Type Theory with Erasure

Constantine Theocharis
University of St Andrews

Edwin Brady
University of St Andrews

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 generates 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, and more generally in Grothendieck toposes equipped with a tiny proposition. We establish conservativity over Martin-Löf type theory 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, synthetic phase distinction, higher-order abstract syntax, logical frameworks

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material *Implementation*: `https://github.com/kontheocharis/erasure-impl`
Formalisation: `https://github.com/kontheocharis/erasure-agda`

Acknowledgements We thank András Kovács, Szumi Xie, and Bhakti Shah 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, 41]. 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) [30, 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 with erasure'. This approach is implemented by Agda [2] and Idris [1]. Besides this, there is a mechanism of 'ghost sorts' [43] planned for Rocq [44] that achieves similar goals.

We contribute yet another approach to erasure, in the style of algebraic, reduction-free type theory. The main difference over previous work is that our formulation is fully structural, corresponding to a well-studied class of type theories first identified by Uemura [42]. 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), compatible with cumulative universes, inductive families, and various other extensions.

From this formulation, we derive both syntactic and semantic results. On the syntactic side, we establish conservativity over ordinary (Martin-Löf) type theory (Theorem 10 and Corollary 11) and the independence of type information from runtime data (Theorem 7). On the semantic side, we extend the standard interpretation of type theory to account for erasure, which is possible when there is a tiny proposition available in the interpretation category; we give a fully worked example in $\mathsf{Fam}(\mathbf{Set})$ and discuss how it extends to Grothendieck toposes. We also construct a code extraction model producing untyped lambda calculus terms that only retain runtime data, along with a correctness proof by gluing that relates it to its **Set** interpretation, reminiscent of a realisability model (Theorem 20). 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 5), including notes on the implementation of pattern unification. Our theoretical results are formalised in Agda.

2 An informal presentation of erasure

We present erasure 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 two modes: the *runtime* mode (ω) and the *erased* mode (0). We order these as $0 < \omega$. Runtime terms are written as $a \overset{\omega}{:} A$, and erased terms as $a \overset{0}{:} A$. We have a *subusaging* rule which states that every runtime term can be used as an erased term. This asymmetry captures that at compile-time everything is accessible but at runtime only non-erased data is. Context variables are also annotated with mode $\{0, \omega\}$. We write $a : A$ for $a \overset{\omega}{:} A$, explicitly indicating only erased terms. In the core language the subusaging rule is replaced by an explicit mechanism (Section 2.1).

Functions $(x \overset{i}{:} A) \to B$ is the type of functions from A to B where the domain is in mode $i \in \{0, \omega\}$. A function with runtime domain $(x : A) \to B$ survives compilation, while a function with erased domain $(x \overset{0}{:} A) \to B$ is compiled to its return value directly. The usage rules above ensure that this is always valid. This results in every function $(n \overset{0}{:} \mathsf{Nat}) \to \mathsf{Nat}$ being constant (Theorem 20).

▶ **Example 1.** The signature describing length-indexed lists, or vectors, is

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

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 parameter of cons.

Pairs Similar to functions, $(x \overset{i}{:} A) \times B$ is the type of pairs; if $i = 0$ then the first projection is erased, and if $i = \omega$ it exists at runtime. For example, $\mathsf{List}\ A \triangleq (n \overset{0}{:} \mathsf{Nat}) \times \mathsf{Vec}\ A\ n$ has the same runtime representation as vectors since only the second projection survives the compilation step. We can also extend this to customise the mode of the second projection, but this can be emulated if we also have a unit type: $(x \overset{i}{:} A) \times (\_ \overset{j}{:} B) \triangleq (x \overset{j}{:} A) \times ((\_ \overset{j}{:} B) \times \mathbb{1})$.

Universes The universe of types 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. The identity $\mathsf{id} : (A \overset{0}{:} \mathsf{U}) \to A \to A$; $\mathsf{id}\ A\ x = x$ extracts to $\lambda x.\ x$. For type dependency, such as in $(x \overset{i}{:} A) \times B$, regardless of the mode i of the bound variable, it suffices for B to depend on an erased $x \overset{0}{:} A$. This is because a type family over a runtime base uniquely corresponds to a type family over an erased base (a consequence of Theorem 7).

Inductive types Inductive types are carried over from standard type theory. The main difference is that the mode of the scrutinee in eliminators must be at least as strong as the mode of the output. For example, given $x \overset{i}{:} \mathsf{Bool}$ and $a, b \overset{j}{:} A$ where $i \geq j$, we have $\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b \overset{j}{:} A$.

2.1 Erasure as a phase distinction

In the core language, erasure is defined not in terms of subusaging, but as a *phase distinction* in the sense of Sterling and Harper [39]. A desirable property of a theory with erasure is that any erased term $a \overset{0}{:} A$ well-formed in context Γ should also be well-formed in the context 0Γ, where all variable bindings are set to mode 0. This property is typically derived by induction on the syntax. We instead have a *structural* formulation of this property via a new propositional sort called the *erasure marker* #, which may appear in contexts. It induces the runtime/erased phase distinction by the following rules:

- 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 discharged in any other way. 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. This also implies that under #, runtime and compile-time terms coincide up to isomorphism. Effectively, erasure is an open modality [33]. The proposition $\# \in \Gamma$ is decidable for any context Γ, so we can build an elaboration algorithm which inserts all $(\uparrow, \downarrow)$ coercions (Section 5). 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 categories with families (CwFs). We work in constructive intensional type theory with a bounded cumulative hierarchy of universes Set_ℓ ($\ell \leq \omega + 1$), natural numbers $\mathbb{N}$, function extensionality, uniqueness of identity proofs, and quotient inductive-inductive types [5]. We 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 [20] or in some theory of signatures (e.g. Section 3.1). 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; 'TT with Π-types' denotes basic type theory judgements with functions.

3.1 Algebraic theories

First we review relevant aspects of GATs and SOGATs; see Bocquet [8] for a thorough treatment. The treatment of Uemura [42] is an alternative, but gives a weak category of models as opposed to Bocquet's strict category of models. We want to eventually generate a GAT from a SOGAT, so we favour the strict approach because it comes with a well-defined procedure to do so.

Generalised algebraic theories A Σ-*CwF* is a category with families (CwF) equipped with $\mathbb{1}$ and Σ types, where morphisms are strict CwF morphisms preserving this structure, inducing a category $\mathbf{CwF}_\Sigma \subseteq \mathbf{CwF}$. A *generalised algebraic theory* (GAT) [8] is a type-presented Σ-CwF, where type presentations consist of types (sorts), terms (operations) and equalities generating a Σ-CwF by freely extending the initial Σ-CwF. Such type presentations are usually specified using *signatures*. The theory of signatures for Σ-CwFs supports $\mathbb{1}$, Σ, equality types with UIP, a universe $\mathcal{U}$ for declaring sorts, and dependent function types Π with domain in $\mathcal{U}$. A model of a GAT T is given by a Σ-CwF morphism $\mathcal{M} : T \to \mathbf{Set}$, with the standard Σ-CwF structure on $\mathbf{Set}$.

▶ **Example 2.** The GAT Cat_1 of a category with a terminal object has sorts $\mathsf{Ob} : \mathcal{U}$ and $\mathsf{Hom} : \mathsf{Ob} \to \mathsf{Ob} \to \mathcal{U}$, a terminal object $1 : \mathsf{Ob}$ with unique morphism $!_A : \mathsf{Hom}\ A\ 1$, identity morphisms $\mathsf{id}_A : \mathsf{Hom}\ A\ A$, composition $- \circ - : \mathsf{Hom}\ B\ C \to \mathsf{Hom}\ A\ B \to \mathsf{Hom}\ A\ C$, and the usual laws (associativity, identity, uniqueness of terminal morphisms).

Second-order generalised algebraic theories A (Σ, Π_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 Π type Π_R with domain in Ty_R, inducing a category $\mathbf{CwF}_{\Sigma,\Pi_\mathsf{R}} \subseteq \mathbf{CwF}_\Sigma$. A *second-order generalised algebraic theory* (SOGAT) [25, def. 13] is a type-presented (Σ, Π_R)-CwF, extending GAT type presentations by designating some sorts as *representable* in $\mathcal{U}_\mathsf{R} \subseteq \mathcal{U}$. There is a function type Π_R with domain in $\mathcal{U}_\mathsf{R}$ yielding a sort in $\mathcal{U}$, allowing operations to bind terms of representable sorts.

▶ **Example 3.** The simplest interesting SOGAT is untyped lambda calculus with $\beta\eta$ rules:

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

comprising a single representable sort Tm of terms with an isomorphism between the function space $\mathsf{Tm} \to \mathsf{Tm}$ and Tm via lambda abstractions and applications. In the future we leave $\beta\eta$-rules implicit in isomorphisms.

A model of a SOGAT T consists of a category $\mathcal{M}_\diamond$ with terminal object and a (Σ, Π_R)-CwF morphism $\mathcal{M} : T \to \mathbf{Psh}(\mathcal{M}_\diamond)$. The codomain $\mathbf{Psh}(\mathcal{M}_\diamond)$ has a standard (Σ, Π_R)-CwF structure where the representable types include the types of $M_\diamond$ if $M_\diamond$ itself is a CwF, by the Yoneda embedding. We will generally leave the Yoneda embedding implicit; a type A in $M_\diamond$ is a (representable) type A in $\mathbf{Psh}(\mathcal{M}_\diamond)$.

▶ Remark 4. (SO)GATs specified by signatures have a universal property: a morphism $\mathcal{F} : T \to E$ where T is a (SO)GAT is uniquely determined by the interpretation of the signature of T in E. For example, a morphism $\mathcal{F} : \mathsf{Cat} \to E$ is determined in E by a type $\mathcal{F}.\mathsf{Ob}$ and a dependent type $\mathcal{F}.\mathsf{Sub}\ \Gamma\ \Delta$ over $\Gamma : \mathcal{F}.\mathsf{Ob}$ and $\Delta : \mathcal{F}.\mathsf{Ob}$, such that identity and composition are preserved.

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 T we can compute a GAT T^{fo} extending Cat_1, such that SOGAT models $\mathcal{M} : T \to \mathbf{Psh}(\mathcal{M}_\diamond)$ are in bijective correspondence with GAT models $\widetilde{\mathcal{M}} : T^{\mathsf{fo}} \to \mathbf{Set}$ where the Cat_1 part of T^{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 $T \mapsto T^{\mathsf{fo}}$ essentially corresponds to the signature-based translation from SOGATs to GATs detailed by Kaposi and Xie [25]: it maps a SOGAT T to the GAT of a category with terminal object and the presheaf interpretation of T over it. Bocquet shows that 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 5.** Translating the SOGAT $(\mathsf{Ty} : \mathcal{U}, \mathsf{Tm} : \mathsf{Ty} \to \mathcal{U}_{\mathsf{R}})$ produces the GAT of categories with families, in other words Martin-Löf type theory without any type formers.

Categories of models For any two models $\mathcal{M}, \mathcal{N}$ of a GAT T, there is a function space model $\mathbf{Func}(\mathcal{M}, \mathcal{N})$ of natural transformations from $\mathcal{M}$ to $\mathcal{N}$, displayed over T. 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 $T \to \mathbf{Func}(\mathcal{M}, \mathcal{N})$. For GATs, this yields the usual notion of GAT morphism that can be computed from signatures [28]. This induces a category of GAT models $\mathbf{Mod}(T)$. The syntax $\mathbf{0}_G$ of a GAT G is the initial G model, given by a quotient inductive-inductive type. The syntax $\mathbf{0}_S$ of a SOGAT S is simply $\mathbf{0}_{S^{fo}}$. A strict morphism of GATs $\mathcal{F} : G \to T$ yields a functor between categories of models $\mathcal{F}^* : \mathbf{Mod}(T) \to \mathbf{Mod}(G)$ by precomposition. By applying this to the initial model $\mathbf{0}_T$ we get a model $\mathcal{F}^*\ \mathbf{0}_T$, and thus a morphism of G models $!_{\mathcal{F}^*\mathbf{0}_T} : \mathbf{0}_G \to \mathcal{F}^*\ \mathbf{0}_T$. We often reuse the symbol $\mathcal{F}$ for the model $\mathcal{F}^*\ \mathbf{0}_T$ and the components of the interpretation $!_{\mathcal{F}^*\mathbf{0}_T}$.

3.2 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 3.2.

$$
\begin{aligned}
&\mathsf{Mode} \triangleq \{0, \omega\}\\
&\mathsf{Ty} : \mathbb{N} \to \mathcal{U}\\
&\mathsf{Tm} : \mathsf{Mode} \to \mathsf{Ty}_\ell \to \mathcal{U}_{\mathsf{R}}\\[1em]
&\# : \mathcal{U}_{\mathsf{R}}\\
&\#\text{-}\mathsf{prop} : (p, q : \#) \to p = q\\
&(\downarrow, \uparrow) : (\# \to \mathsf{Tm}_\omega\ A) \simeq \mathsf{Tm}_0\ A
\end{aligned}
$$

$$
\begin{aligned}
&\Pi_i : (A : \mathsf{Ty}_\ell) \to (\mathsf{Tm}_0\ A \to \mathsf{Ty}_\ell) \to \mathsf{Ty}_\ell\\
&(\mathsf{lam}, \mathsf{app}) : ((x : \mathsf{Tm}_i\ A) \to \mathsf{Tm}_\omega\ (B \downarrow x)) \simeq \mathsf{Tm}_\omega\ (\Pi_i\ A\ B)\\[1em]
&\Sigma_i : (A : \mathsf{Ty}_\ell) \to (\mathsf{Tm}_0\ A \to \mathsf{Ty}_\ell) \to \mathsf{Ty}_\ell\\
&(\mathsf{pair}, \mathsf{proj}) : ((x : \mathsf{Tm}_i\ A) \times \mathsf{Tm}_\omega\ (B \downarrow x)) \simeq \mathsf{Tm}_\omega\ (\Sigma_i\ A\ B)\\[1em]
&\mathsf{U} : (\ell : \mathbb{N}) \to \mathsf{Ty}_{\ell+1}\\
&(\mathsf{El}, \mathsf{code}) : \mathsf{Tm}_0\ \mathsf{U}_\ell \simeq \mathsf{Ty}_\ell
\end{aligned}
$$

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

TT_0 is a theory involving three sorts: types Ty externally indexed by universe levels, terms Tm externally indexed by modes $\{0, \omega\}$ and internally indexed by types, and the *erasure marker* $\#$, forced to be a proposition by $\#\text{-}\mathsf{prop}$. 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 terms in mode 0. Their mode-0 versions are derivable (Section 3.2). Universes have codes *in mode 0*. This induces an isomorphism $(\# \to \mathsf{Ty}_\ell) \simeq \mathsf{Ty}_\ell$ which also 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. Also, it is possible and practically desirable to include cumulativity [36], 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:

$$\begin{aligned}
&\mathsf{lam}_0 : ((x : \mathsf{Tm}_0\ A) \to \mathsf{Tm}_0\ (B\ x)) \to \mathsf{Tm}_0\ (\Pi_i\ A\ B)\\
&\mathsf{lam}_0\ f = \downarrow_{p.}\ \mathsf{lam}\ (\lambda x.\ \uparrow_p (f\ \downarrow x))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{app}_0 : \mathsf{Tm}_0\ (\Pi_i\ A\ B) \to (x : \mathsf{Tm}_0\ A) \to \mathsf{Tm}_0\ (B\ x)\\
&\mathsf{app}_0\ t\ x = \downarrow_{p.}\ \mathsf{app}\ (\uparrow_p t)\ (\uparrow_p x)
\end{aligned}$$

The definitional 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 fulfill 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 superscript $_0$ 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 3.2. 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:

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

We have kept 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 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 14) with such 'forced' eliminators, and other optimisations in [10] like detagging.

We could include general W-types [22] in TT_0, but we would not get 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

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

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:

$$\begin{aligned}&\mathsf{Bool}_{\omega\hookrightarrow\mathsf{true}} : \mathsf{Ty}_\ell\\ &\mathsf{true} : \mathsf{Tm}_\omega\ \mathsf{Bool}_{\omega\hookrightarrow\mathsf{true}}\\ &\mathsf{false} : \mathsf{Tm}_0\ \mathsf{Bool}_{\omega\hookrightarrow\mathsf{true}}\end{aligned}$$

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 [6, 27] with erasure. The general principle is that each judgment in the object theory becomes a type former in the meta-theory. In this case, ignoring universe levels and other type formers, to the base TT_0 setup we add

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

where 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 $\uparrow^{\mathsf{M}}_p x = \langle \uparrow_p {\sim}x \rangle$.

3.3 Generated first-order theory (↺)

Using the results from Section 3.1, we now compute the GAT specification of TT_0. This is the actual 'type system' in the traditional sense, which includes contexts and variables.

Base category structure As with any SOGAT, we start with a category with terminal object described by sorts $\mathsf{Con} : \mathcal{U}$, $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathcal{U}$, constructors $\mathsf{id} : \mathsf{Sub}\ \Gamma\ \Gamma$, $\circ : \mathsf{Sub}\ \Gamma\ \Delta \to \mathsf{Sub}\ \Phi\ \Gamma \to \mathsf{Sub}\ \Phi\ \Delta$, and equations for associativity of $\circ$ and id being an identity for $\circ$. We also have $\bullet : \mathsf{Con}$ that is terminal by the map $\epsilon : \mathsf{Sub}\ \Gamma\ \bullet$ and the uniqueness property $\epsilon\eta : (\sigma : \mathsf{Sub}\ \Gamma\ \bullet) \to \sigma = \epsilon$.

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

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

We suggestively name the presheaf for the sort $\#$ as $\# \in \Gamma$ for some context Γ, because an inhabitant indicates that $\#$ appears somewhere within Γ. Each of these presheaves comes with a strict substitution operation $A[\sigma]$, $t[\sigma]$, $\pi[\sigma]$ which respects $\circ$ and id. For $\# \in -$, we also get a uniqueness property $(x, y : \# \in \Gamma) \to x = y$ from $\#$-prop.

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 Brujin index by q and weakening by p, More explicitly, we get

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

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:

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

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

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

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

$$\begin{aligned} &\Pi : (A : \mathsf{Ty}_\ell\ \Gamma) \to (B : \mathsf{Ty}_\ell\ (\Gamma \rhd_0 A)) \to \mathsf{Ty}_\ell\ \Gamma \\ &(\mathsf{lam}, \mathsf{app}) : \mathsf{Tm}_\omega\ (\Gamma \rhd_i A)\ B[p_i, \downarrow_* q_i] \simeq \mathsf{Tm}_\omega\ \Gamma\ (\Pi_i\ A\ B) \end{aligned}$$

The type of the left side of the isomorphism applies the rather ugly-looking substitution $(p_i, \downarrow_* q_i) : \mathsf{Sub}\ (\Gamma \rhd_i A)\ (\Gamma \rhd_0 A)$ to B. This is because in the second-order formulation (Figure 1), we implicitly weaken x by the erasure marker, and then apply $\downarrow$. We can write the less 'categorical' application operator as

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

where $\langle - \rangle \triangleq (\mathsf{id}, -) : \mathsf{Tm}_i\ \Gamma\ A \to \mathsf{Sub}\ \Gamma\ (\Gamma \rhd_i A)$ is the single term substitution. The derivable erased fragment (Section 3.2) is also carried over to the first-order GAT presentation.

4 Models

In this section we explore some models of TT_0, making use of the results of Section 3.1.

4.1 Syntactic properties

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 $\#$.

▶ **Definition 6** (Zeroing). *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:*

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

The rest of the signature is implemented by the erased fragment (Section 3.2). 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 $\uparrow/\downharpoonleft$ 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 7** (Types and erased terms need nothing). *In the syntax of TT_0 there exists a substitution $\upuparrows : \mathsf{Sub}\ \Gamma\ 0\Gamma$ which becomes the identity under zeroing, such that $(0A)[\upuparrows] = A$ for types and $(0a)[\upuparrows] = \downharpoonleft_\star a$ for terms. This induces natural isomorphisms:*

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

Conservativity over type theory We can also characterise the relationship between TT_0 and TT by an invertible morphism of theories.

▶ **Definition 8** (TT_0 to TT). *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:*

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

▶ **Definition 9** (TT to TT_0). *Conversely, the morphism $\ulcorner - \urcorner : \mathsf{TT} \to \mathsf{TT}_0$ is constructed by using the erased fragment to implement the structure of TT:*

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

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. These bidirectional interpretations yield conservativity results for TT_0 over TT.

▶ **Theorem 10** (Erased conservativity of TT_0 over TT). *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 11** (Runtime conservativity of TT_0 over TT). *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.

4.2 Standard models

The standard semantics of Martin-Löf type theory are in **Set**, and can be generalised to presheaf categories [20] or Grothendieck toposes [16]. Now we explore standard models for TT_0, first at the level of sets, and then briefly at the generality of Grothendieck toposes.

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 [37]. 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 12** (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 [21] universe construction can be performed in $\mathsf{Fam}(\mathbf{Set})$, yielding the universe $\mathcal{U}_\ell \triangleq (\mathbf{Set}_\ell, \lambda A.\ A \to \mathbf{Set}_\ell)$ at level ℓ. 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 4.2. 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 13** ($\mathsf{Fam}(\mathbf{Set})$ model of TT_0). *The standard model of* TT_0 *in families of sets is* $\mathcal{S} : \mathsf{TT}_0 \to \mathbf{Psh}(\mathsf{Fam}(\mathbf{Set}))$, *given by:*

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

For erased functions we interpret $\Pi_0\ A\ B \triangleq (a : (\phi \to A)) \to B\ a$ *and for runtime functions we interpret* $\Pi_\omega\ A\ B \triangleq (a : A) \to B\ (\mathsf{p}\ a)$*, 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 $\mathsf{Fam}(\mathbf{Set})$ model, explicitly We expand $\mathcal{S}$ in first-order form: each context Γ is a set $\Gamma_0 : \mathbf{Set}$ (erased phase) and a family $\Gamma_1 : \Gamma_0 \to \mathbf{Set}$ (runtime phase). A substitution $\Gamma \to \Delta$ is a function $\sigma_0 : \Gamma_0 \to \Delta_0$ (erased) and a family of functions $\sigma_1 : \forall \gamma_0.\ \Gamma_1\ \gamma_0 \to \Delta_1\ (\sigma_0\ \gamma_0)$ (runtime), displayed over the erased function. An ℓ-type in context Γ is interpreted as a family of ℓ-sets $\Gamma_0 \to (A_0 : \mathbf{Set}_\ell) \times (A_1 : A_0 \to \mathbf{Set}_\ell)$ indexed only over Γ_0. 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 $(\gamma_0 : \Gamma_0) \to A_0\ \gamma_0$, while a runtime term is interpreted as a full section $(\sigma_0 : (\gamma_0 : \Gamma_0) \to A_0\ \gamma_0) \times (\sigma_1 : \forall \gamma_0.\ \Gamma_1\ \gamma_0 \to A_1\ (\sigma_0\ \gamma_0))$. The sort $\# \in \Gamma$ is interpreted as a map $\Gamma \to \phi$ which is equivalent to $\forall \gamma_0.\ \Gamma_1\ \gamma_0 \to \mathbb{0}$; the assertion that the runtime part of the context is uninhabited. Finally, we can compute the context formers as:

$$\begin{aligned} \bullet\ &\triangleq \mathbb{1} & &\simeq (\mathbb{1}, \lambda\_.\ \mathbb{1}) \\ \Gamma \rhd_0 A &\triangleq (\gamma : \Gamma) \times (\phi \to A\ (\mathsf{p}\ \gamma)) & &\simeq ((\gamma_0 : \Gamma_0) \times A_0\ \gamma_0,\ \lambda(\gamma_0, a_0).\ \Gamma_1\ \gamma_0) \\ \Gamma \rhd_\omega A &\triangleq (\gamma : \Gamma) \times A\ (\mathsf{p}\ \gamma) & &\simeq ((\gamma_0 : \Gamma_0) \times A_0\ \gamma_0,\ \lambda(\gamma_0, a_0).\ (\gamma_1 : \Gamma_1\ \gamma_0) \times A_1\ \gamma_0\ a_0) \\ \Gamma \rhd \# &\triangleq \Gamma \times \phi & &\simeq (\Gamma_0, \lambda\_.\ \mathbb{0}) \end{aligned}$$

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 [16] have shown (classically) that any Grothendieck topos $\mathbf{G}$ admits a lifting of $\{\mathbf{Set}_\ell\}_{\ell \in \mathbb{N}}$ where each $\mathcal{U}_\ell$ contains all ℓ-small type families; Streicher [40] 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 [29, 32, 38, 34], 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 [35] for essentially the same purpose. This is not an overly restrictive condition either: in a presheaf topos, every representable is tiny, so every representable proposition is a tiny proposition. 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 result, along with the satisfaction of the realignment axiom [16], would allow us to use synthetic Tait computability [37] for the metatheory of TT_0.

4.3 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 tackle the first requirement. To extract code, we form a model of TT_0 using the untyped lambda calculus.

▶ **Definition 14** (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 [11]. This CwF supports Π types by lambda abstraction and application, and also Σ types, natural numbers, and other data types by Church encoding.*

▶ **Definition 15** (Code extraction model of TT_0 ☋). *The code extraction model $\mathcal{E}$ of TT_0 is defined by interpreting into the base category of presheaves over the syntax of the untyped lambda calculus, $\mathbf{Psh}(\mathbf{0}_\lambda)$. In other words, it is a $(\Sigma, \Pi_{\mathsf{R}})$-CwF morphism $\mathcal{E} : \mathsf{TT}_0 \to \mathbf{Psh}(\mathbf{Psh}(\mathbf{0}_\lambda))$ defined as*

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

The double presheaf codomain is needed because we map the representable sort $\#$ to $\mathbb{0}$ which is not representable in $\mathbf{Psh}(\mathbf{0}_\lambda)$.[1] In this model, types and erased terms are trivialised, the erasure marker becomes the false proposition, and runtime terms are interpreted as lambda terms. 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:

$$\bullet = \mathbb{1}\ (\simeq \mathsf{y}0) \qquad \Gamma \rhd_0 A = \Gamma \times \mathbb{1}\ (\simeq \Gamma) \qquad \Gamma \rhd_\omega A = \Gamma \times \mathbf{0}_\lambda.\mathsf{Tm} \qquad \Gamma \rhd \# = \Gamma \times \mathbb{0}\ (\simeq \mathbb{0})$$

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. For type formers, we interpret the *runtime* Π and Σ types as the corresponding type formers in $\mathbf{Psh}(\mathbf{0}_\lambda)$, while the *erased* ones disappear. For example, we have

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

Universes disappear as well, since they only exist in the erased fragment.

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

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:

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

We denote the interpretation of closed terms in the code extraction model by $|t| : \lambda.\mathsf{Tm}\ 0$ through the isomorphism above. 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 16.** *If $\# \notin \Gamma$ (in TT_0 syntax), then $\mathcal{E}\ \Gamma$ is representable—there is a natural number c_Γ which counts the runtime bindings in Γ, satisfying*

$$\mathcal{E}\ \Gamma \simeq \mathsf{y}c_\Gamma\,.$$

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

▶ **Corollary 17.** *For a TT_0 term $t : \mathsf{Tm}_\omega\ \Gamma\ A$ where $\# \notin \Gamma$,*

$$\mathcal{E}\ t :\ \mathcal{E}.\mathsf{Tm}_\omega\ (\mathcal{E}\ \Gamma)\ (\mathcal{E}\ A) = \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathcal{E}\ \Gamma, \mathbf{0}_\lambda.\mathsf{Tm}) \simeq \mathsf{Hom}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathsf{y}c_\Gamma, \mathbf{0}_\lambda.\mathsf{Tm}) \simeq \mathbf{0}_\lambda.\mathsf{Tm}\ c_\Gamma\,.$$

4.3.1 Correctness of code extraction

To show the correctness of programs that come from the code extraction model, we set up a logical relation between the code extraction model $\mathcal{E}$ (Definition 14) and the logical interpration in **Set**. We do so by building a model of TT_0 in a glued category $\mathsf{Fam}(\mathbf{Set}) \downarrow \mathcal{F}$ [24]. Here, $\mathcal{F} : \mathsf{TT}_0^{\mathsf{fo}} \to \mathsf{Fam}(\mathbf{Set})$ is Σ-CwF morphism which interprets the base of the predicates. It is the result of combining two other morphisms $\mathcal{E}_\bullet$ and $\mathcal{S}_{\mathbf{Set}}$. We define $\mathcal{E}_\bullet$ as the composite

$$\begin{tikzcd} \mathsf{TT}_0^{\mathsf{fo}} \arrow[r, "\widetilde{\mathcal{E}}"] & \mathbf{Psh}(\mathbf{0}_\lambda) \arrow[r, "P \mapsto P\ 0"] & \mathbf{Set} \end{tikzcd}$$

which takes the global section of a λ-presheaf produced by code extraction.[2] The global sections functor preserves finite limits so the composite $\mathcal{E}_\bullet$ can be strictified into a Σ-CwF morphism [8]. We also have the **Set** interpretation $\mathcal{S}_{\mathbf{Set}} : \mathsf{TT}_0^{\mathsf{fo}} \to \mathbf{Set}$ through the $\llcorner - \lrcorner$ morphism (Definition 8) composed with the **Set** interpretation of type theory. These can be combined into the $\mathsf{Fam}(\mathbf{Set})$-valued functor $\mathcal{F} \triangleq \langle \mathcal{S}_{\mathbf{Set}}, \mathcal{E}_\bullet \rangle$, sending a context Γ to the family $(\mathcal{S}_{\mathbf{Set}}\ \Gamma, \lambda\_.\ \mathcal{E}_\bullet\ \Gamma)$. The **Set** interpretation is at the base of each object, and the code extraction is at the fibers.

Working in **Set**, the sorts of the displayed model correspond to the induction motives of the logical relation, which are shown in Figure 2. This can be seen as the logical relation version of the model described in Section 4.2. For natural numbers, 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 $s^n\ z \triangleq \mathsf{rec}_{\mathbb{N}}\ z\ s\ n$. We omit the rest of the interpretation; see our Agda formalisation (🐓).[3] From this model we obtain various useful correctness properties of extraction. Below, we write $|-|$ for the code extraction of closed terms, and $[\![-]\!]$ for the **Set** interpretation of closed terms.

[2] For this, we view $\mathcal{E}$ as a model in internal presheaves over the Hofman-Streicher universe in $\mathbf{Psh}(\mathbf{0}_\lambda)$. In other words, $\mathcal{E} : \mathsf{TT}_0 \to \mathsf{psh}_{\mathbf{Psh}(\mathbf{0}_\lambda)}(\mathcal{U})$, which corresponds to $\widetilde{\mathcal{E}} : \mathsf{TT}_0^{\mathsf{fo}} \to \mathbf{Psh}(\mathbf{0}_\lambda)$ [8, sec. 5.2].

[3] 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.

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

Figure 2 The motives of the logical relation for code extraction correctness, expanded in **Set**.

▶ **Theorem 18** (Canonicity). *Every closed term* $n : \mathsf{Tm}\ \bullet\ \mathsf{Nat}$ *is extracted to the numeral of its set interpretation:* $|n| = \mathtt{succ}^{[\![n]\!]}\ \mathtt{zero}$.

▶ **Theorem 19** (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* $k : \mathbb{N}$, $\mathtt{app}\ |f|\ (\mathtt{succ}^k\ \mathtt{zero}) = \mathtt{succ}^{[\![f]\!]\ k}\ \mathtt{zero}$.

▶ **Theorem 20** (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.

5 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 [26]. The surface language 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 Brujin 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 [4], 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 [14]. On the other hand, Agda sometimes fails to detect unsolvable metas in the presence of erasure [12]. 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) and nor a generalisation mechanism (as opposed to Agda). In the repository, we include some test cases that otherwise fail in these languages due to the issues mentioned above. We have justified our formulation of pattern unification by some semi-formal notes in the same repository.

6 Related work

Our approach to erasure is based on *synthetic phase distinctions*, pioneered by Sterling and Harper [39], whose roots go back to the work of Harper and Moggi [19]. In his thesis [37], 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. [17] 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. The closest work along these lines to ours is in the form of a blog post by Sterling [35], which contains some ideas about how synthetic Tait computability relates to quantitative type theory; in particular, he observes the need for squashed universes.

Erasure in dependent types was explored by Mishra-Linger et al. [31] in the context of *pure type systems*. With the work of Gundry and McBride [18] as precursor, the modern approach to erasure has been QTT by McBride [30] and Atkey [7]. It would be useful to explore how our theory relates to QTT, whose models are *quantitative* CwFs (QCwFs). Our initial observation is that every QCwF with $\mathcal{R} = \{0 < \omega\}$ can be turned into a $\mathsf{TT}_0^{\mathsf{fo}}$ model, and vice versa. We are interested in further exploring this relationship in the future.

More recently, Danielsson has explored some constructions in type theory with erasure [13], and Abel et al. have explored its integration with cubical type theory for Cubical Agda [3]. We expect that our system can extend the SOGAT of cubical type theory [42, 4.6.3] with a mode split for terms, and thus replicate Abel's system structurally. Favier [15] has showed that erasure behaves like an open modality in Agda, showing a synthetic Artin fracture theorem. Besides this, there has also work on theories with 'mode splits', notably type theory with colours [23]. 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.

7 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; we are mostly interested in exploring the utility of this feature for programming. Another extension is to make the equational theory of the language dependent on #: for example, we might allow β-reduction only under #. This way we would be able to reason internally about properties of programs that would otherwise be impossible under the usual equations of type theory: for example, we could reason about which runtime β-redexes survive compilation, internally.

References

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

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

3 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.

4 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. 6690 LNCS:10–26. URL: https://link.springer.com/chapter/10.1007/978-3-642-21691-6_5.

5 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. URL: http://dx.doi.org/10.1007/978-3-319-89366-2_16.

6 Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Math. Struct. Comput. Sci.*, 33:688–743, 2023. URL: https://www.cambridge.org/core/services/aop-cambridge-core/content/view/4914DB4F8E8305DFC68F9CDCA9D0C8D0/S0960129523000130a.pdf/div-class-title-two-level-type-theory-and-applications-div.pdf.

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. URL: https://doi.org/10.1145/3209108.3209189.

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

9 Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough, 2023. URL: http://dx.doi.org/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. URL: http://dx.doi.org/10.1007/978-3-540-24849-1_8.

11 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *arXiv [cs.LO]*, 2019. URL: http://arxiv.org/abs/1904.00827.

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

13 Nils Anders Danielsson. Logical properties of a modality for erasure. URL: https://www.cse.chalmers.se/~nad/publications/danielsson-erased.pdf.

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

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

16 Daniel Gratzer, Michael Shulman, and Jonathan Sterling. Strict universes for grothendieck topoi. *arXiv [math.CT]*, 2022. URL: https://share.google/2bfRTY63iMIETOoJg.

17 Harrison Grodin, Runming Li, and Robert Harper. Abstraction functions as types. URL: http://dx.doi.org/10.48550/arXiv.2502.20496.

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

19 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. URL: http://dx.doi.org/10.1145/96709.96744.

20 Martin Hofmann. *Syntax and semantics of dependent types*, pages 79–130. Cambridge University Press, Cambridge, 1997. URL: `http://dx.doi.org/10.1017/CBO9780511526619.004`.

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

22 Jasper Hugunin. Why not W?, 2021. URL: `http://dx.doi.org/10.4230/LIPIcs.TYPES.2020.8`.

23 Bernardy Jean-Philippe and Guilhem Moulin. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ACM. URL: `http://dx.doi.org/10.1145/2500365.2500577`.

24 Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory, 2019. URL: `http://dx.doi.org/10.4230/LIPIcs.FSCD.2019.25`.

25 Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics, 2024. URL: `http://dx.doi.org/10.4230/LIPIcs.FSCD.2024.10`.

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

27 András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6:540–569, 2022. URL: `https://dl.acm.org/doi/10.1145/3547641`.

28 András Kovács. Type-theoretic signatures for algebraic theories and inductive types, 2023. URL: `https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf`.

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

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

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

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

33 Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *arXiv [math.CT]*, 2017. URL: `http://dx.doi.org/10.23638/LMCS-16(1:2)2020`.

34 Mitchell Riley. A type theory with a tiny object. *arXiv [math.CT]*, 2024. URL: `http://arxiv.org/abs/2403.01939`.

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

36 Jonathan Sterling. Algebraic type theory and universe hierarchies. *arXiv [cs.LO]*, 2019. URL: `http://arxiv.org/abs/1902.08848`.

37 Jonathan Sterling. First steps in synthetic tait computability: The objective metatheory of cubical type theory, 2022. URL: `http://dx.doi.org/10.1184/R1/19632681.v1`.

38 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. URL: `http://dx.doi.org/10.1109/LICS52264.2021.9470719`.

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

40 Thomas Streicher. *UNIVERSES IN TOPOSES*, pages 78–90. Oxford University PressOxford, 2005. URL: `http://dx.doi.org/10.1093/acprof:oso/9780198566519.003.0005`.

41 Matúš Tejiščák. Erasure in dependently typed programming, 2020. URL: `http://dx.doi.org/10.17630/STA/677`.

42 Taichi Uemura. Abstract and concrete type theories, 2021. URL: `https://pure.uva.nl/ws/files/62028111/Thesis.pdf`.

43 Théo Winterhalter. Dependent ghosts have a reflection for free. *Proc. ACM Program. Lang.*, 8:630–658, 2024. URL: `https://hal.science/hal-04163836/file/icfp24main-p74-p-dba0e9df86-78716-final.pdf`.

44 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`.