# Logical Relations as Types

Proof-Relevant Parametricity for Program Modules

JONATHAN STERLING, Carnegie Mellon University
ROBERT HARPER, Carnegie Mellon University

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.

The synthetic account of program modules is backed by a proof-relevant generalization of Reynolds's relational theory of parametricity, dubbed *parametricity structures*, to obtain a representation-independence theorem for abstract types. 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 can always accommodate 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.

Additional Key Words and Phrases: modules, phase distinction, computational effects, modal type theory, gluing, logical relations

## 1 INTRODUCTION

Program modules are the application of dependent type theory with universes to the large-scale structuring of programs. As MacQueen [1986] observed, the hierarchical structuring of programs is an instance of dependent sum; consider the example of a type together with a pretty printer:

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

Authors' addresses: Jonathan Sterling, Carnegie Mellon University, Computer Science Department, jmsterli@cs.cmu.edu; Robert Harper, Carnegie Mellon University, Computer Science Department, rwh@cs.cmu.edu.


2475-1421/2018/1-ART1 $15.00
https://doi.org/

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"* [Reynolds 1983]; 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:[1]

```
let DbLvl : EQ = struct type t = int  val eq = int_eq end in
let DbIdx : EQ = struct type t = int  val eq = int_eq end in
body  (* in this context, DbLvl.t ≠ DbIdx.t *)
```

Of course, the example above is judgmentally equal to `let DbLvl : EQ = ... in let DbIdx = DbLvl in ...`, but the well-typedness of body ensures that this coincidence is not exploited.

*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
```

[1] In our example, we are using the ordinary type-theoretic meaning of the annotation `... : EQ`, whereas in Standard ML the colon is used to refer to a more complex *transparent* ascription. In Standard ML, information-hiding can be achieved via the opaque ascription `... :> EQ`, which has an additional effect of rendering the module *impure* and therefore not projectible; our account differs in that we do *not* conflate (purity, transparency, projectibility).

```
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.

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.t` ≠ `NS2.t` 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* [Dreyer et al. 2003; Harper 2016], 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 $\{-\}$; concurrent work of Crary supports the same conclusion [Crary 2020]. ModTT distinguishes between computations $M \div \sigma$ and values $V : \sigma$, and mediates between them using the standard rules of the lax modality [Fairtlough and Mendler 1997]:

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

In this style, one no longer needs the notion of projectibility: a generative functor is nothing more than a module-level function $\sigma \Rightarrow \{\tau\}$, 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. [1990]; Moggi [1989] 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:

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

*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 $\mathcal{B}$ whose objects are kinds and whose morphisms are constructors.
(2) Next one defines an *indexed category* $C : \mathcal{B}^{\mathrm{op}} \longrightarrow \mathbf{Cat}$: for a kind $k$, the fiber category $C(k)$ 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* $\mathcal{G} = \int_{\mathcal{B}} C$, which takes an indexed category to its total category. An object of $\mathcal{G}$ is a pair $(k, \sigma)$ with $k : \mathcal{B}$ and $\sigma : C(k)$; a morphism $(k, \sigma) \longrightarrow (k', \sigma')$ is a morphism $c : k \longrightarrow k' : \mathcal{B}$ together with a morphism $\sigma \longrightarrow c^*\sigma' : C(k)$.

The benefit of considering $\mathcal{G}$ 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. [1990] 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. [2003].

*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 $\mathcal{G}$ and the corresponding fibration $\mathcal{G} \longrightarrow\!\!\!\!\triangleright\ \mathcal{B}$ that projects the static language from the full language. We obtain further leverage by reconstructing $\mathcal{B}$ as a slice $\mathcal{G}_{/\text{🔓}_{\mathrm{st}}}$ for a special object $\text{🔓}_{\mathrm{st}} : \mathcal{G}$.

The view of $\mathcal{B}$ as a slice of $\mathcal{G}$ is inspired by *Artin gluing* [Artin et al. 1972], 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 $\mathcal{G}$ possesses a new context-former $(\Gamma, \text{🔓}_{\text{St}})$ called the "static open";[2] when $\text{🔓}_{\text{St}}$ is in the context, only the static part of a given object has force. For instance, module computations and terms of program type are rendered purely dynamic / statically inert by means of special rules of *static connectivity* under the assumption of $\text{🔓}_{\text{St}}$:

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

$$\textsc{static connectivity (3)}\ \frac{\Gamma \vdash \sigma\ sig \qquad \Gamma \in \text{🔓}_{\text{St}}}{\Gamma \vdash * \div \sigma} \qquad \textsc{static connectivity (3)}\ \frac{\Gamma \vdash \sigma\ sig \qquad \Gamma \vdash M \div \sigma \qquad \Gamma \ni \text{🔓}_{\text{St}}}{\Gamma \vdash M \equiv * \div \sigma}$$

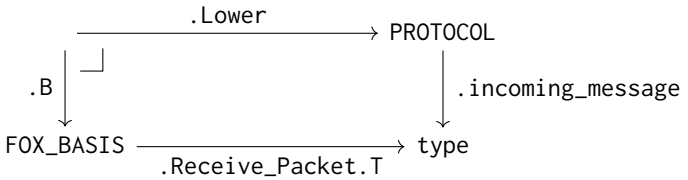
*Signatures, kinds, and static equivalence.* In our account, the phase distinction between signatures/modules and kinds/constructors is expressed by a universal property: a signature $\Gamma \vdash \sigma\ sig$ is called a kind iff the weakening of sets of equivalence classes from $\{[V] \mid \Gamma \vdash V : \sigma\}$ to $\{[V] \mid \Gamma, \text{🔓}_{\text{St}} \vdash V : \sigma\}$ is an isomorphism natural in $\Gamma$. In other words, the exponentiation by $\text{🔓}_{\text{St}}$ defines an *open modality* $\mathrm{St} = (\text{🔓}_{\text{St}} \Rightarrow -)$ in the sense of topos theory, and a kind is nothing more than an St-modal signature. The notion of static equivalence from Dreyer et al. [2003] is then reconstructed as ordinary judgmental equality in the context of $\text{🔓}_{\text{St}}$; the view of phase separation as a projection functor from Moggi [1989] is reconstructed by the weakening $\mathcal{G} \longrightarrow \mathcal{G}_{/\text{🔓}_{\text{St}}}$.

### 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 [Biagioni et al. 1994] 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 [1988], is perfectly appropriate from a semantic perspective; however, it unfortunately renders type checking undecidable [Castellan et al. 2017]. Because types in ML-style languages are meant to provide *lightweight* verification, it is essential that the type checking problem be tractable: therefore,

[2] 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.

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 [2000]. 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 disrupt the decidability of type equivalence [Abel et al. 2009; Stone and Harper 2006].

The truly difficult part of singleton types, dealt with by Stone and Harper [2006], 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 [Crary 2019; Dreyer et al. 2003].

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 $\eta$-laws of each connective, *including* the $\eta$-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 [RedPRL Development Team 2020].[3]

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

One of the design constraints for module systems, embodied in the phase distinction, is that dependency should only involve *static* constructs; the decidable fragment of the *dynamic* algebra of programs is unfortunately too fine to act as more than an obstruction to the composition of program components. From our synthetic view of the phase distinction, it is most natural to rather generalize the type singletons to a signature connective $\{\sigma \mid \text{🔒}_{\mathsf{st}} \to V\}$ that classifies the "static extent" of a module $V : \sigma$ for an arbitrary signature $\sigma$, summarized in the following rules of inference:[4]

FORMATION

$$\frac{\Gamma \vdash \sigma\ \mathit{sig} \qquad \Gamma, \text{🔒}_{\mathsf{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔒}_{\mathsf{st}} \to V\}\ \mathit{sig}}$$

INTRODUCTION

$$\frac{\Gamma \vdash U : \sigma \qquad \Gamma, \text{🔒}_{\mathsf{st}} \vdash U \equiv V : \sigma}{\Gamma \vdash U : \{\sigma \mid \text{🔒}_{\mathsf{st}} \to V\}}$$

ELIMINATION

$$\frac{\Gamma \vdash U : \{\sigma \mid \text{🔒}_{\mathsf{st}} \to V\}}{\Gamma \vdash U : \sigma \qquad \Gamma, \text{🔒}_{\mathsf{st}} \vdash U \equiv V : \sigma}$$

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 $\{\sigma \mid \text{🔒}_{\mathsf{st}} \to V\}$ is not a singleton in general, but it is a singleton when $\sigma$ 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 [2017], already available in a few implementations of cubical type theory [RedPRL Development Team 2018, 2020]. Whereas in cubical type theory one extends along

[3] 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.

[4] 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.

a cofibrant subobject $\phi \rightarrowtail \mathbb{I}^n$ of a cube, in a phase separated module calculus one extends along the open domain $\blacksquare_{\mathsf{st}} \rightarrowtail \mathbf{1}$. The static extent connective is also closely related to the *formal disk bundle* of Wellen [2017], 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 [Milner et al. 1990], in which each structure is in essence tagged with a static identity [MacQueen et al. 2020]; 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 [Milner et al. 1997]. Our *static extents* capture exactly the semantics of weak structure sharing; we note that the strong sharing of SML '90 can be simulated by adding a dummy abstract type to each signature during elaboration.

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

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

*1.4.1 Algebraic metatheory in an equational logical framework.* Many existing calculi for program modules are formulated using raw terms, and animated via a mixture of judgmental equality (for the module layer) and structural operational semantics (for the program layer). In contrast, we formulate ModTT entirely in an equational logical framework,[5] 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 [Staton 2013].

The mechanization of Standard ML [Crary and Harper 2009; Lee et al. 2007] in the Edinburgh Logical Framework [Harper et al. 1993] 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, we employ a semantic logical framework due to Uemura [2019] to account for both typing and judgmental equality of abstract terms. The idea of dependently typed equational logical frameworks goes back to Cartmell [1978] (for theories without binding), and was further developed by Martin-Löf for theories with binding of arbitrary order [Nordström et al. 1990]. 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 [Altenkirch and Kaposi 2016; Coquand 2019; Coquand et al. 2019; Kaposi et al. 2019; Sterling and Angiuli 2020; Sterling et al. 2019, 2020].

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 more or less directly 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 [1991] 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 internal 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 external (surface) language, but the external language is almost never expected to be complete for the internal language, or even to have meaning independently of its elaboration into the internal language. The fulfillment of any such expectation is immediately obstructed by the myriad non-compositional aspects of the elaboration

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

of external 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* [Tait 1967], 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. For instance, let $R_{\mathsf{bool}} \subseteq \mathsf{ClosedTerms(bool)}$ be the property of being either #t or #f; one shows that $R_{\mathsf{bool}}$ 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 [Artin et al. 1972], Artin gluing can be viewed as a tool to *stitch together* a type theory's syntactic category with a category of semantic things, leading to a category of "families of semantic things indexed in syntactic things". Logical relations are then the proof-irrelevant special case of gluing, where families are restricted to have subsingleton fibers.

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

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

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

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 $\Gamma$ is a family of sets $\tilde{\Gamma}$ indexed in *pairs* of closing substitutions for $\Gamma$. A traditional 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 in a *subequational* way on raw terms, 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. [1995]; Fiore [2002] 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 [2013, 2015] and Coquand [2019], is the computability interpretation of *universes*. A universe is a special type $\mathcal{U}$ whose elements $A : \mathcal{U}$ 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 $A : \mathcal{U}$. This would seem to require a "relation of relations", but there can be no such thing: the fibers of relations are subsingleton.

In the past, type theorists have accounted for the logical relations of universes by parameterizing the construction in the graph of an assignment of logical relations to type codes [Allen 1987; Harper 1992], 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".

*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 [Artin et al. 1972; Johnstone 2002]: 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 3.2.1).

Following the methodology pioneered (in another context) by Orton and Pitts [2016], 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 $\mathsf{TT}_{\mathsf{PS}}$ in which every type can be thought of as a parametricity structure.[6] To substantiate the view of *logical relations as types* we extend $\mathsf{TT}_{\mathsf{PS}}$ with the following constructs:

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

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

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

| | | | | |
|---|---|---|---|---|
| **Propositions:** | $\text{🔓}_{\text{syn/l}}, \text{🔓}_{\text{syn/r}}, \text{🔓}_{\text{syn}}, \text{🔓}_{\text{st}} : \text{Prop}$ | $\text{🔓}_{\text{syn}} = \text{🔓}_{\text{syn/l}} \vee \text{🔓}_{\text{syn/r}}$ | $\text{🔓}_{\text{syn/l}} \wedge \text{🔓}_{\text{syn/r}} = \bot$ | |
| **Open modalities:** | $\text{Syn}, \text{St}, : \mathcal{U} \Rightarrow \mathcal{U}$ | $\text{Syn}(A) = \text{🔓}_{\text{syn}} \Rightarrow A$ | $\text{St}(A) = \text{🔓}_{\text{st}} \Rightarrow A$ | |
| **Closed modalities:** | $\text{Sem}, \text{Dyn} : \mathcal{U} \Rightarrow \mathcal{U}$ | $\text{Sem}(A) = A \star \text{🔓}_{\text{syn}}$ | $\text{Dyn}(A) = A \star \text{🔓}_{\text{st}}$ | |

Fig. 2. A summary of the structure available in the internal language of a topos of *synthetic phase separated parametricity structures*. Above, $A \star B$ is the *join* construction, obtained as the pushout $A \sqcup_{A \times B} B$. We consequently have $\text{Syn}(\text{Sem}(A)) \cong \mathbf{1}$ and $\text{St}(\text{Dyn}(A)) = \mathbf{1}$.

simply note that the "semantic modality" Sem is the universal way to trivialize the syntactic part of a parametricity structure, and the "dynamic modality" Dyn is the universal way to trivialize the static part of a parametricity structure.

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

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 [Rossberg 2018], which, as here, uses a universe to represent a signature of "small" types, which classify run-time values. Although ModTT does not have first class modules, there is no obstacle to supporting the packaging of modules of small signature into a type. 1ML also features a module connective analogous to the static extent, though the universal property of this connective is not explicated — in fact, the rules of equality of *modules themselves* are not even stated in Rossberg [2018]; Rossberg et al. [2014]. Consequently, the most substantial difference between ModTT and 1ML (aside from the lack of an abstraction theorem) is that the latter is *defined* by its translation into System Fω, whereas ModTT is given intrinsically as an algebraic theory that expresses equality of modules, with a modality to confine attention to their static parts. To be sure, it is elegant and practical to consider the compilation of modules by a phase-separating translation, as was done for example by Petersen [2005]. Nevertheless, it is also important to give a direct type-theoretic account of program modules *as they are to be used and reasoned about.*

*1.5.2 Modules, Abstraction, and Parametric Polymorphism.* In a pair of recent papers [Crary 2017, 2019], 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 [2020] joins us in advocating that module projectibility be reconstructed in terms of a lax modality.

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

*1.5.3 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 [Leroy 1995], later used in OCaml's module system [Leroy et al. 2020]: whereas generative functors can be thought of as functions $\sigma \Rightarrow \{\tau\}$, applicative functors correspond roughly to $\{\sigma \Rightarrow \tau\}$ as noted by Shao [1999], but subtleties abound. The subtleties of applicative and generative functor semantics (studied by Dreyer et al. [2003] 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 logical relations.* We are not the first to consider proof-relevant versions of parametricity; Sojakova and Johann [2018] define a general framework for parametric models of System F, which can be instantiated to give rise to a proof-relevant version of parametricity. Benton et al. [2013, 2014] use proof-relevant logical relations to work around the fact that logical relations involving an existential quantifier are rarely *admissible*, a problem also faced by [Crary 2017]. 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 Computational effects and the Fire Triangle.* Lax modalities do not interact cleanly with dependent type structure, unlike the idempotent lex and open modalities of Rijke et al. [2017]. A promising approach to the integration of real (non-idempotent) effects into dependent type theory is represented by the $\partial$CBPV calculus of Pédrot and Tabareau [2019], a dependently typed version of Levy's Call-By-Push-Value [Levy 2004] 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 $\partial$CBPV as an improved account of effects in dependent type theory; $\partial$CBPV's design is motivated by deep syntactical and operational concerns, and we hope in future work to reconcile these with our admittedly category theoretic and semantical viewpoint.

*1.5.6 Doubling the syntax.* In Section 5 we consider the copower $2 \cdot \widehat{\mathbb{T}}$ of a topos $\widehat{\mathbb{T}}$ representing the syntax of ModTT; this "doubled topos" serves as a suitable index to a gluing construction, yielding a topos $\mathcal{X} = \left(\left(2 \cdot \widehat{\mathbb{T}}\right) \times \mathbb{S}\right) \sqcup_{2 \cdot \widehat{\mathbb{T}}} \mathbb{S}$ 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 [2007]. This same observation lies at the heart of our convenient Notation 3.1 for working synthetically with the left- and right-hand sides of parametricity structures.

*1.5.7 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. [2012]; Pédrot et al. [2019]; Tabareau et al. [2018], also taken up by Per Martin-Löf in his Ernest Nagel Lecture in 2013 [Martin-Löf 2013]. The essential difference is that the parametricity translations are *analytic*, explicitly transforming types into (proof-relevant) logical relations, whereas our theory of parametricity structures is *synthetic*: we assume that everything in sight is a logical relation, and then identify the ones that are degenerate in either the syntactic or semantic direction via a modality.

## 2 ModTT: A TYPE THEORY FOR PROGRAM MODULES

We introduce ModTT, a type-theoretic internal language for program modules based on the considerations discussed in Section 1. We first give an informal presentation of the language using familiar notations in Section 2.1; in Section 2.2, we discuss the formal definition of ModTT in Uemura's logical framework [Uemura 2019].

### 2.1 Informal presentation of ModTT

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

*2.1.2 Types and dynamic modules.* The simplest module signature is 'Type', the signature classifying the *object-level* types of the programming language, like bool or $s \rightharpoonup t$. Given a module $t : \mathsf{Type}$, 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 Computations via lax modality.* To reconstruct generativity (Section 2.1.4) in a type theoretic way, we employ a modal separation of effects and distinguish computations of modules from values. This is achieved by means of a *strong monad*, presented judgmentally as a lax modality $\{-\}$ mediating between the $\boxed{\Gamma \vdash V : \sigma}$ and $\boxed{\Gamma \vdash M \div \sigma}$ judgments.[7]

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

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

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

display only the formation rules for reason of space:

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

Generative functors are defined as a mode of use of the dependent product combined with the lax modality, taking $((X : \sigma) \Rightarrow^{\mathrm{gen}} \sigma') := (X : \sigma) \Rightarrow \{\sigma'\}$ as in Crary [2020].

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

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

Remark 2.1. *The notation is suggestive of the accounts of modal type theory based on dependent right adjoints [Clouston et al. 2018]; indeed, the context extension $(-, \text{🔓}_{\mathrm{st}})$ 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* notion of judgmental equality in which the dynamic parts of different objects are identified when $\text{🔓}_{\mathrm{st}} \in \Gamma$. Specifically, we add rules to ensure that programs of a type as well as computations of modules are *statically connected* in the sense of having exactly one element under $\text{🔓}_{\mathrm{st}}$, as in Section 1.2.3.

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

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

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

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

The static extent reconstructs both type sharing and weak structure sharing, which appear in SML '97 [Milner et al. 1997] and OCaml [Leroy et al. 2020].

*Example 2.2.* The SML module signature `(SHOW where type t = bool)` is rendered in terms of the static extent as $\{\mathsf{SHOW} \mid \blacksquare^{\mathsf{st}}_{\mathsf{st}} \to [\mathsf{bool}, *]\}$:

$$\dfrac{\dfrac{}{\Gamma, \blacksquare^{\mathsf{st}}_{\mathsf{st}} \vdash \mathsf{bool} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\Gamma, \blacksquare^{\mathsf{st}}_{\mathsf{st}} \ni \blacksquare^{\mathsf{st}}_{\mathsf{st}}}}{\Gamma, \blacksquare^{\mathsf{st}}_{\mathsf{st}} \vdash * : \langle\!|\mathsf{bool} \rightharpoonup \mathtt{string}|\!\rangle}}{\dfrac{\Gamma, \blacksquare^{\mathsf{st}}_{\mathsf{st}} \vdash [\mathsf{bool}, *] : \mathsf{SHOW}}{\Gamma \vdash \{\mathsf{SHOW} \mid \blacksquare^{\mathsf{st}}_{\mathsf{st}} \to [\mathsf{bool}, *]\}\ \mathit{sig}}}$$

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 depenencies 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 lack of space, 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 $\langle\!|s \rightharpoonup t|\!\rangle \cong \langle\!|s|\!\rangle \Rightarrow \{\langle\!|t|\!\rangle\}$.

*2.1.8 External language and elaboration.* We do not present here an *external*/surface language; such a language would include many features not present in the internal 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. [2007] are re-cast as an elaboration strategy guided by $\eta$-laws, as in the elaboration of extension types in the `cooltt` proof assistant [RedPRL Development Team 2020]. The status of subtyping and retyping in ModTT is a significant divergence from previous work, which treated them within the internal language (an untenable position for an *algebraic* account of modules).

## 2.2 Algebraic presentation in a logical framework

Uemura has defined a dependently sorted equational logical framework with support for one level of variable binding, which may be used to define almost any kind of type theory whose contexts enjoy all the structural rules. We defer to Uemura [2019] for a full explication of the details, but we may briefly summarize Uemura's LF as follows:

(1) There is a universe $\square$ of *judgments*, and a subuniverse $*$ of *representable judgments*. A judgment is called representable if it can be assumed in an object-level context; therefore, we have $* \subseteq \square$ but not the other way around. For example, the judgment $A\ \mathit{type}$ is not usually representable, but $a : A$ usually is representable.
(2) The judgments are closed under dependent products (hypothetical judgments) whose base is representable: so if $X : *$ and $Y : X \Rightarrow \square$, then $(x : X) \Rightarrow Y(x) : \square$. Both $\square, *$ are closed under arbitrary dependent sum.[8]
(3) The judgments are closed under extensional equality: $(a =_X b)$ is a judgment for each $x, y : X$.

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

$$\frac{}{\text{🔓}_{\text{st}} : *} \qquad \frac{x, y : \text{🔓}_{\text{st}}}{\mathbf{irr}(x, y) : x =_{\text{🔓}_{\text{st}}} y} \qquad \frac{}{\mathbf{Sig} : \square} \qquad \frac{\sigma : \mathbf{Sig}}{\mathbf{Val}(\sigma) : *} \qquad \frac{\sigma : \mathbf{Sig} \qquad V : \text{🔓}_{\text{st}} \Rightarrow \mathbf{Val}(\sigma)}{\{\sigma \mid \text{🔓}_{\text{st}} \to V\} : \mathbf{Sig}}$$

$$\frac{\sigma : \mathbf{Sig} \qquad V : \text{🔓}_{\text{st}} \Rightarrow \mathbf{Val}(\sigma)}{\mathbf{Ext/iso}(\sigma, V) : \mathbf{Val}(\mathbf{Ext}(\sigma, V)) \cong (U : \mathbf{Val}(\sigma)) \times ((z : \text{🔓}_{\text{syn}}) \Rightarrow U =_{\mathbf{Val}(\sigma)} V(z))}$$

Fig. 3. A fragment of the fully explicit presentation of ModTT in Uemura's logical framework, written in the notation of local inference rules for readability. The introduction, elimination, computation, and uniqueness rules of the static extent are captured in a *single* rule declaring an isomorphism; declarations of this form are a definitional extension of Uemura's LF, because they always boil down to four elementary declarations.

A *signature* in Uemura's LF is given by a dependent telescope of well-typed declarations $\alpha : [\Gamma \rhd \mathfrak{s}]$ where $\mathfrak{s}$ is either $\square$ or $*$ or $X$ such that $X : \square$; the parameters $\Gamma$ are formed as dependent telescopes $\overline{x : X}$. A type theory may then be presented as a signature in the LF; for example, we present a fragment of ModTT in Figure 3 using inference rule notation (with parameters as premises).

*Syntactic category of an LF signature.* A signature $\Sigma$ in the LF *presents* a certain category $\mathbb{T}_\Sigma$ 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. A *model* of $\Sigma$ in a category $C$ equipped with the requisite structure can be viewed in two ways, in the spirit of Lawvere's functorial semantics [Lawvere 1963]:

(1) A model of $\Sigma$ is an algebra for $\Sigma$ internally to $C$, i.e. an *implementation* of all the LF signature clauses of $\Sigma$ in the internal extensional type theory of $C$.
(2) A model of $\Sigma$ is a structure preserving functor $\mathbb{T}_\Sigma \longrightarrow C$.

The induction principle or *universal property* of the syntax states that $\mathbb{T}_\Sigma$ is the *least* model of $\Sigma$; this universal property is the main ingreidient for proving syntactic metatheorems by semantic means, as we advocate and apply in this paper.

Notation 2.3. *We will write $\mathbb{T}$ for the syntactic category of ModTT, generated by a straightforward extension of the signature depicted in Figure 3.*

*Equational presentation of specific effects.* It is important that our use of an equational logical framework does not prevent the extension of ModTT with non-trivial computational effects; although the effect of having a fixed collection of references cells or exceptions is clearly algebraic (see e.g. Plotkin and Power [2002]), 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 [2013] — as Staton's work shows, there is no obstacle to the equational presentation of *any* reasonable form of effect, but semantics are another story. We do not currently make any claim about the extension of our representation independence results to the setting of higher-order store, for instance.

## 3 A TYPE THEORY FOR SYNTHETIC PARAMETRICITY

Our goal is to define a *type theory of parametricity structures* $\mathsf{TT}_{\mathsf{PS}}$, 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*.

$\mathsf{TT}_{\mathsf{PS}}$ is an extension of the internal dependent type theory of a presheaf topos with modal features corresponding to phase separated parametricity: therefore, $\mathsf{TT}_{\mathsf{PS}}$ has dependent products, dependent sums, extensional equality types, a strictly univalent universe $\Omega$ of proof irrelevant

propositions, a strict hierarchy of universes $\mathcal{U}_\alpha$ of types, inductive types, subset types, and effective quotient types (consequently, strict pushouts). We first axiomatize $\mathsf{TT}_{\mathsf{PS}}$ in the style of Orton and Pitts [2016], and in Section 5 we construct a suitable model of $\mathsf{TT}_{\mathsf{PS}}$ using topos theory. Referring to the types of $\mathsf{TT}_{\mathsf{PS}}$, 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 $\mathsf{TT}_{\mathsf{PS}}$ a system of proof irrelevant propositions corresponding to the static part and the disjoint (left)-syntactic and (right)-syntactic parts of a parametricity structure.

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

The (open) *static modality* is defined by exponentiation $\mathsf{St} = \blacksquare_{\mathsf{st}} \Rightarrow -$ as usual; we likewise have an (open) *syntactic modality* $\mathsf{Syn} = \blacksquare_{\mathsf{syn}} \Rightarrow -$ defined in the same way. These modalities isolate the static and syntactic parts of a parametricity structure respectively; because $\blacksquare_{\mathsf{syn/l}}, \blacksquare_{\mathsf{syn/r}}$ have no overlap, we have an isomorphism $\mathsf{Syn}(A) = (\blacksquare_{\mathsf{syn/l}} \Rightarrow A) \times (\blacksquare_{\mathsf{syn/r}} \Rightarrow A)$. This isomorphism is captured more generally by the following *systems* notation of Cohen et al. [2017] from cubical type theory for constructing maps out of disjunctions of propositions:

Notation 3.1 (Systems). *Following Cohen et al. [2017], we employ the notation of* systems *for constructing elements of parametricity structures underneath the assumption of disjunction of propositions $\phi \vee \psi$: when $\phi \wedge \psi$ implies $a = a' : A$, we may write $[\phi \to a \mid \psi \to a']$ for the unique element of $A$ that restricts to $a, a'$ on $\phi, \psi$ respectively.*

Notation 3.2 (Extension). *As foreshadowed by the* static extent *of ModTT, every proposition $\phi : \Omega$ gives rise to an* extension type *connective [Riehl and Shulman 2017]: if $A$ is a parametricity structure and $a$ is an element of $A$ assuming $\phi = \top$, then $\{A \mid \phi \to a\}$ is the parametricity structure of elements $a' : A$ such that $a = a'$ when $\phi = \top$.*

*3.1.1 Complementary closed modalities.* The static modality neutralizes the dynamic part of a parametricity structure (in both syntax and semantics), and the syntactic modality neutralizes 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.

If $\phi : \Omega$ is a proposition, then the *closed* modality complementing the open modality $\phi \Rightarrow -$ is the join $A \mapsto A \star \phi = A \sqcup_{A \times \phi} \phi$, definable by pushout or quotient. Specifically, we will use $\mathsf{Sem}(A) = A \star \blacksquare_{\mathsf{syn}}$ and $\mathsf{Dyn}(A) = A \star \blacksquare_{\mathsf{st}}$. A consequence of this definition is that $\mathsf{Syn}(\mathsf{Sem}(A)) = \mathbf{1}$ and $\mathsf{St}(\mathsf{Dyn}(A)) = \mathbf{1}$, reflecting the idea that the closed modality neutralizes the open modality.

*3.1.2 Universes and modalities.* Each universe $\mathcal{U}_\alpha$ of $\mathsf{TT}_{\mathsf{PS}}$ may be restricted to a *universe of modal parametricity structures* for each modality described above. For instance the universe of purely syntactic parametricity structures is $\mathsf{Syn}(\mathcal{U}_\alpha)$, and the collection of elements of $A : \mathsf{Syn}(\mathcal{U}_\alpha)$ is just $\prod_{* : \blacksquare_{\mathsf{syn}}} A(*)$. Likewise, the collection of elements of $A : \mathsf{Sem}(\mathcal{U}_\alpha)$ is the subset $\left\{a : \mathsf{Sem}\left(\sum_{A : \mathcal{U}_\alpha} a : A\right) \;\middle|\; \mathsf{Sem}(\pi_1(a)) = A\right\}$. We assert that $\mathsf{TT}_{\mathsf{PS}}$ moreover satisfies the following *strictification* axiom of Orton and Pitts [2016]

Axiom 3.3 (Strictification). *Let $\phi : \Omega$ be a proposition, and let $A : \phi \Rightarrow \mathcal{U}_\alpha$ be a partial type defined on the extent of $\phi$, 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 3.3 above plays a critical role in the constructions of Section 3.2, letting $\phi := \blacksquare_{\mathsf{syn}}$.

Corollary 3.4 (Realignment). *Let $A : \mathsf{Syn}(\mathcal{U}_\alpha)$ be a syntactic type, and fix $\tilde{A} : \mathcal{U}_\alpha$ whose syntactic part is isomorphic to A, i.e. we have $f : \mathsf{Syn}(A \cong \tilde{A})$. Then there exists a type $f^*\tilde{A} : \mathcal{U}_\alpha$ with $f^\dagger\tilde{A} : f^*\tilde{A} \cong \tilde{A}$, such that both $\mathsf{Syn}(f^*\tilde{A} = A)$ and $\mathsf{Syn}(f^\dagger\tilde{A} = f)$ strictly.*

*3.1.3 Doubled embedding of syntax.* We assume that $\mathbb{T}$ is *twice* embedded into $\mathsf{TT}_{\mathsf{PS}}$, once "on the left" and once "on the right". Concretely, this is realized by a pair of embeddings $\lfloor - \rfloor_{\mathsf{L}}, \lfloor - \rfloor_{\mathsf{R}}$ that take the syntax of ModTT to objects in $\mathsf{TT}_{\mathsf{PS}}$ that are purely left/right syntactic. These may be combined into a doubled embedding $\lfloor X \rfloor_{\mathsf{LR}} = [\text{🔒}_{\mathsf{syn/l}} \to \lfloor X \rfloor_{\mathsf{L}} \mid \text{🔒}_{\mathsf{syn/r}} \to \lfloor X \rfloor_{\mathsf{R}}]$.

*Example 3.5.* We have a purely-syntactic object of ModTT signatures $\lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}} : \mathsf{Syn}(\mathcal{U})$; in fact, considering Notation 3.2, we may assign the more specific type $\lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}} : \{\mathsf{Syn}(\mathcal{U}) \mid \text{🔒}_{\mathsf{syn/l}} \to \lfloor \mathbf{Sig} \rfloor_{\mathsf{L}} \mid \text{🔒}_{\mathsf{syn/r}} \to \lfloor \mathbf{Sig} \rfloor_{\mathsf{R}}\}$ reflecting the fact that $\lfloor - \rfloor_{\mathsf{LR}}$ is the doubling map.

We additionally require that under the assumption of $\text{🔒}_{\mathsf{syn}}$, we have $\text{🔒}_{\mathsf{st}} = \lfloor \text{🔒}_{\mathsf{st}} \rfloor_{\mathsf{LR}}$; in other words, we require $\text{🔒}_{\mathsf{st}} : \{\Omega \mid \text{🔒}_{\mathsf{syn}} \to \lfloor \text{🔒}_{\mathsf{st}} \rfloor_{\mathsf{LR}}\}$.

### 3.2 A parametric model of ModTT in $\mathsf{TT}_{\mathsf{PS}}$

In this section, we exhibit a second algebra for ModTT in $\mathsf{TT}_{\mathsf{PS}}$ that *lies over* the doubled embedding described in Section 3.1.3. In particular, to every object $A$ of $\mathbb{T}$ we will associate an object $A^*$ of $\mathsf{TT}_{\mathsf{PS}}$ such that assuming $\text{🔒}_{\mathsf{syn}}$, we have $A^* = \lfloor A \rfloor_{\mathsf{LR}}$. We do not show every part of the construction of this "parametric algebra", but instead give several representative cases to illustrate the comparative ease of our approach in contrast to prior work.

*3.2.1 Parametricity structure of judgments.* We define a parametricity structure of signatures *over* the purely syntactic parametricity structure of syntactic signatures $\lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}$. Letting $\alpha < \beta < \gamma$, we define $\mathbf{Sig}^* : \mathcal{U}_\beta$ with the following interface:

$$\mathbf{Sig}^* : \{\mathcal{U}_\gamma \mid \text{🔒}_{\mathsf{syn}} \to \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}\}$$
$$\mathbf{Sig}^* \cong \textstyle\sum_{\sigma : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}} \{\mathcal{U}_\beta \mid \text{🔒}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma)\}$$

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

$$\begin{aligned}
\mathbf{Sig}^{*\prime} &= \textstyle\sum_{\sigma : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}} \{\mathcal{U}_\beta \mid \text{🔒}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma)\} && \text{def. of } \mathbf{Sig}^{*\prime} \\
&= \textstyle\sum_{\sigma : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}} \{\mathcal{U}_\beta \mid \top \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma)\} && \text{🔒}_{\mathsf{syn}} = \top \\
&\cong \textstyle\sum_{\sigma : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}} \mathbf{1} && \text{singleton} \\
&\cong \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}} && \text{trivial}
\end{aligned}$$

Therefore, by Corollary 3.4 we obtain $\mathbf{Sig}^* \cong \mathbf{Sig}^{*\prime}$ *strictly* extending $\lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}$ as desired. Next, we may define the extension of a glued signature directly:

$$\mathbf{Val}^* : \{\mathbf{Sig}^* \Rightarrow \mathcal{U}_\beta \mid \text{🔒}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}\}$$
$$\mathbf{Val}^*(\sigma, \tilde{\sigma}) = \tilde{\sigma}$$

*3.2.2 Parametricity structure of dependent products.* We show that $\mathbf{Sig}^*$ is closed under dependent product (dependent sums are analogous); fixing $\sigma_0 : \mathbf{Sig}^*$ and $\sigma_1 : \mathbf{Val}^*(\sigma_0) \Rightarrow \mathbf{Sig}^*$, we may define $\Pi_{\mathbf{Sig}^*}(\sigma_0, \sigma_1) : \mathbf{Sig}^*$ as follows. We desire the first component to be the syntactic dependent product type $\sigma_\Pi = \lfloor \Pi_{\mathbf{Sig}} \rfloor_{\mathsf{LR}}(\sigma_0, \lambda x : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma_0).\sigma_1(x))$.[9] For the second component, we note that the

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

syntactic modality commutes with dependent products up to isomorphism, so (using Corollary 3.4) we may define the second component lying strictly over $\sigma_\Pi$:

$$\tilde{\sigma}_\Pi : \{\mathcal{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma_\Pi)\}$$
$$\tilde{\sigma}_\Pi \cong \Pi_{\mathcal{U}_\beta}(\mathbf{Val}^*(\sigma_0), \mathbf{Val}^* \circ \sigma_1)$$

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

Remark 3.6. *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. [2019]; Sterling and Angiuli [2020]). In these formulations, one is constantly using the left exactness of the gluing functor, 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 logos [Artin et al. 1972], a result that is easier to prove in generality than any specific type theoretic corollary.*

*3.2.3 Parametricity structure of types.* From the syntax of ModTT, we have the signature of types $\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}} : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}$ and its decoding $\langle\!| - |\!\rangle : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}}) \Rightarrow \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}$; we must provide parametricity structures for both. First, we may define a collection of *small* statically connected parametricity structures for types, using Corollary 3.4:

$$\mathbf{Type}^* : \{\mathcal{U}_\beta \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}})\}$$
$$\mathbf{Type}^* \cong \textstyle\sum_{t : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}})} \{\mathsf{Dyn}(\mathcal{U}_\alpha) \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\langle\!| t |\!\rangle)\}$$

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

$$\mathsf{Type}^* : \{\mathbf{Sig}^* \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \mathsf{Type} \rfloor_{\mathsf{LR}}\} \qquad \langle\!| - |\!\rangle^* : \{\mathbf{Val}^*(\mathsf{Type}^*) \Rightarrow \mathbf{Sig}^* \mid \text{🔓}_{\mathsf{syn}} \to \langle\!| - |\!\rangle\}$$
$$\mathsf{Type}^* = (\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}}, \mathbf{Type}^*) \qquad \langle\!| (t, \tilde{t}) |\!\rangle^* = (\langle\!| t |\!\rangle, \tilde{t})$$

*3.2.4 Parametricity structure of observables.* We have a type $\mathsf{bool} : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\lfloor \mathsf{Type} \rfloor_{\mathsf{LR}})$ and two constants $\#\mathsf{t}, \#\mathsf{f} : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\langle\!| \lfloor \mathsf{bool} \rfloor_{\mathsf{LR}} |\!\rangle)$; we must construct parametricity structures for all these. First, we define the collection of computable booleans as follows, using Corollary 3.4 as usual:[10]

$$\mathbf{bool}^* : \{\mathcal{U}_\alpha \mid \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\langle\!| \lfloor \mathsf{bool} \rfloor_{\mathsf{LR}} |\!\rangle)\}$$
$$\mathbf{bool}^* \cong \textstyle\sum_{b : \mathbf{Val}(\langle\!| \mathsf{bool} |\!\rangle)} \mathsf{Dyn}(\mathsf{Sem}(\{\tilde{b} : 2 \mid b = \mathsf{case}[\tilde{b}](\#\mathsf{t}, \#\mathsf{f})\}))$$

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

$$\mathsf{bool}^* : \{\mathbf{Val}^*(\mathsf{Type}^*) \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \mathsf{bool} \rfloor_{\mathsf{LR}}\}$$
$$\mathsf{bool}^* = (\mathsf{bool}, \mathbf{bool}^*)$$

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

$$\#\mathsf{t}^*, \#\mathsf{f}^* : \{\mathbf{Val}^*(\langle\!| \mathsf{bool}^* |\!\rangle^*) \mid \text{🔓}_{\mathsf{syn}} \to \lfloor \#\mathsf{t} \rfloor_{\mathsf{LR}}, \lfloor \#\mathsf{f} \rfloor_{\mathsf{LR}}\}$$
$$\#\mathsf{t}^* = \left(\#\mathsf{t}, \eta_{\mathsf{Dyn}}(\eta_{\mathsf{Sem}}(0))\right) \qquad \#\mathsf{f}^* = \left(\#\mathsf{f}, \eta_{\mathsf{Dyn}}(\eta_{\mathsf{Sem}}(1))\right)$$

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

*3.2.5 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. [2008] for gluing together two monads along a monad morphism. Emanating from the syntax is an internal monad $\{-\} : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}} \Rightarrow \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}}$ 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 $\mathrm{T} : \mathrm{Sem}(\mathcal{U}_\beta) \Rightarrow \mathrm{Sem}(\mathcal{U}_\beta)$ be such a monad where each $\mathrm{T}(X)$ is *connected* for the open modality St, i.e. $\mathrm{St}(\mathrm{T}(X)) = \mathbf{1}$. We furthermore have an internal functor $F : \lfloor \mathbf{Sig} \rfloor_{\mathsf{LR}} \Rightarrow \mathrm{Sem}(\mathcal{U}_\beta)$ defined by taking the purely semantic part of the collection of modules of every syntactic signature:

$$F(\sigma) = \eta_{\mathsf{Sem}}(\lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\sigma))$$

We parameterize the constructions of this section in a *monad morphism* $\mathsf{run} : \{-\} \Rightarrow \mathrm{T}$ over $F$ in the sense of Street [1972], i.e. an internal natural transformation $\mathsf{run} : \mathrm{T} \circ F \Rightarrow F \circ \{-\}$ satisfying a number of coherence conditions. Following Goubault-Larrecq et al. [2008], we may glue the two monads together along this morphism to define a monad on $\mathbf{Sig}^*$, i.e. the internal category of glued signatures and glued modules. Fixing $\sigma : \mathbf{Sig}^*$, we may define a type $\mathrm{T}_{\mathcal{X}}(\sigma) : \mathcal{U}_\beta$ as follows, writing $\pi_\sigma : \eta_{\mathsf{Sem}}(\mathbf{Val}^*(\sigma)) \Rightarrow \mathrm{T}(F(\sigma))$ for the obvious projection in $\mathrm{Sem}(\mathcal{U}_\beta)$:

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

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

$$\{-\}^* : \mathbf{Sig}^* \Rightarrow \mathbf{Sig}^*$$
$$\{\sigma\}^* = (\{\sigma\}, \mathrm{T}_{\mathcal{X}}(\sigma))$$

If ModTT is suitably extended by monadic operations (such as those corresponding to exceptions, printing, a global reference cell, etc.), then the assumptions of this section are readily substantiated by the corresponding monad on purely semantic objects. Some computational effects may require the constructions of Section 5 to be relativized from **Set** to a suitable presheaf logos — for instance, partiality / general recursion can be modeled by replacing **Set** with the topos of trees as in Birkedal et al. [2011]; Paviotti [2016].

*Example 3.7.* Suppose that ModTT were extended with an operation $\mathsf{throw}_\sigma : \{\sigma\}$ for each signature $\sigma$, such that $\{-\}$ corresponds to the *exception monad*. We may glue this together with the internal monad $\mathrm{T}(X) = \mathrm{Dyn}(1 + X)$ on the internal category of purely semantic parametricity structures. We must define a family of functions $\mathsf{run}_\sigma : \mathrm{T}(F(\sigma)) \Rightarrow F\{\sigma\}$. Because $F\{\sigma\}$ is purely dynamic and Dyn is a lex idempotent modality, any such function $\mathsf{run}_\sigma$ is uniquely determined by a map $1 + F(\sigma) \Rightarrow F(\sigma)$, which we may choose as follows:

$$\mathit{inl}(*) \mapsto \eta_{\mathsf{Sem}}(\mathsf{throw}_\sigma) \qquad\qquad \mathit{inr}(x) \mapsto \eta_{\mathsf{Sem}}(\mathsf{ret}(x))$$

Then, the monad $\mathrm{T}_{\mathcal{X}}(\sigma)$ on a parametricity structure $\sigma : \mathbf{Sig}^*$ associates to each syntactic computation $M : \{\sigma\}$ 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 3.7. 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.

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

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

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

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

### 4.1 A simulation structure between two queues

We may define an abstract type of queues QUEUE together with two implementations as in Harper [2016], depicted in Figure 4. We will observe that the semantic part of QUEUE* is the collection of proof-relevant phase separated simulation relations between two given closed syntactic queues. First, we note the meaning of QUEUE* in the glued setting:

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

The two queue implementations internalize as elements $\mathsf{Q}_0 : \lfloor \mathbf{Val} \rfloor_{\mathsf{L}}(\mathsf{QUEUE}), \mathsf{Q}_1 : \lfloor \mathbf{Val} \rfloor_{\mathsf{R}}(\mathsf{QUEUE})$; these can be combined into $\mathsf{Q}_{01} : \lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\mathsf{QUEUE})$ by splitting, $\mathsf{Q}_{01} = [\text{🔓}_{\mathsf{syn/l}} \to \mathsf{Q}_0 \mid \text{🔓}_{\mathsf{syn/r}} \to \mathsf{Q}_1]$. We may define a purely dynamic type that represents the *invariant structure* on a pair of queues, using Corollary 3.4:

$$\mathsf{invariant} : \{\mathsf{Dyn}(\mathcal{U}_\alpha) \mid \text{🔓}_{\mathsf{syn}} \to \eta_{\mathsf{Dyn}}(\lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\mathsf{Q}_{01}))\}$$

$$\mathsf{invariant} \cong \textstyle\sum_{q:\lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\langle\!| \mathsf{Q}_{01}.\mathsf{t} |\!\rangle)} \mathsf{Sem}(\{\vec{x}, \vec{y}, \vec{z} : \mathsf{Dyn}(2^\star) \mid \vec{x} = (\vec{y} + \mathit{rev}(\vec{z})) \wedge q = [\text{🔓}_{\mathsf{syn/l}} \to \lceil \vec{x} \rceil \mid \text{🔓}_{\mathsf{syn/r}} \to (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]\})$$

We may then *define* a single parametricity structure to unite the two implementations under the invariant above, depicted in Figure 5.

### 4.2 A representation independence result

Let $f : \mathsf{QUEUE} \Rightarrow \langle\!| \mathsf{bool} |\!\rangle$; then we have $f(\mathsf{Q}_0) = f(\mathsf{Q}_1)$. This can be seen by considering the image of $f$ under the parametricity interpretation of ModTT into $\mathsf{TT}_{\mathsf{PS}}$, $f^* : \mathsf{QUEUE}^* \Rightarrow \langle\!| \mathsf{bool}^* |\!\rangle^*$. Applying $f^*$ to the simulation queue defined in Figure 5, we have a single element of $\langle\!| \mathsf{bool}^* |\!\rangle^*$ relating two syntactic booleans:

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

But we have defined bool* along the diagonal (Section 3.2.4), so this actually proves that either $f(\mathsf{Q}_0) = f(\mathsf{Q}_1) = \#\mathsf{t}$ or $f(\mathsf{Q}_1) = f(\mathsf{Q}_2) = \#\mathsf{f}$.

## 5 THE TOPOS OF PHASE SEPARATED PARAMETRICITY STRUCTURES

The simplest way to substantiate the type theory $\mathsf{TT}_{\mathsf{PS}}$ of Section 3 is to use the existing infrastructure of Grothendieck topoi and Artin gluing [Artin et al. 1972]; every topos possesses an extremely rich *internal type theory*, so our strategy will be roughly as follows:

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

$$\begin{aligned} t &: \{\mathbf{Val}^*(\mathsf{Type}^*) \mid \lock_{\mathsf{syn}} \to \mathtt{Q}_{01}.\mathtt{t}\} \\ emp &: \{\mathbf{Val}^*(\langle\!|t|\!\rangle^*) \mid \lock_{\mathsf{syn}} \to \mathtt{Q}_{01}.\mathtt{emp}\} \\ ins &: \{\mathbf{Val}^*(\langle\!|\mathsf{bool}^* * t \rightharpoonup t|\!\rangle^*) \mid \lock_{\mathsf{syn}} \to \mathtt{Q}_{01}.\mathtt{ins}\} \\ rem &: \{\mathbf{Val}^*(\langle\!|t \rightharpoonup \mathsf{bool}^* * t|\!\rangle^*) \mid \lock_{\mathsf{syn}} \to \mathtt{Q}_{01}.\mathtt{rem}\} \end{aligned}$$

These operations are implemented in $\mathsf{TT}_{\mathsf{PS}}$ as follows.

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

where

$$\begin{aligned} &\mathsf{invariant} : \{\mathsf{Dyn}(\mathcal{U}_\alpha) \mid \lock_{\mathsf{syn}} \to \eta_{\mathsf{Dyn}}(\lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\mathtt{Q}_{01}))\} \\ &\mathsf{invariant} \cong \textstyle\sum_{q:\lfloor \mathbf{Val} \rfloor_{\mathsf{LR}}(\langle\!|\mathtt{Q}_{01}.\mathtt{t}|\!\rangle)} \mathsf{Sem}(\{\vec{x},\vec{y},\vec{z} : \mathsf{Dyn}(2^\star) \mid \vec{x} = (\vec{y} + rev(\vec{z})) \wedge q = [\lock_{\mathsf{syn/l}} \to \lceil \vec{x} \rceil \mid \lock_{\mathsf{syn/r}} \to (\lceil \vec{y} \rceil, \lceil \vec{z} \rceil)]\}) \end{aligned}$$

Fig. 5. Constructing a simulation between the two queue implementations becomes a straightforward *programming problem* in $\mathsf{TT}_{\mathsf{PS}}$.

(1) Embed the syntax of ModTT into a topos $\widehat{\mathbb{T}}$; this will be the topos corresponding to the free cocompletion of the syntactic category $\mathbb{T}$ (see Notation 2.3). The copower $2 \cdot \widehat{\mathbb{T}}$ will then serve as a suitable index for *binary* parametricity.
(2) Identify a topos $\mathbb{S}$ that captures the notion of phase distinction: a type in the internal language of $\mathbb{S}$ should be a set that has both a static part and a dynamic part depending on it.
(3) Glue the topos of (doubled) syntax $2 \cdot \widehat{\mathbb{T}}$ and the topos of semantics $\mathbb{S}$ together to form a topos $\mathcal{X}$ of *phase separated parametricity structures*: a type in the internal language of $\mathcal{X}$ will have several aspects corresponding to the orthogonal distinctions ((left syntax, right syntax), semantics) and (static, dynamic). The topos $\mathcal{X}$ then has enough structure to model all of $\mathsf{TT}_{\mathsf{PS}}$.

### 5.1 Topo-logical metatheory of programming languages

We argue that the metatheory of type theories and programming languages can be developed in the language of topoi;[11] as Grothendieck advocated in his 1973 Buffalo lectures (more recently echoed by Anel and Joyal [2019]; Vickers [2007]), one may profitably view topoi *not* as structured categories, but as generalized space. Indeed, the category of topoi behaves not at all like the category of categories, but more like a (constructive) version of the category of topological spaces, and many aspects of the metatheory of type theory can be seen to have intuitive geometrical meaning.

Under the ubiquitous *geometry–algebra* duality, the 2-category of topoi is *anti-equivalent* to the 2-category of presentable categories having universal colimits, disjoint sums, and effective equivalence relations; we will refer to categories of the latter kind as *logoi*, following Anel and Joyal [2019], setting **Topos** = **Logos**$^{\mathrm{op}}$. This anti-equivalence associates to each topos $\mathcal{X}$ its dual category of sheaves $\mathrm{Sh}(\mathcal{X})$; a morphism $f : \mathcal{X} \longrightarrow \mathcal{Y}$ of topoi is taken to its lex and cocontinuous inverse image functor $f^* : \mathrm{Sh}(\mathcal{Y}) \longrightarrow \mathrm{Sh}(\mathcal{X})$, necessarily possessing a right adjoint $f_* : \mathrm{Sh}(\mathcal{X}) \longrightarrow \mathrm{Sh}(\mathcal{Y})$ called the *direct image*. We take the inverse image as a *definition* of morphisms of logoi.

[11] In this paper, every topos will be a *Grothendieck* topos.

*Definition 5.1.* A morphism $F : \mathcal{E} \longrightarrow \mathcal{F}$ of logoi is a left exact and cocontinuous functor.

Remark 5.2. *The left exactness and cocontinuity of the inverse image to a morphism of topoi 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. Sheaves (the objects of logoi) play exactly the same role for topoi as opens do for topological spaces; the notion of an open still makes sense for topoi, but it is a special case of sheaf (see Definition 5.4).*

The relationship between a topos $\mathcal{X}$ and its dual logos $\mathrm{Sh}(\mathcal{X})$ is analogous to that between a topological space and its corresponding frame of opens:[12] these are actually the same objects, but in the one case we view morphisms geometrically (taking points to points), whereas in the latter case we think of morphisms from the perspective of the inverse image, taking opens to opens.

*Example 5.3 (Points of a topos).* The logos of sets **Set** is, classically, the category of sheaves on the one-point space. Therefore, we *define* the topos $* : \mathbf{Topos}$ to be the unique topos such that $\mathrm{Sh}(*) = \mathbf{Set}$. A morphism of topoi $* \longrightarrow \mathcal{X}$ is called a *point* of $\mathcal{X}$; from the perspective of sheaves, a point is therefore a left exact and cocontinuous functor $\mathrm{Sh}(\mathcal{X}) \longrightarrow \mathbf{Set}$; an arbitrary morphism $\mathcal{Y} \longrightarrow \mathcal{X}$ can be called a *generalized point* of $\mathcal{X}$, thinking of $\mathcal{Y}$ as the stage of definition. We can already see one of the directions in which topoi generalize topological spaces: whereas a topological space has a set of points, a topos has a category of points.

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

*Definition 5.4 (Opens of a topos).* An *open* of a topos $\mathcal{X}$ is defined to be a subterminal in $\mathrm{Sh}(\mathcal{X})$, i.e. a proof-irrelevant proposition in the internal type theory of $\mathcal{X}$. We will write $\mathcal{O}_{\mathcal{X}}$ for the frame of opens of the topos $\mathcal{X}$. An open $U : \mathcal{O}_{\mathcal{X}}$ gives rise to a subtopos $\mathcal{X}_U \hookrightarrow \mathcal{X}$: we define $\mathcal{X}_U$ to be the topos corresponding to the slice logos $\mathrm{Sh}(\mathcal{X})_{/U}$.

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

*Example 5.5 (Sierpiński topos, the classifier of opens).* There is a topos $\mathbb{S}$ equipped with two points $\circ, \bullet : * \longrightarrow \mathbb{S}$ with the following property: every open subtopos $\mathcal{X}_U \hookrightarrow \mathcal{X}$ arises in a unique way by pullback along the "open point" $\circ : * \longrightarrow \mathbb{S}$:

$$\begin{array}{ccc} \mathcal{X}_U & \longrightarrow & * \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \circ} \\ \mathcal{X} & \xrightarrow[\lfloor U \rfloor]{} & \mathbb{S} \end{array}$$

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

The view of opens via their (geometrical) characteristic maps will become important for us in Section 5.2.1, where we shall use it to obtain a *phase separated* version of the global sections functor.

*Example 5.6 (Presheaves).* Let $\mathcal{C}$ be a small category; then $\mathrm{Pr}(\mathcal{C})$ is the category of *presheaves* on $\mathcal{C}$, i.e. functors $\mathcal{C}^{\mathrm{op}} \longrightarrow \mathbf{Set}$. We write $\widehat{\mathcal{C}}$ for topos whose sheaves are the presheaves on $\mathcal{C}$, i.e. $\mathrm{Sh}\big(\widehat{\mathcal{C}}\big) = \mathrm{Pr}(\mathcal{C})$. Suppose that $\mathcal{C}$ has finite limits; then a *generalized point* $\mathcal{Y} \longrightarrow \widehat{\mathcal{C}}$ corresponds to a left exact functor $\mathcal{C} \longrightarrow \mathrm{Sh}(\mathcal{Y})$.

[12] To be precise, the perfect analogy is with *locales* and their frames of opens.

### 5.2 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 5.7. *The Sierpiński topos may be constructed in terms of presheaves (Example 5.6): letting $\Delta^1$ be the category containing two objects and an arrow between them, we define $\mathbb{S}$ to be the presheaf topos $\widehat{\Delta^1}$. It then easy to see that $\mathrm{Sh}(\mathbb{S}) = \mathrm{Pr}(\Delta^1) = \mathbf{Set}^{\to}$, i.e. the category of families of sets.*

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

*5.2.1 Phase separated global sections.* Let $\mathbb{T}$ be the syntactic category of ModTT; we may manipulate $\mathbb{T}$ in the language of topoi by enlarging it to $\widehat{\mathbb{T}}$, the topos of presheaves on $\mathbb{T}$ (see Example 5.6). $\widehat{\mathbb{T}}$ can be thought of as a topos of *generalized syntax*.

By the universal property of the Sierpiński topos (Example 5.5), the open $\mathbb{0}_{\mathrm{st}} : \Omega_{\widehat{\mathbb{T}}}$ corresponds to a unique continuous map $\boldsymbol{\gamma} : \widehat{\mathbb{T}} \longrightarrow \mathbb{S}$ of topoi; it is appropriate to think of the direct image $\boldsymbol{\gamma}_* : \mathrm{Pr}(\mathbb{T}) \longrightarrow \mathrm{Sh}(\mathbb{S})$ as a *phase separated* version of the global sections functor, sending each object to the canonical projection map from its collection of global elements to their static parts.

Computation 5.8. *To see that we have correctly understood the action of the direct image, we first note that the inverse image $\boldsymbol{\gamma}^* : \mathrm{Sh}(\mathbb{S}) \longrightarrow \mathrm{Pr}(\mathbb{T})$ is completely determined under the Yoneda embedding $\mathrm{y} : \Delta^1 \hookrightarrow \mathrm{Sh}(\mathbb{S})$ by the diagram $\Delta^1 \longrightarrow \mathrm{Pr}(\mathbb{T})$ corresponding to the open $\mathbb{0}_{\mathrm{st}} \rightarrowtail \mathbf{1}_{\mathrm{Pr}(\mathbb{T})}$. Therefore, we may compute the direct image $\boldsymbol{\gamma}^* \dashv \boldsymbol{\gamma}_* : \mathrm{Pr}(\mathbb{T}) \longrightarrow \mathrm{Sh}(\mathbb{S})$ by adjointness:*

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

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

### 5.3 Topos of parametricity structures

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

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

structures by gluing. We may consider the following morphism $\boldsymbol{\rho} : 2 \cdot \widehat{\mathbb{T}} \longrightarrow \mathbb{S}$ of topoi:

$$2 \cdot \widehat{\mathbb{T}} \xrightarrow{\nabla} \widehat{\mathbb{T}} \xrightarrow{\boldsymbol{\gamma}} \mathbb{S} \qquad (\boldsymbol{\rho} : 2 \cdot \widehat{\mathbb{T}} \to \mathbb{S})$$

Computation 5.9. *The direct image $\boldsymbol{\rho}_* : \mathrm{Pr}(\mathbb{T})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ takes a pair $(X, Y) : \mathrm{Pr}(\mathbb{T})^2$ of (generalized) syntactic objects to $\boldsymbol{\gamma}_* X \times \boldsymbol{\gamma}_* Y$, the product of their phase separated global sections.*

Construction 5.10 (Topos of parametricity structures). *We then obtain a topos $\mathcal{X}$ whose sheaves correspond to parametricity structures by gluing, specifically via a phase separated version of the Sierpiński cone construction: we first form the Sierpiński cylinder $\left(2 \cdot \widehat{\mathbb{T}}\right) \times \mathbb{S}$ and then* pinch *the end corresponding to the closed point $\bullet \in \mathbb{S}$ along $\boldsymbol{\rho}$ as follows:*

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

Remark 5.11. *The Sierpiński topos $\mathbb{S}$ plays* two *roles in Construction 5.10: first, we use $\mathbb{S}$ to form a* cylinder *on $2 \cdot \widehat{\mathbb{T}}$ (which is always done in gluing), and secondly $\mathbb{S}$ is the codomain of the functor we are gluing along. This second use corresponds to the fact that we are constructing* phase separated *parametricity structures rather than ordinary parametricity structures, in which case we would be gluing into the punctual topos $*$.*

Computation 5.12. *The logos $\mathrm{Sh}(\mathcal{X})$ corresponding to $\mathcal{X}$ may be computed by dualizing the diagram of Construction 5.10, noting that $\mathrm{Sh}(\mathcal{Y} \times \mathbb{S}) = \mathrm{Sh}(\mathcal{Y})^{\to}$ and recalling that under this identification, the inverse image of the (closed, open) point is the (domain, codomain) functor:*

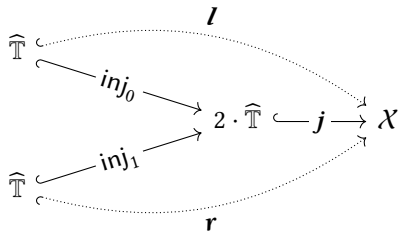
$$\begin{array}{ccc} \mathrm{Sh}(\mathcal{X}) & \longrightarrow & \left(\mathrm{Pr}(\mathbb{T})^2\right)^{\to} \\ {\scriptstyle \boldsymbol{i}^*}\downarrow & & \downarrow{\scriptstyle \mathrm{dom}} \\ \mathrm{Sh}(\mathbb{S}) & \xrightarrow[\boldsymbol{\rho}^*]{} & \mathrm{Pr}(\mathbb{T})^2 \end{array} \qquad \begin{array}{ccc} \mathrm{Sh}(\mathcal{X}) & \longrightarrow & \mathrm{Sh}(\mathbb{S})^{\to} \\ {\scriptstyle \boldsymbol{j}^*}\downarrow & & \downarrow{\scriptstyle \mathrm{cod}} \\ \mathrm{Pr}(\mathbb{T})^2 & \xrightarrow[\boldsymbol{\rho}_*]{} & \mathrm{Sh}(\mathbb{S}) \end{array}$$

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

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

The open embedding $\boldsymbol{j} : 2 \cdot \widehat{\mathbb{T}} \longrightarrow \mathrm{Sh}(\mathcal{X})$ corresponds (by definition) to an open $\blacksquare_{\mathsf{syn}} : \Omega_{\mathcal{X}}$, i.e. the subterminal parametricity structure $\blacksquare_{\mathsf{syn}} = \left(\emptyset_{\mathrm{Sh}(\mathbb{S})} \longrightarrow \boldsymbol{\rho}_*(\mathbf{1}_{\mathrm{Pr}(\mathbb{T})^2})\right)$. Let $\mathcal{Y}$ be a topos and $E$ a finite cardinal; the injections $\mathsf{inj}_e : \mathcal{Y} \hookrightarrow E \cdot \mathcal{Y}$ into the coproduct are in fact *open embeddings* [Johnstone 2002, Lemma B.3.4.1]. By composition, we may therefore reconstruct $\widehat{\mathbb{T}}$ as two different open

subtopoi of $\mathcal{X}$:




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

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

Lemma 5.13. *The logos of parametricity structures* $\mathrm{Sh}(\mathcal{X})$ *is a category of presheaves, i.e. there exists a category* $\mathcal{D}$ *such that* $\mathrm{Sh}(\mathcal{X}) \simeq \mathrm{Pr}(\mathcal{D})$.

Proof. First, we note that $\mathrm{Pr}(\mathbb{T})^2$ is $\mathrm{Pr}(2 \cdot \mathbb{T})$ and $\mathrm{Sh}(\mathbb{S})$ is $\mathrm{Pr}(\mathbf{\Delta}^1)$. Moreover, the direct image $\boldsymbol{\rho}_* : \mathrm{Pr}(\mathbb{T})^2 \longrightarrow \mathrm{Sh}(\mathbb{S})$ 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 9.6.10 (and worked out by Carboni and Johnstone [1995]). □

Consequently, we may construct $\mathrm{Sh}(\mathcal{X})$ such that its internal dependent type theory contains a *strict* hierarchy of universes $\mathcal{U}_\alpha$ à la Hofmann and Streicher [1997] and moreover enjoys the *strictification* axiom of Orton and Pitts [2016], restated here as Axiom 3.3. This is of course only possible because the high-altitude structure of our work respects the principle of equivalence.

## 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.

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 [Eisenberg 2016]. Taking Reynolds's dictum[13] seriously, we believe that the phase distinction is the prototype for any number of *levels of abstraction*, each corresponding to a different open modality.

Our approach is firmly rooted within the tradition of logical frameworks and categorical algebra, which has enabled us to reduce the highly technical (and very syntactic) logical relations arguments

[13] "Type structure is a syntactic discipline for enforcing levels of abstraction" [Reynolds 1983].

of prior work on modules to some trivial type theoretic arguments that are amenable to formalization à la Orton and Pitts [2016]. Actually formalizing the axioms of $\mathsf{TT}_{\mathsf{PS}}$ in a proof assistant like Agda, Coq, or Lean is within reach, thanks to the work of Gilbert et al. [2019].

The lax modality as an account of effects is natural, but admittedly does some violence to the dependent type structure: there can be very few useful laws governing the commutation of (non-degenerate) effects and dependent types. We plan to investigate whether the $\partial$CBPV calculus of Pédrot and Tabareau [2019] can provide a better way forward, replacing the standard "dependent product of a family of types" with the more refined "dependent product of a family of algebras".

Another area for future work is to instantiate ModTT with non-trivial effects, such as recursive types or higher-order store. These features, often accounted for using step-indexing, will likely require relativizing the construction of $\mathsf{TT}_{\mathsf{PS}}$ (Section 5) from **Set** to a logos supporting guarded recursion [Birkedal et al. 2011].

## ACKNOWLEDGMENTS

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

This work was supported in part by Air Force Office of Scientific Research under grants 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

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.

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.

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.

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))*, Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

Mathieu Anel and André Joyal. 2019. Topo-logie. (March 2019). http://mathieu.anel.free.fr/mat/doc/Anel-Joyal-Topo-logie.pdf Preprint.

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.

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

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.

Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract Effects and Proof-Relevant Logical Relations. *SIGPLAN Not.* 49, 1 (Jan. 2014), 619–631. https://doi.org/10.1145/2578855.2535869

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'.

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

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.

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 (LICS '11)*. IEEE Computer Society, Washington, DC, USA, 55–64.

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

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.

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

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).

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

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

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 In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I.

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))*, Herman Geuvers (Ed.), Vol. 131. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

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

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

Karl Crary. 2020. A Focused Solution to the Avoidance Problem. *Journal of Functional Programming* (2020). Special Issue: Essays in Honor of Robert Harper, to appear.

Karl Crary and Robert Harper. 2007. Syntactic Logical Relations for Polymorphic and Recursive Types. *Electronic Notes in Theoretical Computer Science* 172 (2007), 259–299. https://doi.org/0.1016/j.entcs.2007.02.010 Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

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

Derek Dreyer. 2007. A Type System for Recursive Modules. *SIGPLAN Not.* 42, 9 (Oct. 2007), 289–302. https://doi.org/10.1145/1291220.1291196

Derek Dreyer, Karl Crary, and Robert Harper. 2003. A Type System for Higher-Order Modules. *SIGPLAN Not.* 38, 1 (Jan. 2003), 236–249. https://doi.org/10.1145/640128.604151

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular Type Classes. *SIGPLAN Not.* 42, 1 (Jan. 2007), 63–70. https://doi.org/10.1145/1190215.1190229

Derek Dreyer, Robert Harper, and Karl Crary. 2005. *Understanding and Evolving the ML Module System*. Ph.D. Dissertation. USA.

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

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

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

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

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

Robert Harper. 1992. Constructing Type Systems over an Operational Semantics. *Journal of Symbolic Computation* 14, 1 (July 1992), 71–84.

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

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

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

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

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

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

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.

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))*, Herman Geuvers (Ed.), Vol. 131. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

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

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

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

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

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

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

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

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

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

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.

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.

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

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.

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.

Eugenio Moggi. 1989. A Category-Theoretic Account of Program Modules. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 101–117.

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.

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.

Marco Paviotti. 2016. *Denotational semantics in Synthetic Guarded Domain Theory*. Ph.D. Dissertation. Denmark.

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

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

Leaf Eames Petersen. 2005. *Certifying compilation for Standard ML in a type analysis framework.* Ph.D. Dissertation. Carnegie Mellon University.

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.

The RedPRL Development Team. 2018. `redtt`. http://www.github.com/RedPRL/redtt

The RedPRL Development Team. 2020. `cooltt`. http://www.github.com/RedPRL/cooltt

John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*.

Emily Riehl and Michael Shulman. 2017. A type theory for synthetic ∞-categories. *Higher Structures* 1 (2017).

Egbert Rijke, Michael Shulman, and Bas Spitters. 2017. Modalities in homotopy type theory. (2017). arXiv:1706.07526 http://arxiv.org/abs/1706.07526

Andreas Rossberg. 2018. 1ML – Core and modules united. *Journal of Functional Programming* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205

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

Zhong Shao. 1999. Transparent Modules with Fully Syntatic 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

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

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

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

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.

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

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))*, Herman Geuvers (Ed.), Vol. 131. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 31:1–31:25. https://doi.org/10.4230/LIPIcs.FSCD.2019.31 arXiv:1904.08562

Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2020. A cubical language for Bishop sets. (2020). Under review.

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

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

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

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

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

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

Taichi Uemura. 2019. A General Framework for the Semantics of Type Theory. (2019). arXiv:1904.04097 https://arxiv.org/abs/1904.04097

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

Philip Wadler. 2007. The Girard–Reynolds isomorphism (second edition). *Theoretical Computer Science* 375, 1 (2007), 201–226. https://doi.org/10.1016/j.tcs.2006.12.042 Festschrift for John C. Reynolds's 70th birthday.

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