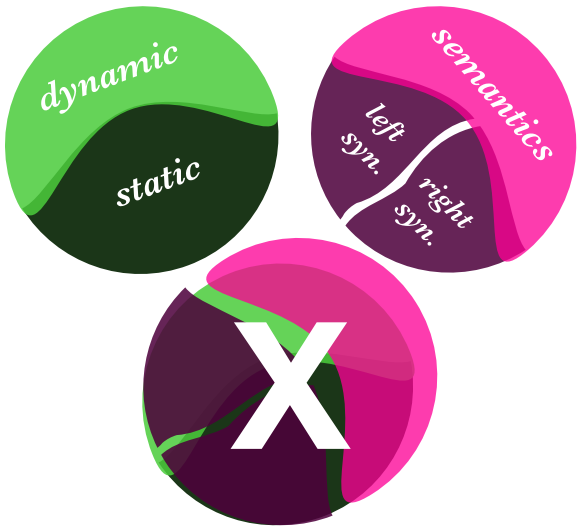


Fig. 6. A geometrical depiction of the topos **X** of parametricity structures: dark and light regions in the same color-range indicate complementary open and closed subtopoi corresponding to the static–dynamic and syntactic–semantic distinctions.

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 [87] and Kripke [72] to interpret the logic into the preorder $\mathcal{O}_X$ 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 $U \in \mathcal{O}_X$ 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 Oles [100], Reynolds [112], but $\mathscr{E}$ is not likely to be of the form $\mathrm{Sh}(X)$ 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 [10]. 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 [7], 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;[12] 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 $\mathbf{X}$ is defined by specifying a logos conventionally called $\mathrm{Sh}(\mathbf{X})$, the category of "sheaves on $\mathbf{X}$"; a continuous map of topoi $f : \mathbf{X} \longrightarrow \mathbf{Y}$ is defined by specifying a morphism of logoi $f^* : \mathrm{Sh}(\mathbf{Y}) \longrightarrow \mathrm{Sh}(\mathbf{X})$ 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 $\mathrm{Sh}(-) : \mathbf{Topos}^{\mathrm{op}} \longrightarrow \mathbf{Logos}$.

[12] 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.

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.

Definition 5.4 (Direct image). For a morphism of topoi $f : \mathbf{X} \longrightarrow \mathbf{Y}$, the cocontinuity of the inverse image $f^* : \mathrm{Sh}(\mathbf{Y}) \longrightarrow \mathrm{Sh}(\mathbf{X})$ implies that it is a left adjoint $f^* \dashv f_*$; the right adjoint f_* is called the *direct image*.

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 $\mathbf{Top}(X)$, setting $\mathrm{Sh}(\mathbf{Top}(X))$ 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 5.5. 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 [51].

5.2 The language of topoi

We give a crash course in the language of topoi insofar as it is pertinent to the present paper. This section can be skipped and referred back to by those comfortable with topoi; not all (or even most) of the material presented here is necessary to understand our constructions, but we provide it to assist the reader in developing topological intuitions for topoi which were important for developing the present work. More details can be found in several cited resources [7, 70, 122, 138, 144].

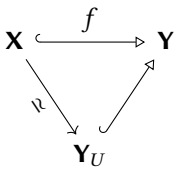
Definition 5.6 (Subtopos). A subtopos $\mathbf{X} \subseteq \mathbf{Y}$ is given by a logos $\mathrm{Sh}(\mathbf{X})$ that is a subcategory of $\mathrm{Sh}(\mathbf{Y})$.

Definition 5.7 (Embedding). A morphism of topoi $f : \mathbf{X} \longrightarrow \mathbf{Y}$ is called an *embedding*, written $f : \mathbf{X} \hookrightarrow \mathbf{Y}$, when the direct image f_* functor is fully faithful.

Definition 5.8 (Equivalence). A morphism of topoi $f : \mathbf{X} \longrightarrow \mathbf{Y}$ is called an *equivalence* when the inverse image functor f^* (equivalently, the direct image functor f_*) is an equivalence of categories.

Definition 5.9 (Opens of a topos). An *open* of a topos $\mathbf{X}$ is defined to be a subterminal object in $\mathrm{Sh}(\mathbf{X})$, *i.e.* a proof-irrelevant proposition in the internal type theory of $\mathbf{X}$. We will write $\mathcal{O}_{\mathbf{X}}$ for the frame of opens of the topos $\mathbf{X}$. An open $U \in \mathcal{O}_{\mathbf{X}}$ gives rise to an *open subtopos* $\mathbf{X}_U \subseteq \mathbf{X}$: we define $\mathrm{Sh}(\mathbf{X}_U)$ to be the full subcategory of $\mathrm{Sh}(\mathbf{X})$ spanned by objects E such that the canonical map $E \longrightarrow E^U$ is an isomorphism. Equivalently, $\mathrm{Sh}(\mathbf{X}_U)$ is the slice logos $\mathrm{Sh}(\mathbf{X})_{/U}$.

Definition 5.10 (Open immersion). An embedding of topoi $f : \mathbf{X} \hookrightarrow \mathbf{Y}$ is called an *open immersion*, written $f : \mathbf{X} \hookrightarrow\!\!\triangleright\, \mathbf{Y}$ when it factors through an equivalence $\mathbf{X} \simeq \mathbf{Y}_U$ and an open subtopos inclusion $\mathbf{Y}_U \subseteq \mathbf{Y}$ for some open $U \in \mathcal{O}_{\mathbf{Y}}$ in the following sense:



Definition 5.11 (Closed complement). Let $U \in \mathcal{O}_{\mathbf{X}}$ be an open of a topos $\mathbf{X}$; the *closed complement* of the open subtopos $\mathbf{X}_U$ can be defined by means of the full subcategory $\mathrm{Sh}(\mathbf{X})^{\vee U} \subseteq \mathrm{Sh}(\mathbf{X})$ spanned by objects E such that the canonical map $E \longrightarrow E \sqcup_{E\times U} U$ is an isomorphism, where $E \sqcup_{E\times U} U$ is the following pushout:

$$\begin{array}{ccc} E \times U & \longrightarrow & U \\ \downarrow & & \downarrow \\ E & \longrightarrow & E \sqcup_{E\times U} U \end{array}$$

Then the closed complement $\mathbf{X}_{\setminus U} \subseteq \mathbf{X}$ is defined by the identification $\mathrm{Sh}(\mathbf{X}_{\setminus U}) = \mathrm{Sh}(\mathbf{X})^{\vee U}$.

It is not hard to show that a sheaf on the closed complement $\mathbf{X}_{\setminus U}$ is the same as a sheaf E on $\mathbf{X}$ that is *U-connected* in the sense that $E \times U \cong U$, or equivalently, $E^U \cong \mathbf{1}_{\mathrm{Sh}(\mathbf{X})}$.

Definition 5.12 (Closed immersion). Likewise an embedding of topoi is called a *closed* immersion, written $f : \mathbf{X} \hookrightarrow \mathbf{Y}$ when it factors through an equivalence $\mathbf{X} \simeq \mathbf{Y}_{\setminus U}$ and a closed subtopos inclusion $\mathbf{Y}_{\setminus U} \subseteq \mathbf{Y}$ for some open $U \in \mathcal{O}_{\mathbf{Y}}$.

The open and closed subtopoi corresponding to $U \in \mathcal{O}_{\mathbf{X}}$ are complementary in the sense of classical topology, but this does not mean there is no substance lying between them. Just as in classical topology, between an open subspace and its closed complement lies a "boundary" $\partial_{\mathbf{X}} U = \overline{\mathbf{X}_U} \cap \mathbf{X}_{\setminus U} \subseteq \mathbf{X}$ where $\overline{U}$ is the closure of U.

Definition 5.13 (Closure of an open subtopos). If $U \in \mathcal{O}_{\mathbf{X}}$ is an open of a topos $\mathbf{X}$, the *closure* of U is the smallest closed subtopos $\overline{\mathbf{X}_U} \subseteq \mathbf{X}$ that contains $\mathbf{X}_U$. Writing $\boldsymbol{j} : \mathbf{X}_U \hookrightarrow \mathbf{X}$ for the open immersion corresponding to U, we have a lex idempotent monad $\boldsymbol{j}_*\boldsymbol{j}^* : \mathcal{O}_{\mathbf{X}} \longrightarrow \mathcal{O}_{\mathbf{X}}$ on the frame of opens given by the adjunction $\boldsymbol{j}^* \dashv \boldsymbol{j}_*$. Considering the characterization of $\mathrm{Sh}(\mathbf{X}_U)$ as the slice $\mathrm{Sh}(\mathbf{X})_{/U}$, we see that $\boldsymbol{j}_*\boldsymbol{j}^*V$ is the Heyting implication $(U \Rightarrow V)$ for any $V \in \mathcal{O}_{\mathbf{X}}$. The *closure* of U can then be computed to be the closed complement of the open $\boldsymbol{j}_*\boldsymbol{j}^*\bot = \neg U$, *i.e.* the closed subtopos $\mathbf{X}_{\setminus\neg U} \subseteq \mathbf{X}$. Explicitly, a sheaf on $\mathbf{X}_{\setminus\neg U}$ is a sheaf on $\mathbf{X}$ that is $\neg U$-connected, *i.e.* becomes a singleton when restricted to $\mathbf{X}_{\neg U}$.

Definition 5.14 (Fringe of an open subtopos). Let $U \in \mathcal{O}_{\mathbf{X}}$ be an open of a topos $\mathbf{X}$; the *fringe* $\partial_{\mathbf{X}} U \subseteq \mathbf{X}$ of the open subtopos $\mathbf{X}_U$ is defined to be the intersection of the closure of $\mathbf{X}_U$ with the closed complement $\mathbf{X}_{\setminus U}$.

$$\begin{array}{ccc} \partial_{\mathbf{X}} U & \hookrightarrow & \overline{\mathbf{X}_U} = \mathbf{X}_{\setminus\neg U} \\ \downarrow & & \downarrow \\ \mathbf{X}_{\setminus U} & \hookrightarrow & \mathbf{X} \end{array}$$

By a further computation, we may observe that the fringe $\partial_{\mathbf{X}} U$ is the closed subtopos corresponding to the open $U \vee \neg U \in \mathcal{O}_{\mathbf{X}}$, *i.e.* we have $\partial_{\mathbf{X}} U = \mathbf{X}_{\setminus U\vee\neg U}$. This is not trivial unless U is simultaneously closed and open!

Remark 5.15. The above shows the geometric sense in which the open and closed subtopoi are complementary; although we always have $\boldsymbol{j}_*\boldsymbol{j}^*\boldsymbol{i}_*\boldsymbol{i}^*E \cong \mathbf{1}_{\mathrm{Sh}(\mathbf{X})}$, we do not have $\boldsymbol{i}_*\boldsymbol{i}^*\boldsymbol{j}_*\boldsymbol{j}^*E \cong \mathbf{1}_{\mathrm{Sh}(\mathbf{X})}$ except when U is clopen.

Definition 5.16 (The fringe functor). Given an open $U \in \mathcal{O}_{\mathbf{X}}$, define what is called the *fringe functor* $F_U : \mathrm{Sh}(\mathbf{X}_U) \longrightarrow \mathrm{Sh}(\mathbf{X}_{\setminus U})$ to be the following composite:

$$\mathrm{Sh}(\mathbf{X}_U) \xrightarrow{\boldsymbol{j}_*} \mathrm{Sh}(\mathbf{X}) \xrightarrow{\boldsymbol{i}^*} \mathrm{Sh}(\mathbf{X}_{\setminus U}) \qquad (F_U)$$

The relationship between the fringe functor corresponding to $U \in \mathcal{O}_{\mathbf{X}}$ (Definition 5.16) and the fringe of the open subtopos $\mathbf{X}_U \subseteq \mathbf{X}$ (Definition 5.14) is expressed in the Lemma 5.17 and Theorem 5.18 below.

Lemma 5.17 (Wraith [144]). *If E is a sheaf on $\mathbf{X}_U$, the sheaf $F_U E$ on $\mathbf{X}_{\setminus U}$ is trivial away from the fringe $\partial_{\mathbf{X}} U \subseteq \mathbf{X}_{\setminus U}$,* i.e. *in $F_U E$ restricts to the terminal sheaf in the open complement of $\partial_{\mathbf{X}} U \subseteq \mathbf{X}_{\setminus U}$.*

Proof. As a closed subtopos of $\mathbf{K} := \mathbf{X}_{\setminus U}$, the fringe $\partial_{\mathbf{X}} U$ is the complement of the open $\boldsymbol{i}^*(U \vee \neg U)$, which is equal to $\boldsymbol{i}^* \neg U$ because inverse image is cocontinuous and $\boldsymbol{i}^* U = \bot$. Therefore we may reconstruct $\partial_{\mathbf{X}} U \subseteq \mathbf{K}$ as $\mathbf{K}_{\setminus \boldsymbol{i}^* \neg U}$ and our goal is to show that $F_U E \in \mathrm{Sh}(\mathbf{K}_{\setminus \boldsymbol{i}^* \neg U}) \subseteq \mathrm{Sh}(\mathbf{K})$, which is the same as to show $F_U E \times \boldsymbol{i}^* \neg U \cong \boldsymbol{i}^* \neg U$.

$$\begin{aligned}
F_U E \times \boldsymbol{i}^* \neg U &\cong \boldsymbol{i}^* \boldsymbol{j}_* E \times \boldsymbol{i}^* \neg U && \text{Definition 5.16} \\
&\cong \boldsymbol{i}^* (\boldsymbol{j}_* E \times \neg U) && \boldsymbol{i}^* \text{ lex} \\
&\cong \boldsymbol{i}^* (\boldsymbol{j}_* \boldsymbol{j}^* \boldsymbol{j}_* E \times \neg U) && \boldsymbol{j}_* \boldsymbol{j}^* \text{ idempotent} \\
&\cong \boldsymbol{i}^* \left((\boldsymbol{j}_* E)^U \times \neg U \right) && \boldsymbol{j}_* \boldsymbol{j}^* \cong (-)^U \\
&\cong \boldsymbol{i}^* \neg U
\end{aligned}$$

□

Theorem 5.18 (Artin gluing / Recollement [10]). *A topos $\mathbf{X}$ can be reconstructed up to equivalence from the data of a partition $U \in \mathcal{O}_{\mathbf{X}}$ into open and closed subtopoi:*[13]

$$\begin{array}{ccc}
\mathrm{Sh}(\mathbf{X}) \simeq \{\mathrm{Sh}(\mathbf{X}_{\setminus U})\} \downarrow F_U & \longrightarrow & \mathrm{Sh}(\mathbf{X}_{\setminus U})^{\to} \\
\Big\downarrow \boldsymbol{j}^* & & \Big\downarrow \mathrm{cod} \\
\mathrm{Sh}(\mathbf{X}_U) & \xrightarrow[F_U]{} & \mathrm{Sh}(\mathbf{X}_{\setminus U})
\end{array}$$

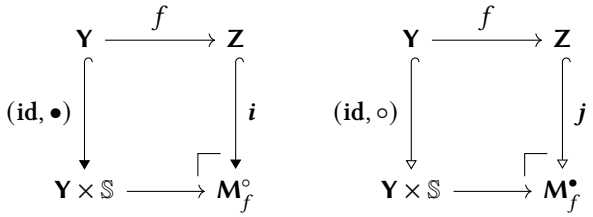
Conversely if $F : \mathrm{Sh}(\mathbf{U}) \longrightarrow \mathrm{Sh}(\mathbf{K})$ is a left exact and accessible functor between logoi, then there exists a topos $\mathbf{X}$ together with an open $U \in \mathcal{O}_{\mathbf{X}}$ such that $\mathbf{U} \simeq \mathbf{X}_U$ and $\mathbf{K} \simeq \mathbf{X}_{\setminus U}$ configured like so:

$$\begin{array}{ccc}
\mathrm{Sh}(\mathbf{X}) & \longrightarrow & \mathrm{Sh}(\mathbf{K})^{\to} \\
\Big\downarrow \boldsymbol{j}^* & & \Big\downarrow \mathrm{cod} \\
\mathrm{Sh}(\mathbf{U}) & \xrightarrow[F]{} & \mathrm{Sh}(\mathbf{K})
\end{array}$$

Above, $\mathbf{X}$ is called the Artin gluing *of F; the open $U \in \mathcal{O}_{\mathbf{X}}$ that reconstructs $(\mathbf{U}, \mathbf{K})$ as $(\mathbf{X}_U, \mathbf{X}_{\setminus U})$ can be defined to be the subterminal sheaf $\left(\mathbf{1}_{\mathrm{Sh}(\mathbf{U})}, \emptyset_{\mathrm{Sh}(\mathbf{K})} \longrightarrow F(\mathbf{1}_{\mathrm{Sh}(\mathbf{U})})\right)$.*

[13] The construction takes place in the category of categories and functors, rather than the category of logoi and morphisms of logoi; this is because the fringe funtor F_U need not be cocontinuous.

Remark 5.19 (Geometric gluing). In certain cases, including those investigated in this paper, the fringe functor F_U turns out to be either the direct image or inverse image part of a morphism of topoi; in those cases, a construction of the Artin gluing taking place in the category of *topoi* rather than the category of categories is available. When F is the global sections functor, the Artin gluing is called the *scone* or *Sierpiński cone*; when $F = f_*$ (resp. $F = f^*$) for a morphism of topoi $f : \mathbf{Y} \longrightarrow \mathbf{Z}$, the Artin gluing is referred to by Johnstone [69] as the open (resp. closed) mapping cylinder of f, depicted below:



Above we have $\mathrm{Sh}(\mathbf{M}^{\circ}_f) \simeq \{\mathrm{Sh}(\mathbf{Z})\} \downarrow f_*$ and $\mathrm{Sh}(\mathbf{M}^{\bullet}_f) \simeq \{\mathrm{Sh}(\mathbf{Y})\} \downarrow f^*$. Indeed, our Construction 5.27 of the topos of parametricity structures in Section 5.4 is an example of the open mapping cylinder depicted above on the left.

5.2.1 Classifying topoi and geometric figures. The same topos $\mathbf{X}$ can be profitably understood in two *different* ways: what happens when you map into it, and what happens when you map out of it. These two perspectives correspond respectively to viewing a topos as *classifying space* of some kind of data, *vs.* as a *geometrical figure*; these correspond to algebraic and geometrical perspectives on topoi respectively. In other words, a morphism of topoi $\mathbf{X} \longrightarrow \mathbf{Y}$ can be thought of as constructing a "point" of $\mathbf{Y}$ in the language of $\mathrm{Sh}(\mathbf{X})$, but it can also be thought of as drawing an $\mathbf{X}$-shaped figure in $\mathbf{Y}$. We use both perspectives in this paper in a critical way; in particular, the Sierpiński topos $\mathbb{S}$ appears in our construction both as a geometrical figure and as a classifier (see Remark 5.28).

Example 5.20 (The punctual topos). The logos of sets **Set** is the category of sheaves on the one-point space. Therefore we define the *punctual* topos $\mathbb{1} : \mathbf{Topos}$ to by the identification $\mathrm{Sh}(\mathbb{1}) = \mathbf{Set}$. From the geometrical point of view, a morphism $\mathbb{1} \longrightarrow \mathbf{X}$ corresponds to constructing a *point* of $\mathbf{X}$. From the algebraic point of view, one thinks of a morphism $\mathbf{X} \longrightarrow \mathbb{1}$ as constructing no data whatsoever in the language of $\mathrm{Sh}(\mathbf{X})$. For this reason, there is always a unique such morphism and hence $\mathbb{1}$ is the terminal topos.

Example 5.21 (Presheaves and finite limit theories). 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.* $\mathrm{Sh}\big(\widehat{\mathscr{C}}\big) = \mathrm{Pr}(\mathscr{C})$. Suppose that $\mathscr{C}$ has finite limits, *i.e.* $\mathscr{C}$ is the classifying category for a finite limit theory $\mathbb{T}$; then Diaconescu's theorem [43] states that a morphism $\mathbf{Y} \longrightarrow \widehat{\mathscr{C}}$ corresponds to a left exact functor $\mathscr{C} \longrightarrow \mathrm{Sh}(\mathbf{Y})$, *i.e.* a model of $\mathbb{T}$ in $\mathrm{Sh}(\mathbf{Y})$.

Hence the algebraic perspective says that $\widehat{\mathscr{C}}$ is the classifier of $\mathbb{T}$-models. On the other hand, $\widehat{\mathscr{C}}$ is a geometric figure that captures the configuration of *all* $\mathbb{T}$-models and their homomorphism. It is helpful to consider the case where $\mathbb{T}$ is the theory of groups: then, for example, a morphism of topoi $\widehat{\mathscr{C}} \longrightarrow \mathbb{S}$ corresponds to a diagram of truth values that is labeled by the collection of *all* groups and group homomorphisms.

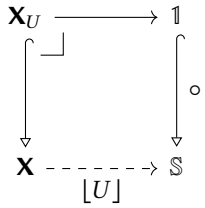
Example 5.22 (Sierpiński topos). The logos of *families* of sets $\mathbf{Set}^{\rightarrow}$ is also the category of sheaves on the classic Sierpiński space $S = \{\{\}, \{\circ\}, \{\bullet, \circ\}\}$. Hence we define the Sierpiński *topos* $\mathbb{S}$ by the identification $\mathrm{Sh}(\mathbb{S}) = \mathbf{Set}^{\rightarrow}$. As a geometrical figure, the Sierpiński topos is a *directed interval* in that it has two points $\bullet, \circ : \mathbb{1} \longrightarrow \mathbb{S}$ and a morphism $\bullet \longrightarrow \circ$. A morphism $\mathbb{S} \longrightarrow \mathbf{X}$ corresponds to a

	Geometric perspective	Algebraic perspective
$\bullet : \mathbb{1} \hookrightarrow \mathbb{S}$	the closed point of $\mathbb{S}$	the proposition $\bot$ in **Set**
$\circ : \mathbb{1} \hookrightarrow \mathbb{S}$	the open point of $\mathbb{S}$	the proposition $\top$ in **Set**
$\bullet \longrightarrow \circ$	the directed interval	the implication $\bot \leq \top$

Table 1. Investigating the points of the Sierpiński topos from the geometrical (topos as figure) and algebraic perspectives (topos as classifier).

pair of points $x, y : \mathbb{1} \longrightarrow \mathbf{X}$ together with a morphism of points $x \longrightarrow y$. The open point determines a distinguished open that we might write $\{\circ\} \in \mathscr{O}_{\mathbb{S}}$, which is defined by the subterminal family of sets $\emptyset_{\mathbf{Set}} \longrightarrow \mathbf{1}_{\mathbf{Set}} : \mathbf{Set}^{\rightarrow}$.

From the algebraic point of view, $\mathbb{S}$ classifies *opens* or *propositions* in that every open subtopos $\mathbf{X}_U \subseteq \mathbf{X}$ arises in an essentially unique way by pullback along the open point $\circ : \mathbb{1} \hookrightarrow \mathbb{S}$:



The characteristic map $\lfloor U \rfloor$ has a universal property in the inverse image direction, namely it is the unique map such that $\lfloor U \rfloor^*\{\circ\} = U \in \mathscr{O}_{\mathbf{X}}$. A geometric/pointwise intuition is also helpful: the characteristic map $\lfloor U \rfloor : \mathbf{X} \longrightarrow \mathbb{S}$ sends a point $x \in \mathbf{X}$ to the open point $\circ \in \mathbb{S}$ if $x \in \mathbf{X}_U$, and sends it to the closed point $\bullet$ if $x \notin \mathbf{X}_U$.

Computation 5.23. Given an open $U \in \mathscr{O}_{\mathbf{X}}$, how is the corresponding characteristic map $\lfloor U \rfloor : \mathbf{X} \longrightarrow \mathbb{S}$ actually constructed? Dualizing into the language of logoi, we must construct a lex and cocontinous functor $\lfloor U \rfloor^* : \mathrm{Sh}(\mathbb{S}) \longrightarrow \mathrm{Sh}(\mathbf{X})$. Note that $\mathrm{Sh}(\mathbb{S})$ is the category of presheaves on the interval category $[1] = \{0 \rightarrowtail 1\}$ and that moreover, $[1]$ has finite limits. Hence as discussed in Example 5.21, Diaonescu's theorem states that a lex and cocontinuous functor $\mathrm{Sh}(\mathbb{S}) \longrightarrow \mathrm{Sh}(\mathbf{X})$ is the same thing as a lex functor $[1] \longrightarrow \mathrm{Sh}(\mathbf{X})$, which can be seen to be the same thing as a subterminal object in $\mathrm{Sh}(\mathbf{X})$, *i.e.* an open of $\mathbf{X}$.

A slightly more elementary way to understand what is happening here is to observe that $\mathrm{Sh}(\mathbb{S})$ is the free cocompletion of $[1]$, so a cocontinuous morphism out of $\mathrm{Sh}(\mathbb{S})$ has freedom only in the base case: it must take formal colimits of $[1]$ to actual colimits of $\mathrm{Sh}(\mathbf{X})$.

It is helpful to investigate the morphisms $\circ, \bullet : \mathbb{1} \longrightarrow \mathbb{S}$ in terms of the geometry–algebra duality, which we depict in Table 1.

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 5.24. 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 $\circ : \mathbb{1} \hookrightarrow\!\triangleright\, \mathbb{S}$ is the codomain functor $\mathrm{cod} : \mathbf{Set}^{\to} \longrightarrow \mathbf{Set}$, and the inverse image of the closed point $\bullet : \mathbb{1} \hookrightarrow\!\blacktriangleright\, \mathbb{S}$ is the domain functor $\mathrm{dom} : \mathbf{Set}^{\to} \longrightarrow \mathbf{Set}$.

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 5.21). $\widehat{\mathscr{C}_{\mathbb{T}}}$ can be thought of as a topos of generalized syntax.

We consider the characteristic map $\lfloor \text{🔒}_{\mathsf{st}} \rfloor : \widehat{\mathscr{C}_{\mathbb{T}}} \longrightarrow \mathbb{S}$ of the open $\text{🔒}_{\mathsf{st}} \in \mathcal{O}_{\widehat{\mathscr{C}_{\mathbb{T}}}}$, so we have $\lfloor \text{🔒}_{\mathsf{st}} \rfloor^* \{\circ\} = \text{🔒}_{\mathsf{st}}$. We will see in Computation 5.25 that the direct image $\lfloor \text{🔒}_{\mathsf{st}} \rfloor_* : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}}) \longrightarrow \mathrm{Sh}(\mathbb{S})$ can be viewed as a "phase separated" version of the global sections functor, sending each object to the weakening map from its closed elements to their static parts.

Computation 5.25. To verify the intuition above, we proceed to compute the action of the direct image $\lfloor \text{🔒}_{\mathsf{st}} \rfloor_*$ on a presheaf $X : \mathrm{Pr}(\mathscr{C}_{\mathbb{T}})$. First, we recognize that the direct image $\lfloor \text{🔒}_{\mathsf{st}} \rfloor_* X$ should be a family of sets (*i.e.* a $[1]^{\mathrm{op}}$-shaped diagram of sets) by definition; we probe this family of sets at the map $0 \rightarrowtail 1 : [1]$ using the Yoneda lemma, adjointness, and the fact that $\lfloor \text{🔒}_{\mathsf{st}} \rfloor$ is the characteristic map of the open 🔒_{st}:

$$
\begin{aligned}
\lfloor \text{🔒}_{\mathsf{st}} \rfloor_* X \{0 \rightarrowtail 1\} &\cong \mathrm{Hom}_{\mathrm{Sh}(\mathbb{S})}\big(\mathsf{y}_{[1]}\{0 \rightarrowtail 1\}, \lfloor \text{🔒}_{\mathsf{st}} \rfloor_* X\big) && \text{by Yoneda lemma} \\
&\cong \mathrm{Hom}_{\mathrm{Sh}(\mathbb{S})}\big(\{\{\circ\} \rightarrowtail \mathbf{1}_{\mathrm{Sh}(\mathbb{S})}\}, \lfloor \text{🔒}_{\mathsf{st}} \rfloor_* X\big) && \text{by Example 5.22} \\
&\cong \mathrm{Hom}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}\big(\lfloor \text{🔒}_{\mathsf{st}} \rfloor^* \{\{\circ\} \rightarrowtail \mathbf{1}_{\mathrm{Sh}(\mathbb{S})}\}, X\big) && \text{by } \lfloor \text{🔒}_{\mathsf{st}} \rfloor^* \dashv \lfloor \text{🔒}_{\mathsf{st}} \rfloor_* \\
&\cong \mathrm{Hom}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}\big(\{\text{🔒}_{\mathsf{st}} \rightarrowtail \mathbf{1}_{\mathrm{Pr}(\mathscr{C}_{\mathbb{T}})}\}, X\big) && \text{by def. of } \lfloor \text{🔒}_{\mathsf{st}} \rfloor
\end{aligned}
$$

Hence $\lfloor \text{🔒}_{\mathsf{st}} \rfloor_* X$ is the diagram $X(\mathbf{1}_{\mathscr{C}_{\mathbb{T}}}) \longrightarrow X(\text{🔒}_{\mathsf{st}})$ of sets that projects from a global element (closed term) of X its static part.

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 $\mathbf{Y}$ a topos. The copower $E \cdot \mathbf{Y} = \coprod_{e \in E} \mathbf{Y}$ is a topos, whose corresponding logos may be computed as follows: $\mathrm{Sh}(E \cdot \mathbf{Y}) = \mathrm{Sh}\big(\coprod_{e \in E} \mathbf{Y}\big) = \prod_{e \in E} \mathrm{Sh}(\mathbf{Y}) = \mathrm{Sh}(\mathbf{Y})^E$.

The codiagonal morphism of topoi $\nabla : E \cdot \mathbf{Y} \longrightarrow \mathbf{Y}$ corresponds under inverse image to the diagonal morphism of logoi $\nabla^* : \mathrm{Sh}(\mathbf{Y}) \longrightarrow \mathrm{Sh}(\mathbf{Y})^E$; indeed, the diagonal map is lex as it is right adjoint to the colimit functor $\mathrm{colim}_E : \mathrm{Sh}(\mathbf{Y})^E \longrightarrow \mathrm{Sh}(\mathbf{Y})$, 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 $E := 2$ 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:

$$
2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} \xrightarrow{\ \nabla\ } \widehat{\mathscr{C}_{\mathbb{T}}} \xrightarrow{\ \lfloor \text{🔒}_{\mathsf{st}} \rfloor\ } \mathbb{S} \qquad \boldsymbol{\rho} \tag{1}
$$

Computation 5.26. The direct image $\boldsymbol{\rho}_* : \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ takes a pair $(X_L, X_R) : \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2$ of (generalized) syntactic objects to $\lfloor \text{🔒}_{\mathrm{st}} \rfloor_* X_L \times \lfloor \text{🔒}_{\mathrm{st}} \rfloor_* X_R$, the product of their phase separated global sections.

Proof. To see that this is the case, we first dualize Diagram 1 into the language of logoi.

$$\mathrm{Sh}(\mathbb{S}) \xrightarrow{\lfloor \text{🔒}_{\mathrm{st}} \rfloor^*} \mathrm{Pr}(\mathscr{C}_\mathbb{T}) \xrightarrow{\nabla^*} \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2 \qquad \boldsymbol{\rho}^* \tag{2}$$

In Diagram 2 above, the inverse image ∇^* is the diagonal functor and hence its right adjoint ∇_* is the product functor. Hence we may compute the direct image part of $\boldsymbol{\rho}$ as follows:

$$\mathrm{Pr}(\mathscr{C}_\mathbb{T})^2 \xrightarrow{(\times)} \mathrm{Pr}(\mathscr{C}_\mathbb{T}) \xrightarrow{\lfloor \text{🔒}_{\mathrm{st}} \rfloor_*} \mathrm{Sh}(\mathbb{S}) \qquad \boldsymbol{\rho}_* \tag{3}$$

Because $\lfloor \text{🔒}_{\mathrm{st}} \rfloor_*$ is continuous, we may commute it past the product functor:

$$\begin{array}{ccc} \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2 & \xrightarrow{\lfloor \text{🔒}_{\mathrm{st}} \rfloor_*^2} & \mathrm{Sh}(\mathbb{S})^2 \\ {\scriptstyle(\times)}\downarrow & \swarrow_{\boldsymbol{\rho}_*} & \downarrow{\scriptstyle(\times)} \\ \mathrm{Pr}(\mathscr{C}_\mathbb{T}) & \xrightarrow[\lfloor \text{🔒}_{\mathrm{st}} \rfloor_*]{} & \mathrm{Sh}(\mathbb{S}) \end{array} \tag{4}$$

Hence $\boldsymbol{\rho}_*$ takes a pair $(X_L, X_R) : \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2$ to $\lfloor \text{🔒}_{\mathrm{st}} \rfloor_* X_L \times \lfloor \text{🔒}_{\mathrm{st}} \rfloor_* X_R$. □

Construction 5.27 (Topos of parametricity structures). We then obtain a topos $\mathbf{X}$ 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:

$$\begin{array}{ccc} 2 \cdot \widehat{\mathscr{C}_\mathbb{T}} & \xrightarrow{\boldsymbol{\rho}} & \mathbb{S} \\ {\scriptstyle(\mathbf{id}, \bullet)}\downarrow & & \downarrow{\scriptstyle \boldsymbol{i}} \\ \left(2 \cdot \widehat{\mathscr{C}_\mathbb{T}}\right) \times \mathbb{S} & \dashrightarrow & \mathbf{X} \\ {\scriptstyle(\mathbf{id}, \circ)}\uparrow & \nearrow_{\boldsymbol{j}} & \\ 2 \cdot \widehat{\mathscr{C}_\mathbb{T}} & & \end{array}$$

The induced embedding $\boldsymbol{j} : 2 \cdot \widehat{\mathscr{C}_\mathbb{T}} \hookrightarrow \mathbf{X}$ can be seen to be an open immersion; moreover, its image is the open complement of the closed immersion. Therefore $\mathbf{X}$ is a topos governing parametricity structures, and restricting along the open immersion projects the (doubled) syntactic

part of a parametricity structure, whereas restricting along the closed immersion projects the (phase separated) semantic part of a parametricity structure.

Remark 5.28. The Sierpiński topos $\mathbb{S}$ plays two distinct roles in Construction 5.27: 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. In the first case, $\mathbb{S}$ is acting as a directed interval figure whereas in the second case, $\mathbb{S}$ is acting as the classifier of opens and $\boldsymbol{\rho}$ is the characteristic map for the open of $2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}$ that restricts on each side to the static open 🔓_{st}. 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 $\mathbb{1}$.

Computation 5.29. We may compute an explicit description of parametricity structures, *i.e.* sheaves on $\mathbf{X}$. A parametricity structure $X : \mathrm{Sh}(\mathbf{X})$ is given by the following data:

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

Proof. We recall the pushout of topoi that defines $\mathbf{X}$ from Construction 5.27.

$$\begin{array}{ccc} 2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}} & \xrightarrow{\boldsymbol{\rho}} & \mathbb{S} \\ {\scriptstyle (\mathbf{id}, \bullet)}\downarrow & & \downarrow{\scriptstyle \boldsymbol{i}} \\ \left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S} & \longrightarrow & \mathbf{X} \end{array} \tag{5}$$

We translate Diagram 5 into the language of logoi, at first only dualizing:

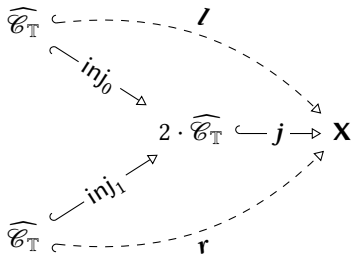
$$\begin{array}{ccc} \mathrm{Sh}(\mathbf{X}) & \longrightarrow & \mathrm{Sh}\left(\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \times \mathbb{S}\right) \\ {\scriptstyle \boldsymbol{i}^*}\downarrow & & \downarrow{\scriptstyle (\mathbf{id}, \bullet)^*} \\ \mathrm{Sh}(\mathbb{S}) & \xrightarrow[\boldsymbol{\rho}^*]{} & \mathrm{Sh}\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \end{array} \tag{6}$$

First we observe that the a sheaf on $\mathbf{Y} \times \mathbb{S}$ is the same as a family of sheaves on $\mathbf{Y}$, and that the closed point is the coordinate for the *domain* of such a family. Therefore we may rewrite the right-hand map of Diagram 6 as follows:

$$\begin{array}{ccc} \mathrm{Sh}(\mathbf{X}) & \longrightarrow & \mathrm{Sh}\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right)^{\to} \\ {\scriptstyle \boldsymbol{i}^*}\downarrow & & \downarrow{\scriptstyle \mathrm{dom}} \\ \mathrm{Sh}(\mathbb{S}) & \xrightarrow[\boldsymbol{\rho}^*]{} & \mathrm{Sh}\left(2 \cdot \widehat{\mathscr{C}_{\mathbb{T}}}\right) \end{array} \tag{7}$$

By Diagram 7, we see that a sheaf on $\mathbf{X}$ carries the data of a sheaf $X^\bullet$ on $\mathbb{S}$, a sheaf $X^\circ = (X^\circ_L, X^\circ_R)$ on $2 \cdot \widehat{\mathscr{C}_\mathbb{T}}$, and a morphism $\boldsymbol{\rho}^* X^\bullet \longrightarrow X^\circ : \mathrm{Sh}(2 \cdot \widehat{\mathscr{C}_\mathbb{T}})$. By adjoint transpose, this is the same a morphism $X^\bullet \longrightarrow \boldsymbol{\rho}_* X^\circ$ and by Computation 5.26, this is a morphism $X^\bullet \longrightarrow \lfloor \mathbb{\Omega}_{\mathsf{st}} \rfloor_* X^\circ_L \times \lfloor \mathbb{\Omega}_{\mathsf{st}} \rfloor_* X^\circ_R$. □

The open immersion $\boldsymbol{j} : 2 \cdot \widehat{\mathscr{C}_\mathbb{T}} \hookrightarrow \mathbf{X}$ corresponds (by definition) to an open $\mathbb{\Omega}_{\mathsf{syn}} \in \mathscr{O}_{\mathbf{X}}$, *i.e.* the subterminal parametricity structure $\mathbb{\Omega}_{\mathsf{syn}} = \big(\emptyset_{\mathrm{Sh}(\mathbb{S})} \longrightarrow \boldsymbol{\rho}_*(\mathbf{1}_{\mathrm{Pr}(\mathscr{C}_\mathbb{T})^2})\big)$. Let $\mathbf{Y}$ be a topos and E a finite cardinal; the injections $\mathsf{inj}_e : \mathbf{Y} \hookrightarrow E \cdot \mathbf{Y}$ into the coproduct are in fact open immersions [70, Lemma B.3.4.1]. Therefore we may reconstruct $\widehat{\mathscr{C}_\mathbb{T}}$ as two different open subtopoi of $\mathbf{X}$:



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

Working synthetically, we may use the modalities $\bigcirc_{\mathsf{syn/l}}, \bigcirc_{\mathsf{syn/r}}$ in the internal language of $\mathrm{Sh}(\mathbf{X})$ 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 $\bigcirc_{\mathsf{syn}}$ isolates the left and right parts of the syntax together, and its *closed complement* $\bullet_{\mathsf{syn}}$ is used to trivialize the syntactic parts and isolate the semantic part: in particular, we have $\bigcirc_{\mathsf{syn}}(\bullet_{\mathsf{syn}}(X)) = \mathbf{1}$. 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 et al. [114].

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

Lemma 5.30. *The logos of parametricity structures* $\mathrm{Sh}(\mathbf{X})$ *is a category of presheaves,* i.e. *there exists a category* $\mathscr{D}$ *such that* $\mathrm{Sh}(\mathbf{X}) \simeq \mathrm{Pr}(\mathscr{D})$.

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 $\mathrm{Pr}([1])$. Moreover, the direct image $\boldsymbol{\rho}_* : \mathrm{Pr}(\mathscr{C}_\mathbb{T})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ 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 [27]). □

Consequently, we may construct $\mathrm{Sh}(\mathbf{X})$ such that its internal dependent type theory contains a *strict* hierarchy of universes $\mathscr{U}_\alpha$ à la Hofmann and Streicher [66] and moreover enjoys the *strictification* axiom of Orton and Pitts [101], restated here as Axiom 3.9. 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 [65, 66, 133].

Theorem 5.31. *The category of sheaves* $\mathrm{Sh}(\mathbf{X})$ *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 [111] in the context of ModTT.

Corollary 5.32 (Generalized abstraction theorem). *Fix two families of signatures* $\sigma, \tau : \mathsf{Val}(\mathsf{type}) \to \mathsf{Sig}$*, and a closed module functor* $V : \mathsf{Val}\big(\prod_{x:\mathsf{type}} \prod_{_:\sigma(x)} \tau(x)\big)$*, together with a pair of closed module values* $U_i : \mathsf{Val}(\sigma(T_i))$ *for a pair of closed types* $T_0, T_1 : \mathsf{Val}(\mathsf{type})$*. Now, fix a family of* α*-small sets* $\tilde{T}$ *indexed in the closed values of type* $T_0 \times T_1$*; the interpretations of* σ, τ *induce a pair of families of phase separated sets* $[\![\sigma]\!](\tilde{T}), [\![\tau]\!](\tilde{T})$ *indexed in the closed values of* $\sigma(T_0) \times \sigma(T_1)$ *and* $\tau(T_0) \times \tau(T_1)$ *respectively. The* ***generalized abstraction theorem*** *states that we have a function of phase separated sets from* $[\![\sigma]\!](\tilde{T})[U_0, U_1]$ *to* $[\![\tau]\!](\tilde{T})[V(T_0, U_0), V(T_1, U_1)]$*, 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 $t : \mathcal{L} \vdash \langle\!|t|\!\rangle$ *sig* of the universe.

6.1 Relaxing the static–dynamic phase distinction

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 [48].

Allowing programs to have a non-trivial static component is also necessary to support *abstraction* in the presence of applicative functors like MkSet, as pointed out by Rossberg et al. [117]. Under the current strong static–dynamic phase distinction, abstraction for applicative functors can still be achieved by "tainting" every value declaration with an abstract type component, but there is reason to be skeptical this is *in fact* more desirable than simply achieving abstraction directly from general type dependency. In light of both Idris 2 and Lean 4 [26, 42], it would indeed be very hard to argue today that full-spectrum type dependency presents any unsurmountable problems for compilation of general-purpose programming languages.

Neither does it appear forced that module *commands* should be statically connected (except simply to reproduce the behavior of existing ML languages); the original difficulty inherent in the question of when two impure modules are identified by sharing seems to be already resolved by the modal separation of effects à la Moggi [96]. Decoupling static connectivity from computational effects significantly simplifies the theory of program modules. Future ML languages may expose *closed* modalities like $\bullet_{\mathrm{st}}$ to enable more flexible and fine-grained imposition of non-interference.

6.2 Let a hundred phase distinctions bloom!

Taking Reynolds's dictum[14] seriously, we believe that the phase distinction is the prototype for any number of *levels of abstraction*, each corresponding to a different open modality. The lax-modal separation of effects renders full type dependency quite unproblematic, hence some of the original motivations for the static–dynamic phase distinction may be weaker than previously thought. In contrast, the concept of *phase distinction* generally is more important than ever.

Example 6.1 (Logical relations). In this paper we considered the phase distinction between **syntactic** and **semantic**, which allows one to prove parametricity results as well as other important metatheorems such as canonicity and normalization [55, 125].

Example 6.2 (Type refinements). Type refinements à la Melliès and Zeilberger [88] can be interpreted by a phase distinction between **computational** and **logical**. Type refinements differ from the built-in verification capabilities of type theory in that logical/specification-level code is guaranteed to not interfere with computational-level code — even when the specification-level information is proof-relevant. The view of type refinements as a phase distinction is a compelling alternative to realizability-style accounts of program extraction [35]. Here, extraction is implemented *internally* by the weakening substitution $\Gamma, \unlock \longrightarrow \Gamma$.

Example 6.3 (Separate compilation). Modules with free variables ranging over their dependencies (referred to as *units* by Flatt and Felleisen [52], Swasey et al. [134]) are an attractive model of separate compilation: each module can be compiled independently of its dependencies, which are then linked by means of simultaneous substitution or *cut*. Separate compilation has the side effect, however, of limiting the ability of the compiler to generate more efficient code by inlining. Short of abandoning separate compilation entirely à la MLton [142], one may consider the suggestion of Stone [128, § 1.5.3] and Leroy [79] to use value-sharing (singletons) to expose definitions for inlining, but this has the destructive effect of breaking all abstraction boundaries imposed intentionally by the programmer. We suggest introducing a phase distinction between "compile-time" and "development-time", exposing inlineable definitions along compile-time extents $\{\sigma \mid \unlock_{\mathsf{cmpl}} \hookrightarrow V\}$.

Example 6.4 (Information flow). Information flow calculi à la Abadi et al. [1] can be interpreted by a phase distinction between **low** and **high security**. The open modality $\bigcirc_\ell$ projects the data that is visible to clients with security clearance ℓ; the closed modality $\bullet_\ell$ hides information from clients with clearance ℓ. Non-interference follows immediately from the laws of the closed modality.

6.3 Formalization of parametricity theorems

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 [101]. Actually formalizing the axioms of ParamTT in a proof assistant like Agda, Coq, or Lean is within reach, thanks to the work of Gilbert et al. [53].

6.4 Non-trivial computational effects

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.

[14] "Type structure is a syntactic discipline for enforcing levels of abstraction" [111].

ACKNOWLEDGMENTS

Thanks to Mathieu Anel, Carlo Angiuli, Steve Awodey, Karl Crary, Derek Dreyer, Daniel Gratzer, Guillaume Munch-Maccagnoni, 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

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, San Antonio, Texas, USA, 147–160. https://doi.org/10.1145/292540.292555

[2] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal parametric polymorphism. *Theoretical Computer Science* 121, 1 (1993), 9–58. https://doi.org/10.1016/0304-3975(93)90082-5

[3] Andreas Abel, Thierry Coquand, and Miguel Pagano. 2009. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance. In *Typed Lambda Calculi and Applications*, Pierre-Louis Curien (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–19.

[4] Stuart Frazier Allen. 1987. A Non-Type-Theoretic Definition of Martin-Löf's Types. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87)*. IEEE, Ithaca, NY, 215–221.

[5] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–199.

[6] Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

[7] Mathieu Anel and André Joyal. 2021. Topo-logie. In *New Spaces in Mathematics: Formal and Conceptual Reflections*, Mathieu Anel and Gabriel Catren (Eds.). Vol. 1. Cambridge University Press, Chapter 4, 155–257. https://doi.org/10.1017/9781108854429.007

[8] Carlo Angiuli. 2019. *Computational Semantics of Cartesian Cubical Type Theory.* Ph.D. Dissertation. Carnegie Mellon University.

[9] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing Representation Independence with Univalence. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021). https://doi.org/10.1145/3434293

[10] Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. 1972. *Théorie des topos et cohomologie étale des schémas.* Springer-Verlag, Berlin. 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.

[11] David Aspinall. 1995. Subtyping with singleton types. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

[12] Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Proceedings of the 21st EACSL Annual Conference / 26th International Workshop on Computer Science Logic (Leibniz International Proceedings in Informatics, Vol. 16)*. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 46–61. https://doi.org/10.4230/LIPIcs.CSL.2012.46

[13] Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A Relationally Parametric Model of Dependent Type Theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Diego, California, USA, 503–515. https://doi.org/10.1145/2535838.2535852

[14] Steve Awodey. 2014. Structuralism, Invariance, and Univalence. *Philosophia Mathematica* 22, 1 (2014), 1–11.

[15] Nick Benton, Martin Hofmann, and Vivek Nigam. 2013. Proof-Relevant Logical Relations for Name Generation. In *Typed Lambda Calculi and Applications*, Masahito Hasegawa (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 48–60.

[16] Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract Effects and Proof-Relevant Logical Relations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Diego, California, USA, 619–631. https://doi.org/10.1145/2535838.2535869

[17] Clemens Berger, Paul-André Melliès, and Mark Weber. 2012. Monads with arities and their associated theories. *Journal of Pure and Applied Algebra* 216, 8 (2012), 2029–2048. https://doi.org/10.1016/j.jpaa.2012.02.039 Special Issue devoted to the International Conference in Category Theory 'CT2010'.

[18] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electron. Notes Theor. Comput. Sci.* 319, C (Dec. 2015), 67–82. https://doi.org/10.1016/j.entcs.2015.12.006

[19] Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-Theory in Color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, Boston, Massachusetts, USA, 61–72. https://doi.org/10.1145/2500365.2500577

[20] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free: Parametricity for Dependent Types. *J. Funct. Program.* 22, 2 (March 2012), 107–152. https://doi.org/10.1017/S0956796812000056

[21] Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. IEEE Computer Society, New Orleans, Louisiana, 135–144. https://doi.org/10.1109/LICS.2012.25

[22] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. 1994. Signatures for a Network Stack: A Systems Application of Standard ML. In *Proceedings of the ACM Conference on LISP and Functional Programming*. ACM Press, Orlando, Florida, 55–64.

[23] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:17.

[24] Lars Birkedal and Rasmus E. Møgelberg. 2005. Categorical models for Abadi and Plotkin's logic for parametricity. *Mathematical Structures in Computer Science* 15, 4 (2005), 709–772. https://doi.org/10.1017/S0960129505004834

[25] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Stovring. 2011. 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*. IEEE Computer Society, Washington, DC, USA, 55–64.

[26] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. (2021). arXiv:2104.00480 [cs.PL] To appear in the proceedings of ECOOP 2021.

[27] Aurelio Carboni and Peter Johnstone. 1995. Connected limits, familial representability and Artin glueing. *Mathematical Structures in Computer Science* 5, 4 (1995), 441–459. https://doi.org/10.1017/S0960129500001183

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

[29] John Cartmell. 1978. *Generalised Algebraic Theories and Contextual Categories.* Ph.D. Dissertation. Oxford University.

[30] Simon Castellan, Pierre Clairambault, and Peter Dybjer. 2017. Undecidability of Equality in the Free Locally Cartesian Closed Category (Extended version). *Logical Methods in Computer Science* 13, 4 (2017).

[31] Evan Cavallo. 2021. *Higher Inductive Types and Internal Parametricity for Cubical Type Theory.* Ph.D. Dissertation. Carnegie Mellon University.

[32] Evan Cavallo and Robert Harper. 2020. Internal Parametricity for Cubical Type Theory. In *28th EACSL Annual Conference on Computer Science Logic (CSL 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 152)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:17. https://doi.org/10.4230/LIPIcs.CSL.2020.13

[33] Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. (2018). arXiv:1804.05236 https://arxiv.org/abs/1804.05236

[34] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: a constructive interpretation of the univalence axiom. *IfCoLog Journal of Logics and their Applications* 4, 10 (Nov. 2017), 3127–3169. http://www.collegepublications.co.uk/journals/ifcolog/?00019

[35] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[36] Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015 arXiv:1810.09367 [cs.PL] In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I.

[37] Thierry Coquand, Simon Huber, and Christian Sattler. 2019. Homotopy canonicity for cubical type theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[38] Karl Crary. 2017. Modules, Abstraction, and Parametric Polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, Paris, France, 100–113. https://doi.org/10.1145/3009837.3009892

[39] Karl Crary. 2019. Fully Abstract Module Compilation. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019). https://doi.org/10.1145/3290323

[40] Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30 (2020), e24. https://doi.org/10.1017/S0956796820000222 *Bob Harper Festschrift Collection.*

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

[42] Leonardo De Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language (System Description). (2021). To appear in the proceedings of the 28th International Conference on Automated Deduction.

[43] Radu Diaconescu. 1975. Change of base for toposes with generators. *Journal of Pure and Applied Algebra* 6, 3 (1975), 191–218. https://doi.org/10.1016/0022-4049(75)90015-8

[44] Derek Dreyer. 2005. *Understanding and Evolving the ML Module System.* Ph.D. Dissertation. Carnegie Mellon University, USA.

[45] Derek Dreyer. 2007. A Type System for Recursive Modules. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).* Association for Computing Machinery, Freiburg, Germany, 289–302. https://doi.org/10.1145/1291151.1291196

[46] Derek Dreyer, Karl Crary, and Robert Harper. 2003. A Type System for Higher-Order Modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03).* Association for Computing Machinery, New Orleans, Louisiana, USA, 236–249. https://doi.org/10.1145/604131.604151

[47] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular Type Classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, Nice, France, 63–70. https://doi.org/10.1145/1190216.1190229

[48] Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

[49] Matt Fairtlough and Michael Mendler. 1997. Propositional Lax Logic. *Information and Computation* 137, 1 (1997), 1–33. https://doi.org/10.1006/inco.1997.2627

[50] Marcelo Fiore. 2002. 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).* ACM, Pittsburgh, PA, USA, 26–37. https://doi.org/10.1145/571157.571161

[51] Marcelo P. Fiore and Giuseppe Rosolini. 1997. Two models of synthetic domain theory. *Journal of Pure and Applied Algebra* 116, 1 (1997), 151–162. https://doi.org/10.1016/S0022-4049(96)00164-8

[52] Matthew Flatt and Matthias Felleisen. 1998. Units: Cool Modules for HOT Languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation.* Association for Computing Machinery, Montreal, Quebec, Canada, 236–248. https://doi.org/10.1145/277650.277730

[53] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* (Jan. 2019), 1–28. https://doi.org/10.1145/329031610.1145/3290316

[54] Jean Goubault-Larrecq, Sławomir Lasota, and David Nowak. 2008. Logical relations for monadic types. *Mathematical Structures in Computer Science* 18, 6 (2008), 1169–1217. https://doi.org/10.1017/S0960129508007172

[55] Daniel Gratzer. 2021. Normalization for Multimodal Type Theory. (2021). https://jozefg.github.io/papers/normalization-for-multimodal-type-theory.pdf Unpublished manuscript.

[56] Daniel Gratzer and Jonathan Sterling. 2020. Syntactic categories for dependent type theory: sketching and adequacy. (2020). arXiv:2012.10783 [cs.LO]

[57] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a Modal Dependent Type Theory. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 107:1–107:29. https://doi.org/10.1145/3341711

[58] Robert Harper. 2016. *Practical Foundations for Programming Languages* (second ed.). Cambridge University Press, New York, NY, USA.

[59] Robert Harper. 2019. **PFPL** Supplement: How to (Re)Invent Tait's Method. (Nov. 2019). http://www.cs.cmu.edu/~rwh/pfpl/supplements/tait.pdf

[60] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143–184. https://doi.org/10.1145/138027.138060

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

[62] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Association for Computing Machinery, San Francisco, California, USA, 341–354. https://doi.org/10.1145/96709.96744

[63] Robert Harper and Christopher Stone. 2000. A Type-theoretic Interpretation of Standard ML. In *Proof, Language, and Interaction*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press, Cambridge, MA, USA, 341–387. http://dl.acm.org/citation.cfm?id=345868.345906

[64] Claudio Alberto Hermida. 1993. *Fibrations, Logical Predicates and Indeterminates.* Ph.D. Dissertation. University of Edinburgh. https://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-277/

[65] Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*. Cambridge University Press, 79–130.

[66] Martin Hofmann and Thomas Streicher. 1997. Lifting Grothendieck Universes. (1997). https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf Unpublished note.

[67] J. M. E. Hyland. 1988. A small complete category. *Annals of Pure and Applied Logic* 40, 2 (1988), 135–165. https://doi.org/10.1016/0168-0072(88)90018-8

[68] J. M. E. Hyland. 1991. First steps in synthetic domain theory. In *Category Theory*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–156.

[69] Peter Johnstone. 1977. *Topos Theory.* Academic Press.

[70] Peter T. Johnstone. 2002. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2.* Number 43 in Oxford Logical Guides. Oxford Science Publications.

[71] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for type theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[72] Saul A. Kripke. 1965. Semantical Analysis of Intuitionistic Logic I. In *Formal Systems and Recursive Functions*, J. N. Crossley and M. A. E. Dummett (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 40. Elsevier, 92–130. https://doi.org/10.1016/S0049-237X(08)71685-9

[73] Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Computer Science Logic 2013 (CSL 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 432–451. https://doi.org/10.4230/LIPIcs.CSL.2013.432

[74] F. William Lawvere. 1963. *Functorial Semantics of Algebraic Theories.* Ph.D. Dissertation. Columbia University.

[75] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Nice, France, 173–184. https://doi.org/10.1145/1190216.1190245

[76] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Portland, Oregon, USA, 109–122. https://doi.org/10.1145/174675.176926

[77] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Francisco, California, USA, 142–153. https://doi.org/10.1145/199448.199476

[78] Xavier Leroy. 1996. A syntactic theory of type generativity and sharing. *Journal of Functional Programming* 6, 5 (1996), 667–698. https://doi.org/10.1017/S0956796800001933

[79] Xavier Leroy. 2000. A Modular Module System. *Journal of Functional Programming* 10, 3 (May 2000), 269–303. https://doi.org/10.1017/S0956796800003683

[80] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system manual.

[81] Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2).* Kluwer Academic Publishers, Norwell, MA, USA.

[82] Saunders Mac Lane and Ieke Moerdijk. 1992. *Sheaves in geometry and logic: a first introduction to topos theory.* Springer, New York.

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

[84] David B. MacQueen. 1986. Using Dependent Types to Express Modular Structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, St. Petersburg Beach, Florida, 277–286. https://doi.org/10.1145/512644.512670

[85] Per Martin-Löf. 2013. 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.

[86] James McKinna and Rod Burstall. 1993. Deliverables: a categorical approach to program development in type theory. In *Mathematical Foundations of Computer Science 1993*, Andrzej M. Borzyszkowski and Stefan Sokołowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–67.

[87] J. C. C. McKinsey and Alfred Tarski. 1946. On Closed Elements in Closure Algebras. *Annals of Mathematics* 47, 1 (1946), 122–162.

[88] Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Mumbai, India.

https://hal.inria.fr/hal-01096910
[89] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.
[90] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.
[91] J. C. Mitchell and R. Harper. 1988. The Essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Diego, California, USA, 28–46.
[92] John C. Mitchell and Gordon D. Plotkin. 1985. Abstract Types Have Existential Types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, New Orleans, Louisiana, USA, 37–51. https://doi.org/10.1145/318593.318606
[93] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065
[94] R. E. Møgelberg and A. Simpson. 2007. Relational Parametricity for Computational Effects. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 346–355.
[95] Eugenio Moggi. 1989. A Category-Theoretic Account of Program Modules. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 101–117.
[96] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.
[97] Bengt Nordström, Kent Peterson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory*. International Series of Monographs on Computer Science, Vol. 7. Oxford University Press, NY.
[98] Andreas Nuyts. 2018. Presheaf Models of Relational Modalities in Dependent Type Theory. (2018). arXiv:1805.08684 https://arxiv.org/abs/1805.08684
[99] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017). https://doi.org/10.1145/3110276
[100] Frank J. Oles. 1986. *Type Algebras, Functor Categories and Block Structure*. Cambridge University Press, USA, 543–573.
[101] Ian Orton and Andrew M. Pitts. 2016. 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*. 24:1–24:19.
[102] Marco Paviotti. 2016. *Denotational semantics in Synthetic Guarded Domain Theory*. Ph.D. Dissertation. Denmark.
[103] Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proc. ACM Program. Lang.* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371126
[104] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. *Proc. ACM Program. Lang.* 3, ICFP (July 2019). https://doi.org/10.1145/3341712
[105] Leaf Eames Petersen. 2005. *Certifying compilation for Standard ML in a type analysis framework*. Ph.D. Dissertation. Carnegie Mellon University.
[106] Andrew M. Pitts. 1987. Polymorphism is set theoretic, constructively. In *Category Theory and Computer Science*, David H. Pitt, Axel Poigné, and David E. Rydeheard (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 12–39.
[107] Gordon Plotkin and Martín Abadi. 1993. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 361–375.
[108] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*. Springer-Verlag, Berlin, Heidelberg, 342–356.
[109] The RedPRL Development Team. 2018. redtt. https://www.github.com/RedPRL/redtt
[110] The RedPRL Development Team. 2020. cooltt. https://www.github.com/RedPRL/cooltt
[111] John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*.
[112] John C. Reynolds. 1995. Using Functor Categories to Generate Intermediate Code. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Francisco, California, USA, 25–36. https://doi.org/10.1145/199448.199452
[113] Emily Riehl and Michael Shulman. 2017. A type theory for synthetic ∞-categories. *Higher Structures* 1 (2017).
[114] Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). https://doi.org/10.23638/LMCS-16(1:2)2020 arXiv:1706.07526 [math.CT]
[115] E. Robinson and G. Rosolini. 1994. Reflexive graphs and parametric polymorphism. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 364–371. https://doi.org/10.1109/LICS.1994.316053
[116] Andreas Rossberg. 2018. 1ML – Core and modules united. *Journal of Functional Programming* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205
[117] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

[118] Zhong Shao. 1999. Transparent Modules with Fully Syntactic Signatures. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, Paris, France, 220–232. https://doi.org/10.1145/317636.317801

[119] Michael Shulman. 2013. Scones, Logical Relations, and Parametricity. https://golem.ph.utexas.edu/category/2013/04/scones_logical_relations_and_p.html Blog post.

[120] Michael Shulman. 2015. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science* 25, 5 (2015), 1203–1277. https://doi.org/10.1017/S0960129514000565

[121] Kristina Sojakova and Patricia Johann. 2018. A General Framework for Relational Parametricity. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Oxford, United Kingdom, 869–878. https://doi.org/10.1145/3209108.3209141

[122] The Stacks Project Authors. 2017. Stacks Project. (2017). http://stacks.math.columbia.edu

[123] Sam Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 519–519.

[124] Jonathan Sterling and Carlo Angiuli. 2020. Gluing models of type theory along flat functors. (2020). Unpublished draft.

[125] Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, New York, NY, USA. arXiv:2101.11479 [cs.LO] To appear.

[126] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 31:1–31:25. https://doi.org/10.4230/LIPIcs.FSCD.2019.31 arXiv:1904.08562 [cs.LO]

[127] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2020. A cubical language for Bishop sets. (2020). arXiv:2003.01491 [cs.LO] Under review.

[128] Christopher Allen Stone. 2000. *Singleton Kinds and Singleton Types*. Ph.D. Dissertation. Carnegie Mellon University.

[129] Christopher A. Stone and Robert Harper. 2000. Deciding Type Equivalence in a Language with Singleton Kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Boston, MA, USA, 214–227. https://doi.org/10.1145/325694.325724

[130] Christopher A. Stone and Robert Harper. 2006. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic* (2006).

[131] Ross Street. 1972. The formal theory of monads. *Journal of Pure and Applied Algebra* 2, 2 (1972), 149–168. https://doi.org/10.1016/0022-4049(72)90019-9

[132] Thomas Streicher. 1991. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhauser Boston Inc., Cambridge, MA, USA.

[133] Thomas Streicher. 2005. Universes in toposes. In *From Sets and Types to Topology and Analysis: Towards practical foundations for constructive mathematics*, Laura Crosilla and Peter Schuster (Eds.). Oxford Logical Guides, Vol. 48. Oxford University Press, Oxford, 78–90. https://doi.org/10.1093/acprof:oso/9780198566519.001.0001

[134] David Swasey, Tom Murphy, Karl Crary, and Robert Harper. 2006. A Separate Compilation Extension to Standard ML. In *Proceedings of the 2006 Workshop on ML (ML '06)*. Association for Computing Machinery, Portland, Oregon, USA, 32–42. https://doi.org/10.1145/1159876.1159883

[135] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.* 2, ICFP (July 2018). https://doi.org/10.1145/3236787

[136] W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

[137] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[138] Steven Vickers. 2007. *Locales and Toposes as Spaces*. Springer Netherlands, Dordrecht, 429–496. https://doi.org/10.1007/978-1-4020-5587-4_8

[139] Vladimir Voevodsky. 2006. A very short note on homotopy λ-calculus. *Unpublished* (Sept. 2006). http://www.math.ias.edu/vladimir/files/2006_09_Hlambda.pdf

[140] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. Association for Computing Machinery, Imperial College, London, United Kingdom, 347–359. https://doi.org/10.1145/99370.99404

[141] Philip Wadler. 2007. The Girard–Reynolds isomorphism (second edition). *Theoretical Computer Science* 375, 1 (2007), 201–226. Festschrift for John C. Reynolds's 70th birthday.

[142] Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML '06)*. Association for Computing Machinery, Portland, Oregon, USA, 1. https://doi.org/10.1145/1159876.1159877

[143] Felix Wellen. 2017. *Formalizing Cartan Geometry in Modal Homotopy Type theory*. Ph.D. Dissertation. Karlsruhe Institute of Technology.

[144] G. C. Wraith. 1975. Lectures on elementary topoi. In *Model Theory and Topoi*, F. William Lawvere, Christian Maurer, and Gavin C. Wraith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–206.