Type Theory with Erasure

Constantine Theocharis
University of St Andrews, United Kingdom

Edwin Brady
University of St Andrews, United Kingdom

Abstract

Erasure enriches type theory with a distinction between runtime relevant and irrelevant data, allowing the compilation step to safely erase the latter. Versions of this feature are implemented by many systems, including Agda, Idris, and Rocq. We present a structural version of type theory with erasure, formulated as a second-order generalised algebraic theory (SOGAT). Erasure is encoded as a phase distinction between runtime and erased terms, in the form of a proposition that can appear in a context. This formulation has several advantages: it has models based on categories with families, is compatible with other structural features such as staging, and provides a better guideline for implementation. Through the model theory of SOGATs, we study the semantics of type theory with erasure in families of sets, which generalises to any Grothendieck topos equipped with a tiny proposition. We establish conservativity over Martin-Löf type theory (MLTT) in both phases. For code extraction, we construct a presheaf model that produces untyped lambda calculus programs and prove its correctness through gluing. Our results are formalised in Agda and we provide a toy elaborator implementation.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Type theory, erasure, dependent types, compilation, synthetic phase distinction, higher-order abstract syntax, logical frameworks

Digital Object Identifier 10.4230/LIPIcs.FSCD.2026.31

Supplementary Material *Software*: `https://github.com/kontheocharis/erasure-impl` [46]
archived at `swh:1:dir:18b74b13bf47b1740130031fe2cc393db1e95615`
Software: `https://github.com/kontheocharis/erasure-agda` [45]
archived at `swh:1:dir:985bed7fde164c22db65b5fc19c69f7ad8896270`

Acknowledgements We thank András Kovács, Szumi Xie, Bhakti Shah, Naïm Camille Favier and the anonymous reviewers for valuable discussions and feedback.

1 Introduction

When types depend on values, it becomes unclear which parts of a program are needed at runtime. One can perform whole-program analysis steps to heuristically detect and remove the majority of data that is not computationally relevant [10, 44]. However, the unbounded abstraction capabilities of dependent types make such techniques brittle and unpredictable for more complicated examples. What seems to have become the standard way to erase runtime-irrelevant data is to enrich the underlying type theory with a separate sort for terms to be erased by compilation. One popular approach is quantitative type theory (QTT) [34, 7]. Parameterised by a *quantitative semiring*, it encapsulates not only erasure but also various forms of substructural variable usage, including linearity. When instantiated to the ordered semiring $\{0 < \omega\}$, the resulting theory could be described as "Martin-Löf type theory (MLTT) with erasure". This approach is implemented by Agda [3] and Idris [25]. Besides this, there is a mechanism of "ghost sorts" [48] planned for Rocq [49] that achieves similar goals.

We contribute yet another approach to erasure, in the style of algebraic, reduction-free "type theory in type theory" [5]. The main difference over previous work is that our formulation is fully structural, corresponding to a well-studied class of type theories identified by Uemura [47]. As a result, we can rely on certain aspects of its metatheory and its implementation that are already understood in the general setting. We present erasure as a modular *second-order generalised algebraic theory* (SOGAT) extending MLTT, compatible with cumulative universes and inductive families.

From this formulation, we derive both syntactic and semantic results. On the syntactic side, we establish conservativity over MLTT (Theorem 17 and Corollary 18) and the independence of runtime data from erased data (Theorem 14). The former ensures that erasure does not alter the proving power of type theory, while the latter is our main theorem of "well-behavedness".

On the semantic side, we extend the standard interpretation of type theory in terms of sets and functions to account for erasure (Definition 20). This is possible when there is a tiny proposition available in the interpretation category. We give a fully worked example in the category of families of sets and discuss how it extends to Grothendieck toposes.

We then construct a code extraction model (Definition 21) producing untyped lambda calculus terms that only retain runtime data, along with a correctness proof by gluing that extracted code always tracks the standard interpretation (Theorem 27). This is intended to be implemented as part of the compilation pipeline post-typechecking.

We provide a demo implementation elaborating a high-level language with erased binders similar to Idris or Agda into the core language presented here (Section 8), and extracting untyped lambda calculus terms in the end. We include supplementary notes on the implementation of pattern unification for metavariables. Our theoretical results are formalised in Agda; the symbol 🔗 appearing throughout the paper is a hyperlink to the relevant part of the formalisation.

2 An informal presentation of erasure

Here we present erasure informally through a high-level dependently typed language to be elaborated into a yet unspecified core syntax. This behaves essentially the same as Idris/Agda with `0`/`@0` annotations. For now, we use type-in-type for simplicity.

Modes and usages. Terms are split into the *runtime* mode (ω) and the *erased* mode (0):

- runtime terms are written as $a \overset{\omega}{:} A$
- erased terms are written as $a \overset{0}{:} A$
- every runtime term can be used as an erased term

We order these as $0 < \omega$. This asymmetry reflects that at compile-time everything is accessible but at runtime only non-erased data is. This is a different kind of compile-time/runtime distinction from program staging or metaprogramming. A compile-time term will not necessarily have a *known value* at compile-time; rather, it will not survive *past* the compilation phase. The runtime phase comes after the compilation phase, so any data that reaches that point is also accessible before it – at compile-time. Variables are also annotated with a mode $\{0, \omega\}$. We will generally write $a : A$ for $a \overset{\omega}{:} A$, explicitly indicating only erased terms. In the core language the subusaging rule will be implemented by an explicit coercion mechanism (Section 2.1).

Functions. We write $(x \overset{i}{:} A) \to B$ for the type of dependent functions from A to B where the domain is at mode $i \in \{0, \omega\}$. A function with a runtime domain $(x : A) \to B$ survives compilation, while a function with an erased domain $(x \overset{0}{:} A) \to B$ is compiled to its return

value directly. In the latter case, the rules of the theory ensure that the return value will not have any dependency on the input x that can be observed at runtime. For example, this results in every closed function $(n \overset{0}{:} \mathsf{Nat}) \to \mathsf{Nat}$ being compiled to a constant numeral (Theorem 27).

Universes. The universe of types, denoted U, is not computationally relevant: we only need an erased code for a type to decode it. In other words, if $A \overset{0}{:} \mathsf{U}$ then A is a type. For type dependency, such as in $(x \overset{i}{:} A) \to B$, regardless of the mode i of the bound variable, it suffices for B to be applied to an erased $x \overset{0}{:} A$ to yield a type. This means there is an equivalence between type families $(x \overset{0}{:} A) \to \mathsf{U}$ and $(x \overset{\omega}{:} A) \to \mathsf{U}$.

▶ **Example 1.** The identity function $\lambda\{A\}\, x.\ x : \{A \overset{0}{:} \mathsf{U}\} \to A \to A$ extracts to the untyped identity function $\lambda x.\ x$, erasing the type argument A. Here we freely use the notation $\{x \overset{i}{:} A\} \to B$ for an implicit function type, which is merely an elaboration feature.

▶ **Example 2.** The signature describing length-indexed lists, or vectors, is

$$\begin{aligned}
&\mathsf{Vec} : (A \overset{0}{:} \mathsf{U}) \to (n \overset{0}{:} \mathsf{Nat}) \to \mathsf{U}\\
&\mathsf{nil} : \{A \overset{0}{:} \mathsf{U}\} \to \mathsf{Vec}\ A\ \mathsf{zero}\\
&\mathsf{cons} : \{A \overset{0}{:} \mathsf{U}\} \to \{n \overset{0}{:} \mathsf{Nat}\} \to (x : A) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n)
\end{aligned}$$

This is a standard example of erasure, where vectors do not store their length at runtime due to the usage of 0 in the n argument of cons. The types are also all erased.

Pairs. Similar to functions, $(x \overset{i}{:} A) \times B$ is the type of mode-aware dependent pairs; if $i = 0$ then the first projection is erased, and if $i = \omega$ it exists at runtime. We can also extend this to customise the mode of the second projection $(x \overset{i}{:} A) \times^j B$, which is equivalent to $(x \overset{i}{:} A) \times ((\_ \overset{j}{:} B) \times \mathsf{Unit})$ but with a better runtime representation in the case where $j = 0$.

▶ **Example 3.** Erased dependent pairs capture the situation where we have some index that needs to be packaged existentially but which we never access at runtime. For example, we can package the erased length of a vector, to get a list:

$$\begin{aligned}
&\mathsf{List}\ A : \mathsf{U}\\
&\mathsf{List}\ A = (n \overset{0}{:} \mathsf{Nat}) \times \mathsf{Vec}\ A\ n
\end{aligned}$$

At runtime, only the second projection of an erased pair survives, so if defined this way, lists and vectors have the same runtime representation.

▶ **Example 4.** If the second projection is erased instead, this captures the situation where we want to carry a proof about a runtime object that shouldn't exist at runtime. We can define the type of natural numbers less than n by

$$\begin{aligned}
&\mathsf{Fin} : \mathsf{Nat} \to \mathsf{U}\\
&\mathsf{Fin}\ n = (k : \mathsf{Nat}) \times_0 \mathsf{Lt}\ k\ n
\end{aligned}$$

for an appropriately defined less-than predicate $\mathsf{Lt} : (k : \mathsf{Nat}) \to (n : \mathsf{Nat}) \to \mathsf{U}$.

Inductive types. Inductive types are carried over from MLTT; the signature in Example 2 is a valid such instance. The only new rule is that the mode of the scrutinee in eliminators must be *at least as strong* as the mode of the output. In other words, we cannot pattern match on compile-time data at runtime. Besides that, there is a new axis of choices from the ability to mark parts of inductive types as erased. We can mark data arguments or recursive arguments as erased, or even entire constructors. This is explored in Section 4.

▶ **Example 5.** For booleans, given $x \overset{i}{:} \mathsf{Bool}$ and $a,\ b \overset{j}{:} A$ where $i \geq j$ we have

$$\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b \overset{j}{:} A\,.$$

This means that if we are in erased mode, we can perform case analysis on any boolean, but if we are in runtime mode, we can only perform case analysis on a runtime boolean.

2.1 Erasure as a phase distinction

When we transition from the surface language to the core language, the subusaging rule of erasure is explained through a *synthetic phase distinction*. The concept of "phase distinction" originates from Cardelli [11], later reformulated synthetically by Sterling and Harper [42].

A synthetic phase distinction is simply an *abstract* proposition #, meaning a propositional sort of our type theory to which we do not attach a specific truth value. We are allowed to bind proofs of # in contexts, but there is no way to produce a closed proof of #, or store proofs of # inside terms or types. If a context Γ contains the proposition #, we write $\# \in \Gamma$, which itself is a *decidable* proposition in the syntax.

▶ **Example 6.** Using $\triangleright$ for context extension and $\bullet$ for the empty context, we have:

$$\# \in (\bullet \triangleright x \overset{0}{:} \mathsf{Nat} \triangleright \# \triangleright y \overset{\omega}{:} \mathsf{Fin}\ x) \qquad \# \notin (\bullet \triangleright x \overset{0}{:} \mathsf{Nat} \triangleright y \overset{\omega}{:} \mathsf{Fin}\ x)$$

In total, we have three kinds of context extension: extending by a runtime term variable, extending by an erased term variable, or extending by a # proof variable.

We call the propositional sort # the *erasure marker*, because a context is considered erased when $\# \in \Gamma$. In such an erased context Γ, runtime and erased terms become interchangeable through two new constructors $\uparrow$ and $\downarrow$. The precise rules surrounding these are:

- If $a \overset{0}{:} A$ and $\# \in \Gamma$, then $\uparrow a \overset{\omega}{:} A$.
- If $a \overset{\omega}{:} A$ in context $\Gamma \triangleright \#$, then $\downarrow a \overset{0}{:} A$ in context Γ.
- $\uparrow$ and $\downarrow$ are mutual inverses up to definitional equality.

To provide an erased term, it suffices to provide a runtime term under the assumption of # – a uniqueness property similar to "every function is a lambda" and "every product is a pair". Crucially, # cannot be introduced or discharged in any other way (it is not a type!). The justification for the first point above is that $\# \in \Gamma$ holds only when we are already in a *subterm* of an erased term. This way, we emulate the *phase distinction requirement* of Cardelli that every subterm of a compile-time term should itself be compile-time.

This mechanism allows us to treat *any* runtime term as erased, eliminating the need to separately axiomatise the erased variant of a term when the runtime variant is already present in the theory. The decidability of $\# \in \Gamma$ means we can build an elaboration algorithm which inserts all $(\uparrow, \downarrow)$ coercions (Section 8). The high-level syntax we have presented thus far is the input to this elaboration.

3 Formal setup

In this section we formally develop type theory with erasure, assuming familiarity with category theory and in particular categories with families (CwFs) [13]. We work in a constructive intensional type theory with a bounded cumulative hierarchy of universes $\mathbf{Set}_\ell$ ($\ell \leq \omega + 1$), natural numbers $\mathbb{N}$, function extensionality, uniqueness of identity proofs, and quotient inductive-inductive types [4]. We write $\mathbf{Psh}_\ell(C)$ for the category of presheaves over C valued in $\mathbf{Set}_\ell$. In general, when we omit ℓ, we mean $\ell = \omega$. We also omit some equality transports for readability. When working internally to categories, we overload type-theoretic notation; $(x : A) \to B$ might denote a function type in a presheaf category [22] or in some theory of signatures (Section 3). We use $\mathcal{U}$ for internal universes, mentioning this explicitly when necessary. We write TT_0 for type theory with erasure and TT for ordinary Martin-Löf type theory. Rather than fixing type formers, we develop the theory modularly. For example, "TT with Π-types" denotes MLTT with only function types.

Generalised algebraic theories. A generalised algebraic theory (GAT) [12] is a description of a theory consisting of sorts, operations and equations which are allowed to appear in arbitrary order where dependency is permitted. Such a description can be neatly given by a context in a certain dependent type theory called the *GAT theory of signatures* [31]. This supports $\mathbb{1}$, Σ, equality types with UIP, a universe $\mathcal{U}$ for declaring sorts, and dependent function types Π with domain in $\mathcal{U}$, and Π_{ext} with an external type (e.g. $\mathbb{N}$) as domain. GATs can be used to describe type theories in an intrinsically well-formed fashion.

▶ **Example 7.** The GAT Cat_1 of a category with a terminal object is given by:

Con	$: \mathcal{U}$	id	$: \mathsf{Sub}\ \Gamma\ \Gamma$
Sub	$: \mathsf{Con} \to \mathsf{Con} \to \mathcal{U}$	$- \circ -$	$: \mathsf{Sub}\ \Phi\ \Delta \to \mathsf{Sub}\ \Gamma\ \Phi \to \mathsf{Sub}\ \Gamma\ \Delta$
$\bullet$	$: \mathsf{Con}$	$\mathsf{id}\circ$	$: \mathsf{id} \circ \sigma = \sigma$
ϵ	$: \mathsf{Sub}\ \Gamma\ \bullet$	$\circ\mathsf{id}$	$: \sigma \circ \mathsf{id} = \sigma$
$\epsilon\eta$	$: (\sigma : \mathsf{Sub}\ \Gamma\ \bullet) \to \sigma = \epsilon$	assoc	$: (\sigma \circ \tau) \circ \rho = \sigma \circ (\tau \circ \rho)$

▶ **Definition 8** (Σ-CwF). *A Σ-CwF is a CwF equipped with $\mathbb{1}$ and Σ types. A morphism of Σ-CwFs is a strict CwF morphism preserving $\mathbb{1}$ and Σ, inducing a category $\mathbf{CwF}_\Sigma \subseteq \mathbf{CwF}$.*

Each GAT G gives rise to a *freely generated* Σ-CwF, where base types are determined by the sorts of G, and base terms by the operations of G quotiented by the equations in G. This is the approach of Bocquet [8], which equips GATs with *functorial semantics* [32]. We identify G with its freely generated Σ-CwF. In this language, a model $\mathcal{M}$ of a GAT G is simply a Σ-CwF morphism $\mathcal{M} : G \to \mathbf{Set}$, with the standard Σ-CwF structure on $\mathbf{Set}$. To give a morphism out of G, it suffices to define its actions on the signature items of G.

▶ **Example 9.** A morphism $\mathcal{F} : \mathsf{Cat}_1 \to E$ is determined by a closed type $\mathcal{F}.\mathsf{Con}$ in E and a type $\mathcal{F}.\mathsf{Sub}\ \Gamma\ \Delta$ over $\Gamma : \mathcal{F}.\mathsf{Con}$ and $\Delta : \mathcal{F}.\mathsf{Con}$ in E, such that the terminal object, identity and composition are preserved.

Second-order generalised algebraic theories. Second-order generalised algebraic theories (SOGATs) [47, 8] are a generalisation of GATs which allow second-order binding to appear in signatures. The *SOGAT theory of signatures* extends the GAT theory of signatures with a subuniverse $\mathcal{U}_{\mathsf{R}} \subset \mathcal{U}$ and a dependent function type Π_{R} in $\mathcal{U}$ with domain in $\mathcal{U}_{\mathsf{R}}$.

▶ **Example 10.** The simplest interesting SOGAT is untyped lambda calculus with $\beta\eta$ laws:

$$\begin{aligned} &\mathsf{Tm} : \mathcal{U}_{\mathsf{R}} \\ &(\mathsf{lam}, \mathsf{app}, \beta, \eta) : (\mathsf{Tm} \to \mathsf{Tm}) \simeq \mathsf{Tm} \end{aligned} \tag{1}$$

The isomorphism $(\mathsf{Tm} \to \mathsf{Tm}) \simeq \mathsf{Tm}$ is shorthand for two operations lam, app and two coherence equations β, η. The forward direction lam implicitly uses Π_{R} to bind the second-order occurrence of Tm, while the backward direction app uses the first-order Π.

SOGATs are an even more convenient tool to describe type theories because they already include a notion of variable binding. This avoids the boilerplate of encoding variables and substitution manually, at least for *structural* theories (ones that do not involve linearity, contextual modalities or other constraints on variable usage).

▶ **Definition 11** ($(\Sigma, \Pi_{\mathsf{R}})$-$\mathsf{CwF}$)**.** *A* $(\Sigma, \Pi_{\mathsf{R}})$-CwF *is a* Σ*-CwF with a subpresheaf* $\mathsf{Ty}_{\mathsf{R}} \subseteq \mathsf{Ty}$ *of* representable types *closed under* $\mathbb{1}$ *and* Σ*, and a function type* Π_{R} *with domain in* Ty_{R}*, inducing a category* $\mathbf{CwF}_{\Sigma,\Pi_{\mathsf{R}}} \subseteq \mathbf{CwF}_{\Sigma}$.

Similar to before, a SOGAT can be viewed as a freely generated $(\Sigma, \Pi_{\mathsf{R}})$-CwF. This time, sorts in $\mathcal{U}_{\mathsf{R}}$ become the base types in Ty_{R}. A model $\mathcal{M}$ of a SOGAT S consists of a category $\mathcal{M}_\diamond$ with terminal object and a $(\Sigma, \Pi_{\mathsf{R}})$-CwF morphism $\mathcal{M} : S \to \mathbf{Psh}(\mathcal{M}_\diamond)$. The codomain $\mathbf{Psh}(\mathcal{M}_\diamond)$ has a standard $(\Sigma, \Pi_{\mathsf{R}})$-CwF structure where the representable types are presheaves with a "context extension" operation on $\mathcal{M}_\diamond$. This includes the types of $\mathcal{M}_\diamond$ if $\mathcal{M}_\diamond$ itself is a CwF, by the Yoneda embedding y. The category $\mathcal{M}_\diamond$ should be thought of as the underlying *category of contexts* of $\mathcal{M}$, where second-order binding in S is interpreted using context extension along representable types in $\mathcal{M}_\diamond$. Once again, to give a morphism out of S it suffices to define its actions on the signature items of S.

Correspondence between GAT and SOGAT models. Bocquet's approach to the model theory of SOGATs is to reduce them to GATs and then reuse the model theory of GATs. From a SOGAT S we can compute a GAT S^{fo} extending Cat_1, such that SOGAT models $\mathcal{M} : S \to \mathbf{Psh}(\mathcal{M}_\diamond)$ are in bijective correspondence with GAT models $\widetilde{\mathcal{M}} : S^{\mathsf{fo}} \to \mathbf{Set}$ where the Cat_1 part of S^{fo} maps to $\mathcal{M}_\diamond$. We often reuse the name $\mathcal{M}$ for $\widetilde{\mathcal{M}}$ as it is unambiguous to do so. The mapping $S \mapsto S^{\mathsf{fo}}$ essentially corresponds to the signature-based translation from SOGATs to GATs detailed by Kaposi and Xie [28]: it maps a SOGAT S to the GAT of a category with terminal object and the presheaf interpretation of S over it. This generates all the variable and substitution boilerplate machinery necessary to interpret the second-order binding in S in terms of first-order context extension. This mapping is *functorial*: a SOGAT morphism $S \to T$ yields a GAT morphism $S^{\mathsf{fo}} \to T^{\mathsf{fo}}$ that preserves the category structure.

▶ **Example 12.** Translating the SOGAT of untyped lambda calculus from Equation (1) produces a GAT extending Cat_1 with a family $\mathsf{Tm} : \mathsf{Con} \to \mathcal{U}$ of terms with substitution $-[-] : \mathsf{Tm}\ \Delta \to \mathsf{Sub}\ \Gamma\ \Delta \to \mathsf{Tm}\ \Gamma$ preserving identity and composition (in other words, a presheaf), a context extension operator $-\triangleright : \mathsf{Con} \to \mathsf{Con}$ representing the presheaf Tm via $\mathsf{Sub}\ \Gamma\ (\Delta\triangleright) \simeq \mathsf{Sub}\ \Gamma\ \Delta \times \mathsf{Tm}\ \Gamma$, and an isomorphism $(\mathsf{lam}, \mathsf{app}) : \mathsf{Tm}\ (\Gamma\triangleright) \simeq \mathsf{Tm}\ \Gamma$ encoding the second-order constructor lam using context extension.

Categories of models. For any two models $\mathcal{M}, \mathcal{N}$ of a GAT G, there is a function space model $\mathbf{Func}(\mathcal{M}, \mathcal{N})$ of natural transformations from $\mathcal{M}$ to $\mathcal{N}$, displayed over G. For example, a context displayed over Γ in $\mathsf{Func}(\mathcal{M}, \mathcal{N})$ is a function $\mathcal{M}\,\Gamma \to \mathcal{N}\,\Gamma$. A strict morphism

of models $\mathcal{M} \to \mathcal{N}$ is thus defined as a dependent Σ-CwF morphism $G \to \mathbf{Func}(\mathcal{M}, \mathcal{N})$. This yields the usual notion of GAT morphism that can be computed from signatures [31], and induces a category of G-models $\mathbf{Mod}(G)$. The syntax $\mathbf{0}_G$ of a GAT G is the initial G-model, which always exists and is given by a quotient inductive-inductive type; for any other G-model $\mathcal{M}$, we write $[\![-]\!]_{\mathcal{M}} : \mathbf{0}_G \to \mathcal{M}$ for the unique G-model morphism. The syntax $\mathbf{0}_S$ of a SOGAT S is simply $\mathbf{0}_{S^{fo}}$. A strict morphism of GATs $\mathcal{F} : G \to H$ yields a functor between categories of models $\mathcal{F}^* : \mathbf{Mod}(H) \to \mathbf{Mod}(G)$ by precomposition. Applying this to $\mathbf{0}_H$, we get a G-model $\mathcal{F}^*\ \mathbf{0}_H$. We often abuse notation and write $\mathcal{F}$ instead of $[\![-]\!]_{\mathcal{F}^*\mathbf{0}_H}$.

4 TT_0 as a SOGAT

Now we proceed to define TT_0 as a SOGAT and compute its GAT translation. The full definition is given in Figure 1, with the extension to inductive types described in Section 4.

$$
\begin{array}{ll}
\mathsf{Mode} \triangleq \{0, \omega\} & \Pi_i : (A : \mathsf{Ty}_\ell) \to (\mathsf{Tm}_i\ A \to \mathsf{Ty}_\ell) \to \mathsf{Ty}_\ell \\
\mathsf{Ty} : \mathbb{N} \to \mathcal{U} & (\mathsf{lam}, \mathsf{app}) : ((x : \mathsf{Tm}_i\ A) \to \mathsf{Tm}_\omega\ (B\ x)) \simeq \mathsf{Tm}_\omega\ (\Pi_i\ A\ B) \\
\mathsf{Tm} : \mathsf{Mode} \to \mathsf{Ty}_\ell \to \mathcal{U}_{\mathsf{R}} & \\
 & \Sigma_i : (A : \mathsf{Ty}_\ell) \to (\mathsf{Tm}_i\ A \to \mathsf{Ty}_\ell) \to \mathsf{Ty}_\ell \\
\# : \mathcal{U}_{\mathsf{R}} & (\mathsf{pair}, \mathsf{proj}) : ((x : \mathsf{Tm}_i\ A) \times \mathsf{Tm}_\omega\ (B\ x)) \simeq \mathsf{Tm}_\omega\ (\Sigma_i\ A\ B) \\
\#\text{-}\mathsf{prop} : (p, q : \#) \to p = q & \\
(\downarrow, \uparrow) : (\# \to \mathsf{Tm}_\omega\ A) \simeq \mathsf{Tm}_0\ A & \mathsf{U} : (\ell : \mathbb{N}) \to \mathsf{Ty}_{\ell+1} \\
 & (\mathsf{El}, \mathsf{code}) : \mathsf{Tm}_0\ \mathsf{U}_\ell \simeq \mathsf{Ty}_\ell
\end{array}
$$

Figure 1 The SOGAT defining TT_0 with Π, Σ, and universes ($\circlearrowleft$).

TT_0 is a theory involving three sorts: types Ty externally indexed by universe levels in $\mathbb{N}$, terms Tm externally indexed by modes $\{0, \omega\}$ and internally indexed by types, and the *erasure marker* $\#$, forced to be a proposition by $\#$-prop. The definitional isomorphism $(\downarrow, \uparrow)$ converts between modes: if $a \overset{0}{:} A$ and $p : \#$, then $\uparrow_p a \overset{\omega}{:} A$; if $a \overset{\omega}{:} A$ under assumption $p : \#$, then $\downarrow_{p.} a \overset{0}{:} A$. In $\uparrow_p a$, p is an argument, while in $\downarrow_{p.} a$, p is *bound* in a, shorthand for $\downarrow(\lambda p.\ a)$. Since we can always weaken $a : \mathsf{Tm}_\omega\ A$ to $\lambda\_.a : \# \to \mathsf{Tm}_\omega\ A$, we can convert ω terms to 0 terms in any context. By induction on i, we implicitly extend this to $\downarrow : \mathsf{Tm}_i\ A \to \mathsf{Tm}_0\ A$.

We include Σ and Π types whose introduction and elimination rules involve terms in mode ω, while type formation binds a term of mode i matching the domain mode. Universes have codes *in mode 0*. This induces an isomorphism $(\# \to \mathsf{Ty}_\ell) \simeq \mathsf{Ty}_\ell$ which allows us to convert between type families $\mathsf{Tm}_\omega\ A \to \mathsf{Ty}_\ell$ and $\mathsf{Tm}_0\ A \to \mathsf{Ty}_\ell$. Without universes we would have to include this isomorphism as a primitive. This yields an alternative equivalent formulation of Π_i and Σ_i where the binder in the type formation rule is always indexed by $\mathsf{Tm}_0\ A$, for example: $\Sigma_i : (A : \mathsf{Ty}_\ell) \to (\mathsf{Tm}_0\ A \to \mathsf{Ty}_\ell) \to \mathsf{Ty}_\ell$. Also, it is possible and practically desirable to include strict cumulativity [39] for universes, which we omit here.

The erased fragment. The theory only explicitly includes terms for the runtime fragment ω. Usually, presentations of theories with erasure will include a copy of terms in each mode. In our case, this is not necessary. By virtue of the isomorphism $(\downarrow, \uparrow)$, we automatically get the entire erased fragment "for free". For example, erased terms for function types at any

domain mode are derivable:

$$\mathsf{lam}_0 : ((x : \mathsf{Tm}_0\ A) \to \mathsf{Tm}_0\ (B\ x)) \to \mathsf{Tm}_0\ (\Pi_i\ A\ (B \circ \downarrow))$$
$$\mathsf{lam}_0\ f = \downarrow_p \mathsf{lam}\ (\lambda x.\ \uparrow_p (f\ \downarrow x))$$

$$\mathsf{app}_0 : \mathsf{Tm}_0\ (\Pi_i\ A\ B) \to (x : \mathsf{Tm}_0\ A) \to \mathsf{Tm}_0\ ((B \circ \uparrow)\ x)$$
$$\mathsf{app}_0\ t\ x = \downarrow_p \mathsf{app}\ (\uparrow_p t)\ (\uparrow_p x)$$

Here we use $- \circ \uparrow$ and $- \circ \downarrow$ for the two directions of $(\mathsf{Tm}_\omega\ A \to \mathsf{Ty}_\ell) \simeq (\mathsf{Tm}_0\ A \to \mathsf{Ty}_\ell)$. The isomorphism providing β and η for $(\mathsf{lam}, \mathsf{app})$ also holds for $(\mathsf{lam}_0, \mathsf{app}_0)$. The idea is that if the goal is of the form $\mathsf{Tm}_0\ X$, then we introduce $\downarrow_p$ and reduce the goal to $\mathsf{Tm}_\omega\ X$ while having access to p. Then we use whatever runtime term former to fulfil the goal. When such a runtime term former *expects* a $\mathsf{Tm}_\omega\ Y$ but all we have is a $\mathsf{Tm}_0\ Y$, we wrap it in $\uparrow_p$ using the p we bound earlier. We can derive the rest of the erased fragment this way. From now on, we use the subscript $_0$ as in lam_0 to denote the erased variant of some term former.

Inductive types. It is straightforward to include any indexed inductive type in TT_0. We only need to include it in mode ω, and then we can derive its erased fragment using the technique in Section 4. However, the main affordance of erasure is that we can choose for some constructor data, usually indices, to *always* be erased. For example, here is the inductive family of length-indexed vectors, where the length is considered erased data:

$$\mathsf{Vect} : \mathsf{Tm}_0\ \mathsf{Nat} \to \mathsf{Ty}_\ell \to \mathsf{Ty}_\ell$$
$$\mathsf{nil} : \mathsf{Tm}_\omega\ (\mathsf{Vect}\ \mathsf{zero}\ A)$$
$$\mathsf{cons} : \{k : \mathsf{Tm}_0\ \mathsf{Nat}\} \to \mathsf{Tm}_\omega\ A \to \mathsf{Tm}_\omega\ (\mathsf{Vect}\ k\ A) \to \mathsf{Tm}_\omega\ (\mathsf{Vect}\ (\mathsf{succ}_0\ k)\ A)$$
$$\begin{aligned}\mathsf{elim} : {} & (P : (n : \mathsf{Tm}_0\ \mathsf{Nat}) \to \mathsf{Tm}_0\ (\mathsf{Vect}\ n\ A) \to \mathsf{Ty}_\ell) \\ & \to \mathsf{Tm}_\omega\ (P\ \mathsf{zero}_0\ \mathsf{nil}_0) \\ & \to (\{k : \mathsf{Tm}_0\ \mathsf{Nat}\} \to (x : \mathsf{Tm}_\omega\ A) \to (xs : \mathsf{Tm}_\omega\ (\mathsf{Vect}\ k\ A)) \\ & \quad \to \mathsf{Tm}_\omega\ (P\ k\ xs) \to \mathsf{Tm}_\omega\ (P\ (\mathsf{succ}_0\ k)\ (\mathsf{cons}_0\ x\ xs))) \\ & \to \{n : \mathsf{Tm}_0\ \mathsf{Nat}\} \to (v : \mathsf{Tm}_\omega\ (\mathsf{Vect}\ n\ A)) \to \mathsf{Tm}_\omega\ (P\ n\ v)\end{aligned}$$

We have omitted type indexing and computation rules for brevity. Erased constructor arguments that appear in return indices can sometimes be converted to *relevant* arguments. A sufficient condition for this is Brady et al.'s forcing analysis [10], which can be applied here: if we make the n in the output above *relevant*, we can also make k in the cons method relevant because k can be computed at runtime by stripping succ from n. It is possible to extend code extraction (Definition 21) with such "forced" eliminators, and other optimisations in [10] like detagging.

We could include general W-types [24] in TT_0, but this would not afford us the most generality. This is because we have a choice of erased data both in the non-recursive and the recursive arguments. The former would be possible with standard W-types because we have mode-dependent Σ types, but the latter wouldn't. For example, consider the type

$$\mathsf{Nat0} : \mathsf{Ty}_\ell$$
$$\mathsf{zero0} : \mathsf{Tm}_\omega\ \mathsf{Nat0}$$
$$\mathsf{succ0} : \mathsf{Tm}_0\ \mathsf{Nat0} \to \mathsf{Tm}_\omega\ \mathsf{Nat0}$$

This cannot be encoded as a regular W-type because W-types encode *arities* of recursive arguments, but now there is also the axis of $0/\omega$ to choose from. Additionally, we might

want entire constructors to only exist in mode 0:

$$\mathsf{Bool}_{\omega\to\mathsf{true}} : \mathsf{Ty}_\ell$$
$$\mathsf{true} : \mathsf{Tm}_\omega\ \mathsf{Bool}_{\omega\to\mathsf{true}}$$
$$\mathsf{false} : \mathsf{Tm}_0\ \mathsf{Bool}_{\omega\to\mathsf{true}}$$

This is the type of booleans that are always true at runtime. So there is also an axis of $0/\omega$ for the return sort of each constructor. Altogether these "exotic" inductive types suggest that it would be useful to investigate *signatures* for inductive types with erasure to better understand this space of possibilities. We leave this to future work.

Compatibility with two-level type theory. Having formulated erasure as a SOGAT, we can now easily combine it with other features formulated as SOGATs. For example, we can form a theory $\mathsf{2LTT}_0$ of two-level type theory [30] with erasure. Aside from its use in HoTT [6], two-level type theory can be viewed as a type theoretic formulation of staging/metaprogramming. It allows us to control which binders are evaluated at compile-time versus runtime, existing in the same space of features as erasure – features that aim to improve the feasibility of dependent types for practical programming.

The general principle of two-level type theory is that some or all judgments in the object theory become type formers in the meta-theory. In this case, our object theory is type theory with erasure. Therefore, we promote Tm_ω and Tm_0 from sorts to meta-level types (but not $\#$, to retain the decidability of its presence in any context):

$$\mathsf{Ty}^{\mathsf{M}} : \mathcal{U} \qquad \Uparrow : (i : \mathsf{Mode}) \to \mathsf{Ty} \to \mathsf{Ty}^{\mathsf{M}}$$
$$\mathsf{Tm}^{\mathsf{M}} : \mathsf{Ty}^{\mathsf{M}} \to \mathcal{U}_{\mathsf{R}} \qquad (\langle - \rangle, \sim -) : \mathsf{Tm}^{\mathsf{M}}\ \Uparrow_i A \simeq \mathsf{Tm}_i\ A$$

Here the superscript M denotes the meta fragment, while the base theory corresponds to the object fragment. In the meta fragment, we have two separate lifting operations $\Uparrow_i$, one for each mode i. The erasure marker $\#$ and coercions $\downarrow/\uparrow$ remain unchanged from before. From them we can derive coercions in the meta level as well:

$$(\downarrow^{\mathsf{M}}, \uparrow^{\mathsf{M}}) : (\# \to \mathsf{Tm}^{\mathsf{M}}\ \Uparrow_\omega A) \simeq \mathsf{Tm}^{\mathsf{M}}\ \Uparrow_0 A$$

These are given by $\downarrow^{\mathsf{M}}\ f = \langle \downarrow_{p.}\ \sim(f\ p) \rangle$ and $\uparrow^{\mathsf{M}}_p\ x = \langle \uparrow_p\ \sim x \rangle$.

4.1 Generated first-order theory ($\mathcal{U}$)

Using the results from Section 3, we now compute the GAT specification of TT_0. This is the actual "type system" in the traditional sense, which includes contexts and variables. As with any SOGAT, we start with a category with terminal object described in Example 7.

Sorts. For each sort in the SOGAT, we get a presheaf on this category:

$$\mathsf{Ty} : \mathbb{N} \to \mathsf{Con} \to \mathcal{U} \qquad \mathsf{Tm} : \mathsf{Mode} \to (\Gamma : \mathsf{Con}) \to \mathsf{Ty}_\ell\ \Gamma \to \mathcal{U} \qquad \# \in - : \mathsf{Con} \to \mathcal{U}$$

There is also an operation $(x, y : \# \in \Gamma) \to x = y$ from #-prop that forces $\# \in -$ to be proposition-valued. We now see that $\# \in \Gamma$ (Section 2.1) is simply the presheaf generated by the sort $\#$ evaluated at a context Γ. Each of these presheaves comes with a substitution operation $A[\sigma]$, $t[\sigma]$, $\pi[\sigma]$ which respects $\circ$ and id.

Context extensions. The representability of mode-indexed terms and the erasure marker yields the context extension operations

$$- \rhd_i - : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}_\ell\ \Gamma \to \mathsf{Con} \qquad - \rhd \# : \mathsf{Con} \to \mathsf{Con}$$

with corresponding isomorphisms

$$\begin{aligned}((-,_i -),\ \mathsf{pq}_i) &: (\sigma : \mathsf{Sub}\ \Gamma\ \Delta) \times \mathsf{Tm}_i\ \Gamma\ A[\sigma] \simeq \mathsf{Sub}\ \Gamma\ (\Delta \rhd_i A)\\ ((-,_\# -),\ \mathsf{pq}_\#) &: \mathsf{Sub}\ \Gamma\ \Delta \times (\# \in \Gamma) \simeq \mathsf{Sub}\ \Gamma\ (\Delta \rhd \#)\end{aligned}$$

defining that substitutions can be extended by terms and erasure marker witnesses, and that terms in both modes as well as the erasure marker have the "0th" de Bruijn index by q and weakening by p. More explicitly, we get

$$\begin{aligned}\mathsf{p}_i &: \mathsf{Sub}\ (\Gamma \rhd_i A)\ \Gamma & \mathsf{p}_\# &: \mathsf{Sub}\ (\Gamma \rhd \#)\ \Gamma\\ \mathsf{q}_i &: \mathsf{Tm}_i\ (\Gamma \rhd_i A)\ A[\mathsf{p}_i] & \mathsf{q}_\# &: \# \in (\Gamma \rhd \#)\end{aligned}$$

which are derivable from the isomorphisms above, satisfying the usual CwF rules.

Erasure coercions. The defining isomorphism of # is presented in the GAT as

$$(\downarrow, \uparrow) : \mathsf{Tm}_\omega\ (\Gamma \rhd \#)\ A \simeq \mathsf{Tm}_0\ \Gamma\ A$$

relating runtime terms in a context extended by # to erased terms. The $\uparrow$ direction outputs a term in an extended context. We can also derive a form which stores the data of the substitution needed to make the context general:

$$\uparrow' : \# \in \Gamma \to \mathsf{Tm}_0\ \Gamma\ A \to \mathsf{Tm}_\omega\ \Gamma\ A \qquad \uparrow_p{}' t = (\uparrow t)[\mathsf{id}\ ,_\#\ \mathsf{p}_\#]$$

Just like before, we can extend $\downarrow$ to operate on a term in any mode i – now explicitly:

$$\downarrow_* : \mathsf{Tm}_i\ \Gamma\ A \to \mathsf{Tm}_0\ \Gamma\ A \qquad \downarrow_*\ \{i = 0\}\ t = t \qquad \downarrow_*\ \{i = \omega\}\ t = \downarrow (t[\mathsf{p}_\#])$$

Standard types. The rest of the theory is translated to a first-order representation that looks very similar to the usual CwF structures. For Π types we get

$$\begin{aligned}&\Pi_i : (A : \mathsf{Ty}_\ell\ \Gamma) \to (B : \mathsf{Ty}_\ell\ (\Gamma \rhd_i A)) \to \mathsf{Ty}_\ell\ \Gamma\\ &(\mathsf{lam}, \mathsf{app}) : \mathsf{Tm}_\omega\ (\Gamma \rhd_i A)\ B \simeq \mathsf{Tm}_\omega\ \Gamma\ (\Pi_i\ A\ B)\end{aligned}$$

matching the usual CwF formulation: the codomain B lives over the mode-i extension $\Gamma \rhd_i A$. We can write the less "categorical" application operator as

$$\mathsf{app}' : \mathsf{Tm}_\omega\ \Gamma\ (\Pi_i\ A\ B) \to (x : \mathsf{Tm}_i\ \Gamma\ A) \to \mathsf{Tm}_\omega\ \Gamma\ B[\langle x \rangle]$$

where $\langle - \rangle \triangleq (\mathsf{id},_i -) : \mathsf{Tm}_i\ \Gamma\ A \to \mathsf{Sub}\ \Gamma\ (\Gamma \rhd_i A)$ is the single term substitution. The derivable erased fragment (Section 4) is also carried over to the first-order GAT presentation.

5 Syntactic properties

In this section, we explore some properties of the first-order syntax of TT_0.

Zeroing. The first property we show is that erased terms do not depend on runtime variables. In particular, an erased term in Γ should uniquely correspond to an erased term in 0Γ, which is Γ with the modes of all the variables set to 0, and without any erasure markers $\#$. This verifies that the phase distinction truly prevents erased data from depending on runtime data. We do this by showing that the erased fragment of TT_0 is an appropriate model for the entire theory (the *zeroing* model), such that zeroing erased TT_0 terms is a bijection.

▶ **Definition 13** (Zeroing 𝒪). *The* zeroing $(\Sigma, \Pi_{\mathsf{R}})$*-CwF endomorphism* $0 : \mathsf{TT}_0 \to \mathsf{TT}_0$ *uses the erased fragment of* TT_0 *to implement the relevant fragment. It sets the mode of all terms to erased, and removes any erasure markers* $\#$*. It is defined by its action on the signature components:*

$$0.\mathsf{Ty}_\ell \triangleq \mathsf{Ty}_\ell \qquad 0.\mathsf{Tm}_0\ A \triangleq \mathsf{Tm}_0\ A$$
$$0.\# \triangleq \mathbb{1} \qquad 0.\mathsf{Tm}_\omega\ A \triangleq \mathsf{Tm}_0\ A$$

The rest of the signature is implemented by the erased fragment (Section 4). For example $0.\Pi_i\ A\ B \triangleq \Pi_i\ A\ B$, $0.\mathsf{lam}\ f \triangleq \mathsf{lam}_0\ f$ *and* $0.\mathsf{app}\ f\ x \triangleq \mathsf{app}_0\ f\ x$*. The* $\uparrow/\downarrow$ *become trivial.*

By acting on the syntax, this yields a morphism of $\mathsf{TT}_0^{\mathsf{fo}}$ models $[\![-]\!]_0 : \mathbf{0}_{\mathsf{TT}_0} \to 0^*\ \mathbf{0}_{\mathsf{TT}_0}$. We can compute the actions on syntactic sorts as $0_{\mathsf{Con}} : \mathsf{Con} \to \mathsf{Con}$, $0_{\mathsf{Ty}} : \mathsf{Ty}\ \Gamma \to \mathsf{Ty}\ 0\Gamma$, $0_{\mathsf{Tm}_\omega} : \mathsf{Tm}_\omega\ \Gamma\ A \to \mathsf{Tm}_0\ 0\Gamma\ 0A$, $0_{\mathsf{Tm}_0} : \mathsf{Tm}_0\ \Gamma\ A \to \mathsf{Tm}_0\ 0\Gamma\ 0A$, and $0_\# : \# \in \Gamma \to \mathbb{1}$.

▶ **Theorem 14** (Types and erased terms need nothing 𝒪). *In the syntax of* TT_0 *there exists a substitution* $\Uparrow : \mathsf{Sub}\ \Gamma\ 0\Gamma$ *which becomes the identity under zeroing, such that* $(0A)[\Uparrow] = A$ *for types and* $(0a)[\Uparrow] = \downarrow_\star a$ *for terms. This induces natural isomorphisms:*

$$\mathsf{Ty}_\ell\ 0\Gamma \simeq \mathsf{Ty}_\ell\ \Gamma \qquad \mathsf{Tm}_0\ 0\Gamma\ 0A \simeq \mathsf{Tm}_0\ \Gamma\ A$$

Conservativity over type theory. We can also characterise the relationship between TT_0 and TT. In particular, we would like to ensure that erasure does not allow the production of any exotic terms that are not possible in ordinary type theory. The formal notion of this condition is *conservativity*: if there exists a proof of a TT-theorem in TT_0, then a proof also exists in TT. This can be shown by constructing bidirectional interpretations.

▶ **Definition 15** (TT_0 to TT 𝒪). *The morphism* $\llcorner - \lrcorner : \mathsf{TT}_0 \to \mathsf{TT}$ *is constructed by using* TT *types and terms to implement* TT_0 *types and terms (both runtime and erased). In this model, mode annotations and the erasure marker are forgotten:*

$$\llcorner - \lrcorner.\mathsf{Ty}_\ell \triangleq \mathsf{Ty}_\ell \qquad \llcorner - \lrcorner.\mathsf{Tm}_0\ A \triangleq \mathsf{Tm}\ A$$
$$\llcorner - \lrcorner.\# \triangleq \mathbb{1} \qquad \llcorner - \lrcorner.\mathsf{Tm}_\omega\ A \triangleq \mathsf{Tm}\ A$$

▶ **Definition 16** (TT to TT_0 𝒪). *Conversely, the morphism* $\ulcorner - \urcorner : \mathsf{TT} \to \mathsf{TT}_0$ *is constructed by using the erased fragment to implement the structure of* TT*:*

$$\ulcorner - \urcorner.\mathsf{Ty}_\ell \triangleq \mathsf{Ty}_\ell \qquad \ulcorner - \urcorner.\mathsf{Tm}\ A \triangleq \mathsf{Tm}_0\ A$$

The composition $\ulcorner \llcorner - \lrcorner \urcorner$ is almost equivalent to zeroing 0 (but doesn't preserve Π and Σ modes), while $\llcorner \ulcorner - \urcorner \lrcorner$ is the identity. In the terminology of Bocquet [8, p. 40], $\llcorner - \lrcorner$ is a *trivial fibration*, yielding the following properties:

▶ **Theorem 17** (Erased conservativity of TT_0 over TT 𝔘). *TT_0's erased fragment is conservative over TT: there is a surjective natural map $\mathbf{0}_{\mathsf{TT}_0}.\mathsf{Tm}_0\ \ulcorner\Gamma\urcorner\ \ulcorner A\urcorner \to \mathbf{0}_{\mathsf{TT}}.\mathsf{Tm}\ \Gamma\ A$.*

▶ **Corollary 18** (Runtime conservativity of TT_0 over TT 𝔘). *TT_0's runtime fragment is conservative over TT: given a TT_0 context Γ' and type A' such that $0\Gamma' = \ulcorner\Gamma\urcorner$ and $0A' = \ulcorner A\urcorner$, there is a natural map $\mathbf{0}_{\mathsf{TT}_0}.\mathsf{Tm}_\omega\ \Gamma'\ A' \to \mathbf{0}_{\mathsf{TT}}.\mathsf{Tm}\ \Gamma\ A$.*

We might be tempted to ask for a stronger conservativity result for erased terms, namely that this map is actually an isomorphism. This is not possible if we have mode-aware binder types. For example, consider for each mode i the term $\mathsf{pair}\ (\mathsf{code}\ (\Pi_i\ \mathsf{Nat}\ \mathsf{Nat}))\ (\mathsf{lam}_0\ \mathsf{q}_0) : \mathsf{Tm}_0\ \bullet\ (\Sigma_0\ \mathsf{U}\ (\mathsf{El}\ \mathsf{q}_0))$. For either choice of i we get the same TT term, so $\llcorner - \lrcorner$ is not injective even on erased terms.

6 Standard models

The standard semantics of Martin-Löf type theory are in **Set**, and can be generalised to presheaf categories [22] or Grothendieck toposes [18]. Now we explore standard models for TT_0, first at the level of sets, and then briefly at the generality of Grothendieck toposes. The utility of such models, besides showing that the theory can be modelled by known and common mathematical objects, lies in their use when proving metatheorems via gluing [27].

The involvement of a phase distinction in TT_0 requires the standard semantics to take place not just in plain sets, but rather in some kind of *phase-separated* sets. The isomorphism $(\# \to \mathsf{Tm}_\omega\ A) \simeq \mathsf{Tm}_0\ A$ shows us that in the semantics we must have an object $\#$ such that exponentiating by $\#$ *progresses* the phase from runtime to erased. It is already known that languages with phase separations have semantics in glued categories [40]. A glued category is a comma category of the form $\mathsf{Id}_C \downarrow F$ for a functor $F : D \to C$. In the simplest case, we take $C = D = \mathbf{Set}$ and $F = \mathsf{Id}_{\mathbf{Set}}$. This yields the arrow category of **Set**, which is the category of presheaves over the interval $0 \to 1$, and also equivalent to the category of families of sets $\mathsf{Fam}(\mathbf{Set})$. We choose to work with $\mathsf{Fam}(\mathbf{Set})$, using its internal language when convenient.

The category $\mathsf{Fam}(\mathbf{Set})$ serves as the simplest standard model of TT_0, which we denote by $\mathcal{S}$. For a family $(X_0 : \mathbf{Set}) \times (X_1 : X_0 \to \mathbf{Set})$, the base X_0 stores the erased data, and the fibers store the runtime data. Consider the object $\phi \triangleq (\mathbb{1}, \lambda\_.\ \mathbb{0})$, the family with a single empty fiber. It is a proposition in the sense that any two maps into ϕ are equal. Exponentiation of $X = (X_0, X_1)$ by ϕ acts as $(\phi \to X) \simeq (X_0,\ \lambda\_.\ \mathbb{1})$, so maps out of ϕ isolate the base component, trivialising the fibers. This suggests that the erasure marker $\#$ should be interpreted as ϕ. Because we are now in a semantic setting, we can interpret erased terms directly as maps from ϕ to runtime terms: $\mathcal{S}.\mathsf{Tm}_0\ A = (\phi \to \mathcal{S}.\mathsf{Tm}_\omega\ A)$.

▶ **Definition 19** (P-modal). *Given a proposition P, an object X is* P-modal *if the weakening map $\mathsf{p} : X \to (P \to X)$ defined by $\mathsf{p}\ x\ \_ = x$ is an isomorphism.*

In $\mathsf{Fam}(\mathbf{Set})$, an object is ϕ-modal if its fibers are contractible. A consequence of this approach is that any part of the language that is erased should be ϕ-modal. This notably includes universes. The standard Hofmann-Streicher [23] universe construction can be performed in $\mathsf{Fam}(\mathbf{Set})$, yielding the universe $\mathcal{U}_\ell \triangleq (\mathbf{Set}_\ell, \lambda A.\ A \to \mathbf{Set}_\ell)$ for $\ell < \omega$. However, this is not ϕ-modal, since its fibers are not contractible. We need an alternative universe construction. Let us suggestively denote this by $\sqrt[\phi]{\mathcal{U}_\ell}$ at level ℓ. We will revisit this notation in Section 6. For the $\mathsf{El}/\mathsf{code}$ isomorphism of TT_0, we have:

$$\mathcal{S}.\mathsf{U}_\ell \triangleq \sqrt[\phi]{\mathcal{U}_\ell} \qquad \mathcal{S}.\mathsf{code}\ (A : \sqrt[\phi]{\mathcal{U}_\ell}) \triangleq \mathsf{p}\ A : \phi \to \sqrt[\phi]{\mathcal{U}_\ell} \qquad \mathcal{S}.\mathsf{El}\ (c : \phi \to \sqrt[\phi]{\mathcal{U}_\ell}) \triangleq \boxed{?} : \sqrt[\phi]{\mathcal{U}_\ell}$$

An assignment to $\boxed{?}$ such that El and code form an isomorphism is possible when $\sqrt[\phi]{\mathcal{U}_\ell}$ is ϕ-modal. Luckily, $\mathsf{Fam}(\mathbf{Set})$ supports a universe which is ϕ-modal:

$$\sqrt[\phi]{\mathcal{U}_\ell} \triangleq ((A : \mathbf{Set}_\ell) \times (A \to \mathbf{Set}_\ell), \lambda\_.\mathbb{1})$$

The decoding map of this universe is of the form $[-] : \sqrt[\phi]{\mathcal{U}_\ell} \to \mathcal{U}_\ell$, and is defined by first projection for the base, and second projection for the fibers. It supports all base types of $\mathcal{U}_\ell$, and is closed under dependent products and sums. We call this universe *squashed* because it is ϕ-modal but still retains all the original structure. Now we are ready to define the full model:

▶ **Definition 20** ($\mathsf{Fam}(\mathbf{Set})$ model of TT_0 🗘). *The standard model of* TT_0 *in families of sets is* $\mathcal{S} : \mathsf{TT}_0 \to \mathbf{Psh}_{\omega+1}(\mathsf{Fam}(\mathbf{Set}))$*, given by:*

$$\begin{array}{llll} \mathcal{S}.\mathsf{Ty}_\ell & \triangleq \sqrt[\phi]{\mathcal{U}_\ell} & \mathcal{S}.\mathsf{Tm}_0\ A & \triangleq \phi \to [A] \\ \mathcal{S}.\# & \triangleq \phi & \mathcal{S}.\mathsf{Tm}_\omega\ A & \triangleq [A] \end{array}$$

For erased functions we interpret $\mathcal{S}.\Pi_0\ A\ B \triangleq (a : (\phi \to A)) \to B\ a$ *and for runtime functions we interpret* $\mathcal{S}.\Pi_\omega\ A\ B \triangleq (a : A) \to B\ (\mathsf{p}\ a)$*, both using the function type in* $\sqrt[\phi]{\mathcal{U}_\ell}$*. Because* $B : (\phi \to A) \to \sqrt[\phi]{\mathcal{U}_\ell}$ *in both cases, we must use the weakening map* p *in the* ω *case to "forget" the runtime data of* a*.*

The Fam(Set) model, explicitly (🗘). We expand $\mathcal{S}$ in first-order morphism form: each context Γ is interpreted as a set $[\![\Gamma]\!]^0_{\mathcal{S}} : \mathbf{Set}$ (erased phase) and a family $[\![\Gamma]\!]^1_{\mathcal{S}} : [\![\Gamma]\!]^0_{\mathcal{S}} \to \mathbf{Set}$ (runtime phase). A substitution $\Gamma \to \Delta$ is a function $\sigma_0 : [\![\Gamma]\!]^0_{\mathcal{S}} \to [\![\Delta]\!]^0_{\mathcal{S}}$ (erased) and a family of functions $\sigma_1 : \forall\gamma_0.\ [\![\Gamma]\!]^1_{\mathcal{S}}\ \gamma_0 \to [\![\Delta]\!]^1_{\mathcal{S}}\ (\sigma_0\ \gamma_0)$ (runtime), displayed over the erased function. An ℓ-type in context Γ is interpreted as a family of ℓ-sets $[\![\Gamma]\!]^0_{\mathcal{S}} \to (A_0 : \mathbf{Set}_\ell) \times (A_1 : A_0 \to \mathbf{Set}_\ell)$ indexed only over $[\![\Gamma]\!]^0_{\mathcal{S}}$. This is the result of expanding a map into the squashed universe. An erased term of type A in context Γ is interpreted as a section of the erased components $(\gamma_0 : [\![\Gamma]\!]^0_{\mathcal{S}}) \to [\![A]\!]^0_{\mathcal{S}}\ \gamma_0$, while a runtime term is interpreted as a full section $(\sigma_0 : (\gamma_0 : [\![\Gamma]\!]^0_{\mathcal{S}}) \to [\![A]\!]^0_{\mathcal{S}}\ \gamma_0) \times (\sigma_1 : \forall\gamma_0.\ [\![\Gamma]\!]^1_{\mathcal{S}}\ \gamma_0 \to [\![A]\!]^1_{\mathcal{S}}\ (\sigma_0\ \gamma_0))$. The sort $\# \in \Gamma$ is interpreted as $[\![\Gamma]\!]_{\mathcal{S}} \to \phi$ which is equivalent to $\forall\gamma_0.\ [\![\Gamma]\!]^1_{\mathcal{S}}\ \gamma_0 \to \mathbb{0}$; the assertion that the runtime part of the context is uninhabited. Finally, we can compute the context formers as:

$$\begin{array}{lll} [\![\bullet]\!]_{\mathcal{S}} & \triangleq \mathbb{1} & \simeq (\mathbb{1}, \lambda\_.\ \mathbb{1}) \\ [\![\Gamma \rhd_0 A]\!]_{\mathcal{S}} & \triangleq (\gamma : [\![\Gamma]\!]_{\mathcal{S}}) \times (\phi \to [\![A]\!]_{\mathcal{S}}\ (\mathsf{p}\ \gamma)) & \simeq ((\gamma_0 : [\![\Gamma]\!]^0_{\mathcal{S}}) \times [\![A]\!]^0_{\mathcal{S}}\ \gamma_0,\ \lambda(\gamma_0, a_0).\ [\![\Gamma]\!]^1_{\mathcal{S}}\ \gamma_0) \\ [\![\Gamma \rhd_\omega A]\!]_{\mathcal{S}} & \triangleq (\gamma : [\![\Gamma]\!]_{\mathcal{S}}) \times [\![A]\!]_{\mathcal{S}}\ (\mathsf{p}\ \gamma) & \simeq ((\gamma_0 : [\![\Gamma]\!]^0_{\mathcal{S}}) \times [\![A]\!]^0_{\mathcal{S}}\ \gamma_0, \\ & & \quad \lambda(\gamma_0, a_0).\ (\gamma_1 : [\![\Gamma]\!]^1_{\mathcal{S}}\ \gamma_0) \times [\![A]\!]^1_{\mathcal{S}}\ \gamma_0\ a_0) \\ [\![\Gamma \rhd \#]\!]_{\mathcal{S}} & \triangleq [\![\Gamma]\!]_{\mathcal{S}} \times \phi & \simeq ([\![\Gamma]\!]^0_{\mathcal{S}}, \lambda\_.\ \mathbb{0}) \end{array}$$

Erased context extension only extends the erased part of the context, runtime context extension extends both parts, and adding $\#$ makes the runtime part empty.

Models in Grothendieck toposes. The construction of $\mathcal{S}$ can be generalised beyond $\mathsf{Fam}(\mathbf{Set})$, to an arbitrary Grothendieck topos $\mathbf{G}$ that supports squashed universes. Gratzer, Shulman and Sterling [18] have shown (classically) that any Grothendieck topos $\mathbf{G}$ admits a lifting of $\{\mathbf{Set}_\ell\}_{\ell<\omega}$ where each $\mathcal{U}_\ell$ contains all ℓ-small type families; Streicher [43] showed this constructively for presheaf toposes. So all Grothendieck toposes support universes. Which support squashed universes? We have written $\sqrt[\phi]{\mathcal{U}}$ to imply that $\sqrt[\phi]{-}$ is a functor.

For $\mathsf{Fam}(\mathbf{Set})$, it takes an object $X \triangleq (X_0, X_1)$ to $((x : X_0) \times X_1\ x,\ \lambda\_.\mathbb{1})$. It is uniquely characterised by the fact that it is the *right adjoint* to exponentiation by ϕ:

$$(\phi \to -) \dashv \sqrt[\phi]{-}$$

This construction has been studied in the context of type theory before [33, 36, 41, 37], frequently denoted by the square root $\sqrt{-}$ symbol. When a proposition ϕ has a right adjoint, it is called *tiny*. Therefore, if $\mathbf{G}$ has a tiny proposition ϕ, it supports squashed universes with respect to ϕ. This has been observed by Sterling [38] for essentially the same purpose. This is not an overly restrictive condition either: in a presheaf topos $\mathbf{Psh}(C)$, a representable $\mathsf{y}X$ is tiny whenever C has products with X. As a result, any Grothendieck topos with a tiny proposition supports a model of TT_0. We leave spelling out the details of the general construction for future work. This, along with the satisfaction of the realignment axiom [18], would justify the use of synthetic Tait computability [40] for the metatheory of TT_0.

7 Code extraction

The main purpose of TT_0 is to provide a practical language for programming with dependent types, so it is useful to be able to extract executable code from TT_0 programs. In particular, the code extraction process should erase all erased terms, and preserve the computational behaviour of runtime terms. In this section, we show how to extract code from TT_0 programs by interpreting into a *presheaf model* of TT_0 over the untyped lambda calculus.

▶ **Definition 21** (Untyped lambda calculus)**.** *The untyped lambda calculus λ quotiented by $\beta\eta$-equality is a SOGAT given by Equation* (1)*. Its initial GAT model is the CwF $\mathbf{0}_\lambda$, where contexts are natural numbers, substitutions $\mathbf{0}_\lambda.\mathsf{Sub}\ n\ m$ are m-tuples of lambda terms with n free variables, types are trivial (a single type $\star$), and terms $\mathbf{0}_\lambda.\mathsf{Tm}\ n$ are untyped lambda terms with n free variables [13]. This CwF supports Π types by lambda abstraction and application, and (positive) Σ types, natural numbers, and other data types by Church encoding.*

▶ **Definition 22** (Code extraction model of TT_0 🏆)**.** *The code extraction model $\mathcal{E}$ interprets TT_0 into the base category of presheaves over the syntax of the untyped lambda calculus, $\mathbf{Psh}(\mathbf{0}_\lambda)$, meaning it is a (Σ, Π_R)-CwF morphism $\mathcal{E} : \mathsf{TT}_0 \to \mathbf{Psh}_{\omega+1}(\mathbf{Psh}(\mathbf{0}_\lambda))$ where:*

$$\mathcal{E}.\mathsf{Ty}_\ell \triangleq \mathbb{1} \qquad \mathcal{E}.\mathsf{Tm}_0\ A \triangleq \mathbb{1}$$
$$\mathcal{E}.\# \triangleq \mathbb{0} \qquad \mathcal{E}.\mathsf{Tm}_\omega\ A \triangleq \mathsf{y}\mathbf{0}_\lambda.\mathsf{Tm}$$

We interpret the runtime *(mode ω) Π and Σ as the corresponding Church-encoded untyped structures in $\mathbf{Psh}(\mathbf{0}_\lambda)$, while the* erased *ones disappear. For example, we have*

$$\mathcal{E}.\Pi_0\ A\ B \triangleq \star \quad \textit{(all types are trivial)} \qquad \mathcal{E}.\mathsf{lam}_0\ t \triangleq t\ \star \qquad \mathcal{E}.\mathsf{app}_0\ f\ a \triangleq f$$

The erasure coercions $\uparrow$ and $\downarrow$ reduce to the ex falso quodlibet *principle and the terminal map respectively. Universes disappear as well, since they only exist in the erased fragment.*

The double presheaf codomain is needed because we map the representable sort $\#$ to $\mathbb{0}$ which is not representable in $\mathbf{Psh}(\mathbf{0}_\lambda)$ (there is no $\mathbf{0}_\lambda$-context n such that $\mathsf{y}n \simeq \mathbb{0}$).[1] Upon unfolding the GAT model $\widetilde{\mathcal{E}} : \mathsf{TT}_0^{\mathsf{fo}} \to \mathbf{Set}$ corresponding to the above, we can compute that contexts are $\mathbf{0}_\lambda$-presheaves, where context extensions are interpreted as:

$$[\![\bullet]\!]_\mathcal{E}\ n = \mathbb{1} \qquad [\![\Gamma \rhd_0 A]\!]_\mathcal{E}\ n = [\![\Gamma]\!]_\mathcal{E}\ n \times \mathbb{1}\ (\simeq [\![\Gamma]\!]_\mathcal{E}\ n)$$
$$[\![\Gamma \rhd_\omega A]\!]_\mathcal{E}\ n = [\![\Gamma]\!]_\mathcal{E}\ n \times \mathbf{0}_\lambda.\mathsf{Tm}\ n \qquad [\![\Gamma \rhd \#]\!]_\mathcal{E}\ n = [\![\Gamma]\!]_\mathcal{E}\ n \times \mathbb{0}\ (\simeq \mathbb{0})$$

[1] Alternatively, we could form a *higher-order model*, followed by contextualisation [9] to get a GAT model in $\mathbf{Psh}(\mathbf{0}_\lambda)$.

Up to isomorphism, adding an erased variable does nothing, adding a runtime variable adds a lambda term, and adding an erasure marker makes the context uninhabited.

To extract a program from a closed TT_0 term $t : \mathbf{0}_{\mathsf{TT}_0}.\mathsf{Tm}_\omega \bullet A$, we interpret it in the code extraction model to get a closed lambda term:

$$[\![t]\!]_{\mathcal{E}} : \mathcal{E}.\mathsf{Tm}_\omega\ [\![\bullet]\!]_{\mathcal{E}}\ [\![A]\!]_{\mathcal{E}} = \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathbb{1}, \mathbf{0}_\lambda.\mathsf{Tm}) \simeq \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathsf{y}0, \mathbf{0}_\lambda.\mathsf{Tm}) \simeq \mathbf{0}_\lambda.\mathsf{Tm}\ 0\,.$$

To interpret open terms, we observe that syntactic contexts which do not contain an erasure marker are representable in $\mathbf{Psh}(\mathbf{0}_\lambda)$.

▶ **Lemma 23.** *If $\# \notin \Gamma$ (in TT_0 syntax), then $[\![\Gamma]\!]_{\mathcal{E}}$ is representable – there is a natural number c_Γ which counts the runtime bindings in Γ, satisfying*

$$[\![\Gamma]\!]_{\mathcal{E}} \simeq \mathsf{y}c_\Gamma\,.$$

Proof. By induction on contexts $\Gamma : \mathbf{0}_{\mathsf{TT}_0}.\mathsf{Con}$. ◀

▶ **Corollary 24.** *For $\# \notin \Gamma$, there is an extraction map $|-| : \mathbf{0}_{\mathsf{TT}_0}.\mathsf{Tm}_\omega\ \Gamma\ A \to \mathbf{0}_\lambda.\mathsf{Tm}\ c_\Gamma$.*

Proof.

$$\begin{aligned}[\![t]\!]_{\mathcal{E}} :\ \mathcal{E}.\mathsf{Tm}_\omega\ [\![\Gamma]\!]_{\mathcal{E}}\ [\![A]\!]_{\mathcal{E}} &= \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}([\![\Gamma]\!]_{\mathcal{E}}, \mathbf{0}_\lambda.\mathsf{Tm}) \\ &\simeq \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathsf{y}c_\Gamma, \mathbf{0}_\lambda.\mathsf{Tm}) \\ &\simeq \mathbf{0}_\lambda.\mathsf{Tm}\ c_\Gamma\,.\end{aligned}$$

◀

This is the "purest" code extraction model we can formulate; in practice, we would choose a richer untyped target that includes primitives for pairs and inductive types (and one that would support full negative pairs), but its construction would be entirely analogous.

7.1 Correctness of code extraction

To show that code extraction preserves the computational behaviour of TT_0 programs, we set up a logical relation between the code extraction model $\mathcal{E}$ (Definition 21) and the logical interpretation in **Set**. We do so by building a model $\mathsf{Gl}(F)$ of TT_0 extended with natural numbers Nat in a glued category. This model is given by gluing along a morphism of TT_0-models $F : \mathbf{0}_{\mathsf{TT}_0} \to \mathcal{S}$. It is the result of combining two other morphisms. The *first* morphism is the composite

$$\mathbf{0}_{\mathsf{TT}_0} \xrightarrow{[\![-]\!]_{\mathcal{E}}} \mathcal{E} \xrightarrow{P \mapsto P\,0} \mathcal{S}_{\mathbf{Set}}$$

which evaluates a $\mathbf{0}_\lambda$-presheaf produced by code extraction at the empty context 0 (in other words, the global sections pseudo-morphism [27]). Its target is $\mathcal{S}_{\mathbf{Set}}$, the standard model of TT_0 valued in **Set** rather than $\mathsf{Fam}(\mathbf{Set})$ with $\#$ interpreted by the singleton set $\mathbb{1}$. The *second* morphism is the interpretation morphism $[\![-]\!]_{\mathcal{S}_{\mathbf{Set}}} : \mathbf{0}_{\mathsf{TT}_0} \to \mathcal{S}_{\mathbf{Set}}$ itself. These two define the morphism F, valued in the standard $\mathsf{Fam}(\mathbf{Set})$ model $\mathcal{S}$, via:

$$\mathbf{0}_{\mathsf{TT}_0} \xrightarrow{\Gamma \mapsto ([\![\Gamma]\!]_{\mathcal{S}_{\mathbf{Set}}}, \lambda\_.\ [\![\Gamma]\!]_{\mathcal{E}}\,0)} \mathcal{S}$$

sending a context Γ to the family $F\Gamma \triangleq ([\![\Gamma]\!]_{\mathcal{S}_{\mathbf{Set}}}, \lambda\_.\ [\![\Gamma]\!]_{\mathcal{E}}\ 0)$. The standard **Set** interpretation is at the base of each object, and the code extraction is at the fibers. From F, we can construct a *displayed* TT_0 model $\mathsf{Gl}(F)$ analogously to the construction of Kaposi et al. [27].

The underlying category of $\mathsf{Gl}(F)$ is $\mathsf{Fam}(\mathbf{Set}) \downarrow F$ (where F really means the *underlying functor* between categories of contexts). Each context in $\mathsf{Gl}(F)$ consists of:

- a syntactic context $\Gamma : \mathbf{0}_{\mathsf{TT}_0}.\mathsf{Con}$
- a "base" predicate $\Gamma_0^{\approx} : [\![\Gamma]\!]_{\mathcal{S}_{\mathbf{Set}}} \to \mathbf{Set}_\omega$
- a "fiber" predicate $\Gamma_1^{\approx} : (\gamma_{\mathcal{S}} : [\![\Gamma]\!]_{\mathcal{S}_{\mathbf{Set}}}) \to \Gamma_0^{\approx}\ \gamma_{\mathcal{S}} \to [\![\Gamma]\!]_{\mathcal{E}}\ 0 \to \mathbf{Set}_\omega$

This comes with an evident projection morphism $(\Gamma, \Gamma_0^{\approx}, \Gamma_1^{\approx}) \mapsto \Gamma$ into the syntax $\mathbf{0}_{\mathsf{TT}_0}$, which by initiality has a section. This can be approximately thought of as a single binary relation that relates the set interpretation with code extraction. The nuance is that, in order to handle universes correctly, which are erased but must still carry logical predicate data, we need a base predicate $\Gamma_0^{\approx}$ which exists even in erased contexts to store them.

The sorts of the displayed model correspond to the induction motives of the logical relation, which are presented in Figure 2. This can be thought of as the logical relation version of the

$$
\begin{aligned}
&(\Gamma : \mathsf{Con})^{\approx} && : (\Gamma_0^{\approx} : [\![\Gamma]\!]_{\mathcal{S}_{\mathbf{Set}}} \to \mathbf{Set}_\omega) \times (\Gamma_1^{\approx} : \forall \gamma_{\mathcal{S}}.\ [\![\Gamma]\!]_{\mathcal{E}}\ 0 \to \Gamma_0^{\approx}\ \gamma_{\mathcal{S}} \to \mathbf{Set}_\omega)\\
&(\sigma : \mathsf{Sub}\ \Gamma\ \Delta)^{\approx} && : (\sigma_0^{\approx} : \forall \gamma_{\mathcal{S}}.\ \Gamma_0^{\approx}\ \gamma_{\mathcal{S}} \to \Delta_0^{\approx}\ ([\![\sigma]\!]_{\mathcal{S}_{\mathbf{Set}}}\ \gamma_{\mathcal{S}}))\\
&&& \quad \times (\sigma_1^{\approx} : \forall \gamma_{\mathcal{E}}, \gamma_0.\ \Gamma_1^{\approx}\ \gamma_{\mathcal{E}}\ \gamma_0 \to \Delta_1^{\approx}\ ([\![\sigma]\!]_{\mathcal{E}}\ 0\ \gamma_{\mathcal{E}})\ (\sigma_0^{\approx}\ \gamma_0))\\
&(A : \mathsf{Ty}_\ell\ \Gamma)^{\approx} && : (A_0^{\approx} : \forall \gamma_{\mathcal{S}}.\ \Gamma_0^{\approx}\ \gamma_{\mathcal{S}} \to [\![A]\!]_{\mathcal{S}_{\mathbf{Set}}}\ \gamma_{\mathcal{S}} \to \mathbf{Set}_\ell)\\
&&& \quad \times (A_1^{\approx} : \forall \gamma_0, a_{\mathcal{S}}.\ \mathbf{0}_\lambda.\mathsf{Tm}\ 0 \to A_0^{\approx}\ \gamma_0\ a_{\mathcal{S}} \to \mathbf{Set}_\ell)\\
&(a : \mathsf{Tm}_0\ \Gamma\ A)^{\approx} && : \forall \gamma_{\mathcal{S}}.\ (\gamma_0 : \Gamma_0^{\approx}\ \gamma_{\mathcal{S}}) \to A_0^{\approx}\ \gamma_0\ ([\![a]\!]_{\mathcal{S}_{\mathbf{Set}}}\ \gamma_{\mathcal{S}})\\
&(a : \mathsf{Tm}_\omega\ \Gamma\ A)^{\approx} && : (a_0^{\approx} : \forall \gamma_{\mathcal{S}}.\ (\gamma_0 : \Gamma_0^{\approx}\ \gamma_{\mathcal{S}}) \to A_0^{\approx}\ \gamma_0\ ([\![a]\!]_{\mathcal{S}_{\mathbf{Set}}}\ \gamma_{\mathcal{S}}))\\
&&& \quad \times (a_1^{\approx} : \forall \gamma_{\mathcal{E}}, \gamma_0.\ \Gamma_1^{\approx}\ \gamma_{\mathcal{E}}\ \gamma_0 \to A_1^{\approx}\ ([\![a]\!]_{\mathcal{E}}\ 0\ \gamma_{\mathcal{E}})\ (a_0^{\approx}\ \gamma_0))\\
&(p : \# \in \Gamma)^{\approx} && : [\![\Gamma]\!]_{\mathcal{E}}\ 0 \to \mathbb{0}
\end{aligned}
$$

Figure 2 The motives of the logical relation for code extraction correctness.

model described in Section 6; the interpretation of types is "squashed" because it is indexed by erased contexts only, but packs both base and fiber data. The main interesting component for our purposes is the type of natural numbers, where we relate the two interpretations in the fiber:

$$\mathsf{Nat}^{\approx} \triangleq (\lambda \gamma_0, n_{\mathsf{s}}.\ \mathbb{1},\ \ \lambda\{n_{\mathsf{s}}\}, n_{\mathsf{e}}, \star.\ n_{\mathsf{e}} = \mathtt{succ}^{n_{\mathsf{s}}}\ \mathtt{zero})$$

where $a^n\ b \triangleq \mathsf{rec}_{\mathbb{N}}\ b\ a\ n$. We omit the rest of the interpretation; see our Agda formalisation (🦉).[2] From this model we obtain various useful correctness properties of extraction. Below, we write $|-|$ for the code extraction of closed terms, $[\![-]\!]$ for the **Set** interpretation of closed terms, and operate purely in the syntax $\mathbf{0}_{\mathsf{TT}_0}$.

▶ **Theorem 25** (Canonicity 🦉). *Every closed term* $n : \mathsf{Tm}_\omega \bullet \mathsf{Nat}$ *is extracted to the numeral of its set interpretation:* $|n| = \mathtt{succ}^{[\![n]\!]}\ \mathtt{zero}$.

▶ **Theorem 26** (Tracking 🦉). *The extraction of any syntactic function tracks its semantic interpretation – for all* $f : \mathsf{Tm}_\omega \bullet (\Pi\ \mathsf{Nat}\ \mathsf{Nat})$ *and all* $k : \mathbb{N}$, $\mathtt{app}\ |f|\ (\mathtt{succ}^k\ \mathtt{zero}) = \mathtt{succ}^{[\![f]\!]\ k}\ \mathtt{zero}$.

[2] The formalisation works internally to $\mathbf{Psh}(\mathbf{0}_\lambda)$, which allows us to directly use a second-order model of λ rather than closed first-order terms. We include some notes there about its relation to this version.

▶ **Theorem 27** (Non-interference 🤔). *Every erased function is constant at runtime – for all* $f : \mathsf{Tm}_\omega \bullet (\Pi_0\ \mathsf{Nat}\ \mathsf{Nat})$*, there exists a* $k : \mathbb{N}$ *such that for all* $x : \mathsf{Tm}_0 \bullet \mathsf{Nat}$*,* $|\mathsf{app}_0\ f\ x| = |f| = \mathtt{succ}^k\ \mathtt{zero}$*. Moreover,* $[\![f]\!]$ *is the constant function returning* k*.*

8 Implementation

We have implemented a toy elaborator for type theory with erasure. This is based on András Kovács' `elaboration-zoo` which contains toy implementations of dependent type theory. Our implementation is a modification of the implementation of implicit arguments and metavariables [29]. The surface language is essentially the language presented in Section 2, with mode-aware Π types and a single universe $\mathsf{U} : \mathsf{U}$. The coercions $\uparrow/\downarrow$ and the marker # are inserted automatically; the user never interacts with them.

In the repository, we include two variants of the elaboration algorithm: one which *inserts* coercions during elaboration, and one which keeps coercions implicit. The latter is closer to existing implementations of erasure. We keep the former as a proof of concept that it is possible to have a structural phase distinction which can be elaborated from a "substructural" source language. In both cases, we make a simplification to the representation of contexts: the theory TT_0 as presented features context extensions by # which can end up anywhere in the context, and can appear multiple times. However, # is a proposition, so it doesn't matter which particular witness we use. It is therefore sufficient to keep a boolean flag indicating the mere *presence* of # in a context otherwise containing only $0/\omega$ bindings. This simplifies the handling of variables, since we don't need to offset de Bruijn indices/levels by #.

The predominant source of complexity in elaborating such languages is the pattern unification algorithm that solves metavariables. Although there are theoretical foundations of pattern unification for type theory [2], such a formal analysis has not been performed for languages with erasure or other modalities. The implementations of pattern unification in the wild are thus "engineering efforts", which can go wrong. Despite checking quantities only *after* the whole program has been elaborated, Idris 2 sometimes solves metas in a weaker quantity than required, which can lead to undefined behaviour at runtime [16]. On the other hand, Agda sometimes fails to detect unsolvable metas in the presence of erasure [14].

Luckily, because our implementation is based on a structural core, we can directly reuse the theory of pattern unification to obtain a correct implementation. Our unification algorithm does not require a separate mode check after typechecking as opposed to Idris. It also does not need a dedicated generalisation mechanism for promoting erased metavariables to runtime metavariables as opposed to Agda. In the repository, we include some test cases that otherwise fail in these languages due to the incompleteness of their respective mechanisms (they are the test cases in the issues linked above). We have justified our formulation of pattern unification by some semi-formal notes in the same repository. The core idea is that the process of renaming and performing occurrence checking on candidate solutions handles not only regular variables, but also witnesses of the erasure marker # (which are morally also just variables). As a result, we only need one kind of metavariable (runtime) and generalisation is no longer necessary.

9 Related work

Our approach to erasure is based on *synthetic phase distinctions*, pioneered by Sterling and Harper [42], whose roots go back to the work on phase distinctions of Cardelli [11] and Harper, Mitchell, and Moggi [21]. Cardelli's work is the closest to erasure in terms of purpose,

but is formulated as an "indexed" type theory. In his thesis [40], Sterling develops a theory of synthetic phase distinctions as a tool for constructing logical relations (synthetic Tait computability), but with various applications to programming languages. Most recently, Grodin et al. [19] have showcased the possible use cases of a language with phase distinctions. In such settings one has access to open as well as closed modalities, corresponding to open and closed subtoposes of the topos in which the language has semantics. Using this terminology, our formulation of erasure is the open modality for the proposition #. The closest work along these lines to ours is in the form of a blog post by Sterling [38], which contains some ideas about how synthetic Tait computability relates to QTT; in particular, he observes the need for squashed universes.

Erasure in dependent types was explored by Mishra-Linger et al. [35] in the context of *pure type systems*. With the work of Gundry and McBride [20] as precursor, the modern approach to erasure has been QTT by McBride [34] and Atkey [7]. We intend to characterise the relationship between our theory and QTT, whose models are *quantitative* CwFs (QCwFs), in the future. For now, we make the observation that a *structural* QCwF (meaning with the semiring $\mathcal{R} = \{0 < \omega\}$) is an indexed CwF $U : \mathcal{L} \to \mathcal{C}$. Gluing along the reindexing functor $U^\star : \mathbf{Psh}(\mathcal{C}) \to \mathbf{Psh}(\mathcal{L})$ yields a presheaf category with a tiny proposition that models TT_0. In the other direction, given a TT_0-model $\mathcal{M}$, strictifying the pseudo-morphism $\mathcal{M} \to \mathcal{M}/\#$ into the slice CwF $\mathcal{M}/\#$ yields a structural QCwF.

More recently, Danielsson has explored some constructions in type theory with erasure [15], and Abel et al. have explored its integration with cubical type theory for Cubical Agda [1]. We expect that our system can extend the SOGAT of cubical type theory [47, 4.6.3] with a mode split for terms, and thus replicate Abel's system structurally. Favier [17] has shown that erasure behaves like an open modality in Agda, showing a synthetic Artin fracture theorem. Besides this, there has also been work on theories with "mode splits", notably type theory with colours [26]. This style of system, where terms at each mode need not be the same, can be replicated in our system; we are free to add equations that apply only under the # marker, collapsing data in the erased phase that otherwise exists at the runtime phase.

10 Conclusion

We have developed a fully structural theory of erasure using the formalism of SOGATs, and explored various syntactic and semantic models. In the future, we would like to explore more extensions to the theory. The most immediate is to support runtime types. This is relatively straightforward to add and simplifies the semantics by avoiding the need for squashed universes; we are mostly interested in exploring the utility of this feature for programming. Another extension is to add more phase distinctions. Suppose we allow equations like β-reduction only under #. We could then add a second *disjoint* phase distinction \$ to internalise the *code extraction* morphism. Doing so would allow us to control and reason about compilation output internally to the language, for example to specify runtime optimisations.

References

1 Andreas Abel, Nils Anders Danielsson, and Andrea Vezzosi. Compiling programs with erased univalence, 2022. URL: https://www.cse.chalmers.se/~nad/publications/abel-danielsson-vezzosi-erased-univalence.pdf.

2 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. *Lect. Notes Comput. Sci.*, 6690 LNCS:10–26, 2011. doi:10.1007/978-3-642-21691-6_5.

3 Run-time irrelevance — Agda 2.9.0 documentation. URL: https://agda.readthedocs.io/en/latest/language/runtime-irrelevance.html.

4 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *Foundations of Software Science and Computation Structures*, pages 293–310. Springer International Publishing, 2018. doi:10.1007/978-3-319-89366-2_16.

5 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837638.

6 Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Math. Struct. Comput. Sci.*, 33:688–743, 2023. doi:10.1017/S0960129523000130.

7 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209189.

8 Rafaël Bocquet. *Relative induction principles for second-order generalized algebraic theories*. PhD thesis, Eötvös Loránd University, 2025. URL: https://rafaelbocquet.gitlab.io/pdfs/thesis.pdf.

9 Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough, 2023. doi:10.4230/LIPIcs.FSCD.2023.18.

10 Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, pages 115–129. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24849-1_8.

11 L Cardelli. Phase distinctions in type theory. Unpublished manuscript, 1988. URL: http://lucacardelli.name/Papers/PhaseDistinctions.A4.pdf.

12 John Cartmell. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Logic*, 32:209–243, 1986. doi:10.1016/0168-0072(86)90053-9.

13 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *arXiv [cs.LO]*, 2019. arXiv:1904.00827.

14 Jesper Cockx. agda/agda: Should erasure violations be hard errors more often? # 5703, 2021. URL: https://github.com/agda/agda/issues/5703.

15 Nils Anders Danielsson. Logical properties of a modality for erasure. Technical report, University of Gothenburg, 2020. URL: https://www.cse.chalmers.se/~nad/publications/danielsson-erased.pdf.

16 Steve Dunham. idris-lang/Idris2: Holes at unerased quantity produce bad code # 3367, 2024. URL: https://github.com/idris-lang/Idris2/issues/3367.

17 Naïm Camille Favier. ErasureOpen. URL: https://agda.monade.li/ErasureOpen.

18 Daniel Gratzer, Michael Shulman, and Jonathan Sterling. Strict universes for grothendieck topoi. *arXiv [math.CT]*, 2022. URL: https://share.google/2bfRTY63iMIET0oJg.

19 Harrison Grodin, Runming Li, and Robert Harper. Abstraction functions as types. *arXiv [cs.PL]*, 2025. doi:10.48550/arXiv.2502.20496.

20 Adam Gundry and Conor Mcbride. Phase your erasure, 2013. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/phtt.pdf.

21 Robert Harper, John C Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '90*, pages 341–354, New York, New York, USA, 1990. ACM Press. doi:10.1145/96709.96744.

22 Martin Hofmann. *Syntax and semantics of dependent types*, pages 79–130. Cambridge University Press, Cambridge, 1997. doi:10.1017/CBO9780511526619.004.

23 Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997.

24 Jasper Hugunin. Why not W?, 2021. doi:10.4230/LIPIcs.TYPES.2020.8.

25 Multiplicities — Idris2 0.0 documentation. URL: https://idris2.readthedocs.io/en/latest/tutorial/multiplicities.html#multiplicities.

26 Bernardy Jean-Philippe and Guilhem Moulin. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 2013. ACM. doi:10.1145/2500365.2500577.

27 Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory, 2019. doi:10.4230/LIPIcs.FSCD.2019.25.

28 Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics, 2024. doi:10.4230/LIPIcs.FSCD.2024.10.

29 András Kovács. 04-implicit-args at master · AndrasKovacs/elaboration-zoo. URL: https://github.com/AndrasKovacs/elaboration-zoo/tree/master/04-implicit-args.

30 András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6:540–569, 2022. doi:10.1145/3547641.

31 András Kovács. *Type-theoretic signatures for algebraic theories and inductive types*. PhD thesis, Eötvös Loránd University, 2023. URL: https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf.

32 F William Lawvere. *Functorial semantics of algebraic theories*. PhD thesis, Columbia University, 1963. URL: http://www.tac.mta.ca/tac/reprints/articles/5/tr5.pdf.

33 Daniel R Licata, Ian Orton, Andrew M Pitts, and Bas Spitters. Internal universes in models of homotopy type theory, 2018. doi:10.4230/LIPIcs.FSCD.2018.22.

34 Conor McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Lecture notes in computer science. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-30936-1_12.

35 Nathan Mishra-Linger and Tim Sheard. *Erasure and Polymorphism in Pure Type Systems*, pages 350–364. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-78499-9_25.

36 Andreas Nuyts and Dominique Devriese. Transpension: The right adjoint to the pi-type. *arXiv [cs.LO]*, 2020. doi:10.48550/arXiv.2008.08533.

37 Mitchell Riley. A type theory with a tiny object. *arXiv [math.CT]*, 2024. doi:10.48550/arXiv.2403.01939.

38 Jon Sterling. On the relationship between QTT and STC, 2023. URL: https://www.jonmsterling.com/0094/.

39 Jonathan Sterling. Algebraic type theory and universe hierarchies. *arXiv [cs.LO]*, 2019. arXiv:1902.08848.

40 Jonathan Sterling. First steps in synthetic tait computability: The objective metatheory of cubical type theory, 2022. doi:10.1184/R1/19632681.v1.

41 Jonathan Sterling and Carlo Angiuli. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15. IEEE, 2021. doi:10.1109/LICS52264.2021.9470719.

42 Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *J. ACM*, 68:1–47, 2021. doi:10.1145/3474834.

43 Thomas Streicher. *UNIVERSES IN TOPOSES*, pages 78–90. Oxford University PressOxford, 2005. doi:10.1093/acprof:oso/9780198566519.003.0005.

44 Matúš Tejiščák. Erasure in dependently typed programming, 2020. doi:10.17630/STA/677.

45 Constantine Theocharis. kontheocharis/erasure-agda. Software, swhId: swh:1:dir:985bed7fde164c22db65b5fc19c69f7ad8896270 (visited on 2026-07-01). URL: https://github.com/kontheocharis/erasure-agda, doi:10.4230/artifacts.26955.

46 Constantine Theocharis. kontheocharis/erasure-impl. Software, swhId: swh:1:dir:18b74b13bf47b1740130031fe2cc393db1e95615 (visited on 2026-07-01). URL: https://github.com/kontheocharis/erasure-impl, doi:10.4230/artifacts.26954.

47 Taichi Uemura. *Abstract and concrete type theories*. PhD thesis, University of Amsterdam, 2021. URL: https://pure.uva.nl/ws/files/62028111/Thesis.pdf.

48 Théo Winterhalter. Dependent ghosts have a reflection for free. *Proc. ACM Program. Lang.*, 8:630–658, 2024. doi:10.1145/3674647.

49 Théo Winterhalter, Johann Rosain, and Matthieu Sozeau. A ghost sort for proof-relevant yet erased data in rocq and MetaRocq . *TYPES 2025 Abstract*, 2025. URL: https://msp.cis.strath.ac.uk/types2025/abstracts/TYPES2025_paper81.pdf.