Dependent Ghosts Have a Reflection for Free

Théo Winterhalter

▶ **To cite this version:**

Théo Winterhalter. Dependent Ghosts Have a Reflection for Free. 2024. hal-04163836v2

HAL Id: hal-04163836
https://hal.science/hal-04163836v2

Preprint submitted on 29 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Dependent Ghosts Have a Reflection for Free

THÉO WINTERHALTER, Inria Saclay, France

We introduce ghost type theory (GTT) a dependent type theory extended with a new universe for ghost data that can safely be erased when running a program but which is not proof irrelevant like with a universe of (strict) propositions. Instead, ghost data carry information that can be used in proofs or to discard impossible cases in relevant computations. Casts can be used to replace ghost values by others that are propositionally equal, but crucially these casts can safely be ignored for conversion. We provide a type-preserving erasure procedure which gets rid of all ghost data and proofs, a step which may be used as a first step to program extraction. We give a syntactical model of GTT using a program translation akin to the parametricity translation and thus show consistency of the theory. Because it is a parametricity model, it can also be used to derive free theorems about programs using ghost code. We further extend GTT to support equality reflection and show that we can eliminate its use without the need for the usual extra axioms of function extensionality and uniqueness of identity proofs. In particular we validate the intuition that indices of inductive types—such as the length index of vectors—do not matter for computation and can safely be considered modulo theory. The results of the paper have been formalised in Coq.

CCS Concepts: • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: dependent types, termination, consistency, extensionality

ACM Reference Format:
Théo Winterhalter. 2024. Dependent Ghosts Have a Reflection for Free. 1, 1 (February 2024), 28 pages.

1 INTRODUCTION

Dependent type theory—the underlying theory of numerous proof assistants such as Agda [Norell 2007] or Coq [Coq development team 2023]—typically feature two notions of equality: definitional equality that is external and determines which types are the same; and propositional equality which is internal and which allows the user to perform equational reasoning.

Definitional equality is crucial in practice as it makes the life of the user easier by identifying objects on the nose. For instance, it allows one to consider the type of lists of length $3 + 2$—usually written $\mathsf{vec}\ A\ (3 + 2)$—as being the same as the type of those of length 5, *i.e.* $\mathsf{vec}\ A\ 5$. This in turn makes it possible to state and prove a propositional equality between $[1; 2; 3; 4; 5]$ and the concatenation of $[1; 2; 3]$ and $[4; 5]$.[1] That said, be it in Agda or in Coq, this notion of definitional equality is limited in order to remain decidable. A typical instance is that while $0 + n$ and n are identified, it is not the case in general for $n + 0$ to be equal to n. This can be somewhat alleviated by extending the proof assistant with rewrite rules [Cockx and Abel 2016; Cockx et al. 2021] but then we are still stuck when comparing $n + m$ and $m + n$. A remedy was once proposed in the form of Coq modulo theory (CoqMT) by Strub [2010] but this process cannot be complete while remaining decidable. As such, other proof assistants such as F* [Swamy et al. 2016], Andromeda 1 [Bauer et al. 2016] or NuPRL [Constable and Bates 2014] embrace undecidability and implement extensional type theories which make propositional and definitional equalities coincide, a phenomenon called equality reflection.

[1] These two lists are in fact definitionally equal themselves.

Author's address: Théo Winterhalter, theo.winterhalter@inria.fr, Inria Saclay, 1 Rue Honoré d'Estienne d'Orves, Palaiseau, France, 91120.

2023. ACM XXXX-XXXX/2024/2-ART
https://doi.org/

Extensional type theory (ETT) has been well studied and Hofmann [1995] showed it was conservative over intensional type theory (ITT, *i.e.* without equality reflection) extended with two equality principles: uniqueness of identity proofs (UIP), which equates any two proofs of the same propositional equality; and function extensionality (funext) which allows one to equate two functions as long as they are pointwise equal. This conservativity result was later made constructive and turned into an effective translation [Oury 2005; Winterhalter et al. 2019]. The need for the UIP axiom is at odds with homotopy type theory and in particular the univalence axiom as they are inconsistent together. The two can still be made to coexist in a single type theory using so-called two-level type theories as proposed by Altenkirch et al. [2016]. Thankfully, the translation proposed by Winterhalter et al. [2019] is general enough to apply to a two-level setting. Still it would be interesting to delineate a subsystem of ETT that does not require the use of funext and UIP so as to remain as agnostic as possible.

Furthermore, even though the lack of decidable type checking does not prevent systems such as F* to be usable in practice—through the use of SMT automation in the case of F*—the equality reflection rule has many drawbacks. First, in an inconsistent context, anything goes: since all equalities are provable, then all terms have all types; for instance, assuming $D = D \to D$ is enough to encode untyped λ-calculus and thus non-terminating functions. Second, even under consistent contexts one can construct nonsensical terms; for instance, assuming $\mathsf{nat} \to \mathsf{nat} = \mathsf{nat} \to \mathsf{bool}$ (validated by the cardinal model proposed by Bauer and Winterhalter [2020]), one can write the identity function for natural numbers $\lambda x.x$ and give it type $\mathsf{nat} \to \mathsf{bool}$ so that $(\lambda x.x)\ \mathsf{0}$ is of type bool when it should be equal to $\mathsf{0}$. The solution to this problem is to annotate both function abstraction and application by their domain and codomain and restrict β-reduction to the cases where they match individually, essentially bypassing the lack of injectivity of Π-types (which would in particular say that $\mathsf{nat} = \mathsf{bool}$ follows from $\mathsf{nat} \to \mathsf{nat} = \mathsf{nat} \to \mathsf{bool}$, a fact that usually holds in ITT) but as a result terms become exponential in size compared to their ITT counterparts. Third, for similar reasons, ETT has no power to discriminate between different type formers, there is no way for the type checker to conclude that a definitional equality between *e.g.* nat and $\mathsf{nat} \to \mathsf{nat}$ is not possible and as such it cannot terminate early and report an error. All these points show the difficulty for a checker to give useful feedback and error messages to the user. They also raise problems about extraction of programs written in ETT. In fact ETT, has no concrete evaluation semantics.

We propose a restriction of ETT that remedies several[2] of these drawbacks by allowing equality reflection for ghost values, *i.e.* those that can be erased at extraction. This extends the definitional proof irrelevance of Prop, introduced by Gilbert et al. [2019], which identifies all proofs of the same proposition. Let us illustrate this intuition by looking at the example of vectors—or length-indexed lists—introduced above. In order to write the reversal of vectors using an accumulator (to obtain a tail-recursive version) we have to consider both the length of the list to be reversed and that of the accumulator to express the length of the result. As such one would write the following:

$$\begin{aligned}
&\mathsf{rev} : \forall\ n\ m.\ \mathsf{vec}\ A\ n \to \mathsf{vec}\ A\ m \to \mathsf{vec}\ A\ (n + m)\\
&\mathsf{rev}\ \mathsf{0}\ m\ \mathsf{vnil}\ acc := acc\\
&\mathsf{rev}\ (\mathsf{S}\ k)\ m\ (\mathsf{vcons}\ a\ k\ v)\ acc := \mathsf{rev}\ k\ (\mathsf{S}\ m)\ v\ (\mathsf{vcons}\ a\ m\ acc)
\end{aligned}$$

While this definition is perfectly fine for regular lists, it suffers from two issues:

(1) The second clause of the pattern matching is not well typed in ITT as the recursive call is of type $\mathsf{vec}\ A\ (k + \mathsf{S}\ m)$ instead of the expected $\mathsf{vec}\ A\ (\mathsf{S}\ k + m)$ which is only propositionally

[2]We in fact conjecture that all of these drawbacks are remedied by our proposal but leave the question of evaluation to future work.

```
type 'a vec =
| Vnil
| Vcons of 'a * nat * 'a vec
```

```
let rec rev _ m v acc =
 match v with
 | Vnil → acc
 | Vcons (a, n, v0) →
   Obj.magic (rev n (S m) v0 (Vcons (a, m, acc)))
```

Fig. 1. Extraction of vector reversal from Coq to OCaml

equal to it. In ITT one would need to use a transport along said equality to make the term type check, at the cost of polluting the produced term. In ETT the definition would be accepted as is by reflecting that same equality, but running the translation of Winterhalter et al. [2019] on it would also result in a transport.

(2) n and m appear in the function definition and will thus remain in the extracted program, even if they are essentially there for typing purposes and do not intervene in the computation. One could argue that extraction should figure it out, but bear in mind that there is nothing preventing the user from using n and m in a meaningful way. The same goes for the natural numbers contained in the vector (k in vcons a k v). We show how extraction of vectors and rev extract from the current Coq to OCaml in Figure 1. Not only are some useless natural stored, they are also passed around. Notice also how `Obj.magic` is used to perform an unsafe cast, it comes from the equality rewrite, even though it is no longer necessary.

We thus argue that instead of taking n and m to be of type nat, they should instead be of type erased nat. The erased modality indicates that its values should be considered ghost and cannot therefore be used in a way that is relevant for computation. It is nevertheless different from the squash ‖nat‖ which identifies all its elements (meaning ‖nat‖ is equivalent to $\top$, the true proposition type with exactly one inhabitant). In particular, we are still able to distinguish different erased values such as hide 0 and hide (S 0); which is needed to write the function that takes the head of a vector whose length it not hide 0.

To define rev we can then exploit a property of ghost types we establish that lets us safely coerce from $P\ u$ to $P\ v$ as long as u and v are propositionally equal ghost values. What's new is that this operation is essentially invisible: it doesn't get in the way of computation (unlike usual rewriting with propositional equality) and it disappears at extraction, leaving us with the desired lists together with their usual reversal. Some of this can already be achieved in F* which does support ghost computations but to the best of our knowledge, there is no formal justification for them. This work can be seen as the presentation of a well-behaved subset of what F* offers.[3]

We in fact take inspiration from this intuition coming from erasure and extraction to build the model of our ghost type theory by adapting a translation of Pédrot and Tabareau [2018] that handles exceptions in dependent type theory: all ghost and proof informations are erased and inaccessible branches locally correspond to raised exceptions; then a parametricity translation ensures that these exceptions are never actually raised.

Outline of the paper. In Section 2 we introduce GTT, a dependent type theory with a universe of ghost types along with ghost casts. Then in Section 3 we give a model of GTT by means of three translations accounting for the three levels of relevance (for propositions, ghost types and regular types). We derive consistency and type former discrimination from it in Section 4. We then show how to extend both GTT and its model with inductive types such as vectors in Section 5, allowing us to show some concrete examples and to illustrate how one can leverage parametricity to get free

[3]We include as supplementary material a file showing the vector example written in F*.

theorems. Finally we give in Section 6 a definition of GRTT, a version of GTT with ghost reflection instead of casts and show how we can adapt (and simplify!) the proof of Oury [2005]; Winterhalter et al. [2019] to translate GRTT to GTT.

Formalisation. Most of our results have been formalised in Coq,[4] we will provide pointers next to definitions and theorems to their counterpart in the Coq development. These pointers will be indicated as [`Module_name.def_name`] to refer to `def_name` in the file `Module_name.v`. Our Coq development relies on the Autosubst 2 library to deal with binders [Dapprich and Dudenhefner 2021; Stark et al. 2019].

2 GHOST TYPE THEORY

Ghost type theory (GTT) is essentially a version of Martin-Löf type theory (MLTT) with a universe of definitional proof-irrelevant propositions Prop [Gilbert et al. 2019] to which we add a hierarchy of universes of ghost types written Ghost_i. The only way to construct a ghost type is through the erased type former and the inhabitants of erased A are obtained from values v of type A, written hide v. One can also eliminate erased values thanks to the eliminator reveal; of course erased values may not be used to produce relevant information, only ghost or propositional. The names erased, hide and reveal are taken from F* convention [Swamy et al. 2016][5] for those. Equality of ghost expressions $u, v : A$ is a proposition written $u \approx_A v$, which is reflexive (thanks to its only constructor gh-refl) and which is eliminated through the cast operator.

2.1 Syntax of GTT [`GAST.term`]

The syntax of GTT is mostly standard and described below. We put implicit arguments in subscript so we can omit them easily.

$$
\begin{array}{rcll}
\Gamma, \Delta & ::= & \bullet \mid \Gamma, x^{\mathfrak{s}} : A & \\
t, u, A, B & ::= & x \mid \mathsf{Kind}_i \mid \mathsf{Type}_i \mid \mathsf{Ghost}_i \mid \mathsf{Prop} & (i \in \mathbb{N}) \\
& \mid & \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B \mid \lambda(x^{\mathfrak{s}} : A).t \mid t\ u & \\
& \mid & \mathsf{erased}\ A \mid \mathsf{hide}\ t \mid \mathsf{reveal}\ t\ P\ p & \\
& \mid & \mathsf{Reveal}\ t\ p \mid \mathsf{toRev}_{t,p}\ u \mid \mathsf{fromRev}_{t,p}\ u & \\
& \mid & u \approx_A v \mid \mathsf{gh\text{-}refl}_A\ u \mid \mathsf{cast}_{A,u,v}\ e\ P\ t & \\
& \mid & \bot \mid \mathsf{exfalso}_{\mathfrak{s}}\ A\ p & \\
\mathfrak{r}, \mathfrak{s} & ::= & \mathbb{K} \mid \mathbb{T} \mid \mathbb{G} \mid \mathbb{P} &
\end{array}
$$

Several things might appear surprising so we explain them one by one.

First, notice how, beyond Ghost and Prop, we have two other universe hierarchies, Kind and Type. The main reason for this is that for our model, we need the parametricity translation to produce propositions, even on Type, which would be ill typed if we had $\mathsf{Type}_i : \mathsf{Type}_{i+1}$. We thus implement the solution of Keller and Lasson [2012][6] of separating regular types from universes and thus having $\mathsf{Type}_i : \mathsf{Kind}_{i+1}$ but also $\mathsf{Prop} : \mathsf{Kind}_0$ and $\mathsf{Ghost}_i : \mathsf{Kind}_{i+1}$.

A related observation is that we also have extra annotations on binders as well as on variables and on the eliminator for the empty type $\bot$. These *modes* have to be $\mathbb{K}, \mathbb{T}, \mathbb{G}$ or $\mathbb{P}$ and respectively correspond to Kind, Type, Ghost and Prop without the universe levels. We will omit them when they are not important and can be inferred from the context in order to improve readability. As we will see, they can indeed be inferred from typing alone (Lemma 4.4).

[4] Publicly available at https://github.com/TheoWinterhalter/ghost-reflection

[5] Our reveal is an eliminator rather than a projection however.

[6] In their paper, they call Set what we call Type, and Type what we call Kind. We believe ours is the most natural choice.

Finally, one might notice how we both have reveal as the eliminator for erased and Reveal. The idea is that reveal can only be used in mode $\mathbb{P}$ or $\mathbb{G}$ since it *reveals* relevant content that is supposed to be erased. There is however the need to be able to construct propositions from erased values in order to be able to discriminate *e.g.* hide 0 and hide 1. As such, Reveal $t\ p$ is a proposition obtained by eliminating t such that Reveal (hide t) p is logically equivalent to $p\ t$, logical equivalence that is materialised with the pair (toRev, fromRev).

As usual, we write $A \to B$ for $\forall^{\mathfrak{r}}(x^{\mathfrak{s}} : A).B$ when B does not depend on A and when the modes are inferable from the context. $\forall$ is also annotated with universe levels, we will omit them when irrelevant or obvious from the context. We will also omit universe levels from universes and write *e.g.* Type. We will write $\mathrm{Sort}_i^{\mathfrak{s}}$ with the convention that $\mathrm{Sort}_i^{\mathbb{K}} = \mathrm{Kind}_i$, $\mathrm{Sort}_i^{\mathbb{T}} = \mathrm{Type}_i$, $\mathrm{Sort}_i^{\mathbb{G}} = \mathrm{Ghost}_i$ and $\mathrm{Sort}_i^{\mathbb{P}} = \mathrm{Prop}$.

We write $t[x := u]$ for the substitution of x by u in term t.

2.2 Cast-free syntax [CastRemoval.castrm]

We want to ensure that ghost casts do not get in the way of equality. Consider for instance the expression cast $e\ (\lambda x.\,\mathrm{nat})\ 0$ which is to be understood as taking some ghost equality $e : u \approx v$ to be able to cast 0 of type $(\lambda x.\,\mathrm{nat})\ u$ to type $(\lambda x.\,\mathrm{nat})\ v$. As both sides are in fact nat, we would like to be able to state that this is equal to 0. Even more so, we would like to conclude that $(\mathrm{cast}\ e\ (\lambda x.\,\mathrm{nat})\ 0) + n$ is equal to n. The problem is not limited to cases where the cast is essentially doing nothing, consider the cast of a λ-abstraction: cast $e\ (\lambda x.A\ x \to B\ x)\ (\lambda(y : A\ u).\ t)$, where $e : u \approx v$, thus mapping the function of type $A\ u \to B\ u$ to $A\ v \to B\ v$. We would like to apply this function and have it compute, in other words we would like it be a λ-abstraction itself, typically by casting inside the body of the function to move the argument from $A\ v$ to $A\ u$ and the result from $B\ u$ to $B\ v$:

$$\lambda(z : A\ v).\ \mathrm{cast}\ e\ B\ t[y := \mathrm{cast}\ e^{-1}\ A\ z]$$

where $e^{-1} : v \approx u$ is just symmetry applied to e.

To achieve this we introduce an operator $|-|$ that removes all casts from a term (or a context). Our conversion rule is then going to compare terms for which we removed all casts, giving us the desired equalities we mentioned above. The model is going to justify this approach. The definition of $|-|$ is straightforward so we only show a few examples.

$$|\mathrm{cast}\ e\ P\ t| := |t| \qquad |t\ u| := |t|\ |u| \qquad |\forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B| := \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : |A|).|B|$$

2.3 Mode of a term [TermMode.md, Scoping.scoping]

Similarly to Gilbert et al. [2019, Section 4.5] we want to distinguish relevant and irrelevant computations syntactically, and the same goes for ghost computations. Like for them, it allows us to define conversion syntactically, while handling a priori semantic rules like proof irrelevance. For us it also fills another role as the definition of the translations of Section 3 uses this information. This information is essentially recorded in the context Γ as it consists of type and mode declarations $(x^{\mathfrak{s}} : A)$. We thus define a scoping judgement $\Gamma \vdash t :: \mathfrak{s}$ which says that t has mode $\mathfrak{s}$. The idea is that whenever $t : A : \mathrm{Sort}_i^{\mathfrak{s}}$ then $t :: \mathfrak{s}$. This fact will be established after we give the model (Lemma 4.4). We additionally define a function which determines the *mode* of a term, syntactically. Note that the two agree on well-scoped terms so we omit the definition of scoping altogether and opt for the more concise md definition. We leave the context Γ implicit and instead allow ourselves to write $x^{\mathfrak{s}}$ to mean $(x^{\mathfrak{s}} : A) \in \Gamma$ for some A.

$$\mathrm{md}(x^{\mathfrak{s}}) := \mathfrak{s} \qquad \mathrm{md}(\mathrm{Sort}_i^{\mathfrak{s}}) := \mathbb{K} \qquad \mathrm{md}(\forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B) := \mathbb{K} \qquad \mathrm{md}(\lambda(x^{\mathfrak{s}} : A).t) := \mathrm{md}(t)$$

$$\mathrm{md}(t\ u) := \mathrm{md}(t) \qquad \mathrm{md}(\mathrm{erased}\ A) := \mathbb{K} \qquad \mathrm{md}(\mathrm{hide}\ t) := \mathbb{G}$$

$$\mathrm{md}(\mathrm{reveal}\ t\ P\ p) := \mathrm{if}\ \mathrm{md}(p) = \mathbb{G}\ \mathrm{then}\ \mathbb{G}\ \mathrm{else}\ \mathbb{P} \qquad \mathrm{md}(\mathrm{Reveal}\ t\ p) := \mathbb{K}$$

$$\mathrm{md}(\mathrm{toRev}_{t,p}\ u) := \mathbb{P} \qquad \mathrm{md}(\mathrm{fromRev}_{t,p}\ u) := \mathbb{P} \qquad \mathrm{md}(u \approx_A v) := \mathbb{K} \qquad \mathrm{md}(\text{gh-refl}_A\ u) := \mathbb{P}$$

$$\mathrm{md}(\mathrm{cast}_{A,u,v}\ e\ P\ t) := \mathrm{md}(t) \qquad \mathrm{md}(\bot) := \mathbb{K} \qquad \mathrm{md}(\mathrm{exfalso}_{\mathfrak{s}}\ A\ p) := \mathfrak{s}$$

2.4 Typing rules of GTT [Typing.typing, Typing.conversion, Typing.wf]

GTT features three kinds of judgements: $\vdash \Gamma$ (context formation), $\Gamma \vdash t : A$ (typing), and $\Gamma \vdash u \equiv v$ (definitional equality, or conversion). Context formation and typing rules are given in Figure 2, while an excerpt of definitional equality rules is given in Figure 3. We omit the structural and congruence rules and instead focus on computation rules and proof irrelevance, the remaining rules are found in the formalisation. We do not include η-rules or cumulativity for now as we believe them to be mostly orthogonal to the problem at hand.

Universes. In order to factorise the different rules involving universes we define umax to compute the universe level $\mathrm{umax}\ \mathfrak{s}\ \mathfrak{r}\ i\ j$ of $\forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B$ and usup for the universe of a sort. [Univ.usup, Univ.umax]

$$\mathrm{usup}\ \mathbb{P}\ i := 0 \qquad \mathrm{usup}\ \mathfrak{s}\ i := i + 1 \quad \text{(otherwise)}$$

$$\mathrm{umax}\ \mathfrak{s}\ \mathbb{P}\ i\ j := 0 \qquad \mathrm{umax}\ \mathbb{P}\ \mathfrak{r}\ i\ j := j \qquad \mathrm{umax}\ \mathfrak{s}\ \mathfrak{r}\ i\ j := \max(i, j) \quad \text{(otherwise)}$$

Ghost-specific rules. You can see how reveal is used to produce proofs (of a proposition) or ghost values, but also how Reveal extends this to producing propositions directly. We shall see in Section 5.2 how Reveal is useful to discriminate constructors of an erased inductive type which is a specificity of ghost types. We highlight the rules regarding casts in blue. Note how we cannot use cast in sort Kind as our model does not support it. The conversion rule completely removes casts before comparing terms which subsumes any computation rule for casts.

Scoping premises. You may have noticed that some rules feature scoping conditions. This is very important in the proof irrelevance rule, where we require both sides to be *proofs* without worrying about their type, following what Gilbert et al. [2019, Section 4.5] do for the implementation of proof irrelevance in Coq. What may appear more surprising is that other rules feature scoping premises. The conversion rule essentially requires the new type to be compatible with the mode of the term being converted. We conjecture that this extra requirement is not necessary and it was proven formally in Coq for strict propositions [Leray 2022]. Computation rules also require terms to be well scoped. This is a limitation due to the way we build our model as we need to know that conversion preserves scoping (Lemma 2.4). They should not be necessary for an implementation as reduction, contrarily to conversion, does preserve scoping. Finally, note that the rules we consider in the formalisation actually feature many more such premises that we only show later are admissible (in the `Admissible` module, Lemmas 2.6 and 4.4) *after* having built the model and assuming injectivity of $\forall$ holds for our conversion. We chose to present the simpler rules for the sake of clarity.

2.5 Preliminary results

Without the model we can already establish various lemmas, in particular lemmas that connect semantic information (typing and conversion) and syntactic information (modes).

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathrm{Sort}^{\mathfrak{s}}}{\vdash \Gamma, x^{\mathfrak{s}} : A} \qquad \frac{(x^{\mathfrak{s}} : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash t :: \mathfrak{s} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash |A| \equiv |B| \qquad \Gamma \vdash B : \mathrm{Sort}^{\mathfrak{s}}}{\Gamma \vdash t : B} \qquad \frac{}{\Gamma \vdash \mathrm{Sort}^{\mathfrak{s}}_i : \mathrm{Kind}_{\mathrm{usup}\ \mathfrak{s}\ i}}$$

$$\frac{\Gamma \vdash A : \mathrm{Sort}^{\mathfrak{s}}_i \qquad \Gamma, x^{\mathfrak{s}} : A \vdash B : \mathrm{Sort}^{\mathfrak{r}}_j}{\Gamma \vdash \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B : \mathrm{Sort}^{\mathfrak{r}}_{\mathrm{umax}\ \mathfrak{s}\ \mathfrak{r}\ i\ j}}$$

$$\frac{\Gamma \vdash A : \mathrm{Sort}^{\mathfrak{s}}_i \qquad \Gamma, x^{\mathfrak{s}} : A \vdash B : \mathrm{Sort}^{\mathfrak{r}}_j \qquad \Gamma, x^{\mathfrak{s}} : A \vdash t : B}{\Gamma \vdash \lambda(x^{\mathfrak{s}} : A).t : \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B} \qquad \frac{\Gamma \vdash t : \forall^{\mathfrak{r}}(x^{\mathfrak{s}} : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}$$

$$\frac{}{\Gamma \vdash \bot : \mathrm{Prop}} \qquad \frac{\Gamma \vdash A : \mathrm{Sort}^{\mathfrak{s}} \qquad \Gamma \vdash p : \bot}{\Gamma \vdash \mathrm{exfalso}_{\mathfrak{s}}\ A\ p : A} \qquad \frac{\Gamma \vdash A : \mathrm{Type}_i}{\Gamma \vdash \mathrm{erased}\ A : \mathrm{Ghost}_i}$$

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathrm{hide}\ t : \mathrm{erased}\ A}$$

$$\frac{\Gamma \vdash t : \mathrm{erased}\ A \qquad \Gamma \vdash P : \mathrm{erased}\ A \to \mathrm{Sort}^{\mathfrak{s}} \qquad \Gamma \vdash p : \forall^{\mathfrak{s}} x^{\mathbb{G}}.\ P\ (\mathrm{hide}\ x) \qquad \mathfrak{s} \in \{\mathbb{P}, \mathbb{G}\}}{\Gamma \vdash \mathrm{reveal}(t, P, p) : P\ t}$$

$$\frac{\Gamma \vdash t : \mathrm{erased}\ A \qquad \Gamma \vdash p : A \to \mathrm{Prop}}{\Gamma \vdash \mathrm{Reveal}\ t\ p : \mathrm{Prop}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : A \to \mathrm{Prop} \qquad \Gamma \vdash u : p\ t}{\Gamma \vdash \mathrm{toRev}_{t,p}\ u : \mathrm{Reveal}\ t\ p}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : A \to \mathrm{Prop} \qquad \Gamma \vdash u : \mathrm{Reveal}\ t\ p}{\Gamma \vdash \mathrm{fromRev}_{t,p}\ u : p\ t}$$

$$\frac{\Gamma \vdash A : \mathrm{Ghost} \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u \approx_A v : \mathrm{Prop}} \qquad \frac{\Gamma \vdash A : \mathrm{Ghost} \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathrm{gh\text{-}refl}_A\ u : u \approx_A u}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathrm{Ghost} \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A \\ \Gamma \vdash e : u \approx_A v \qquad \Gamma \vdash P : A \to \mathrm{Sort}^{\mathfrak{s}} \qquad \Gamma \vdash t : P\ u \qquad \mathfrak{s} \neq \mathbb{K}\end{array}}{\Gamma \vdash \mathrm{cast}_{A,u,v}\ e\ P\ t : P\ v}$$

Fig. 2. Typing rules of GTT

Lemma 2.1. [BasicMetaTheory.scoping_subst] *Substitution preserves scoping.*
If $\Gamma, x^{\mathfrak{s}} : A \vdash t :: \mathfrak{r}$ *and* $\Gamma \vdash u :: \mathfrak{s}$ *then* $\Gamma \vdash t[x := u] : \mathfrak{r}$.

Lemma 2.2. [BasicMetaTheory.scoping_castrm] *Cast removal preserves scoping.*
If $\Gamma \vdash t : \mathfrak{s}$ *then* $\Gamma \vdash |t| : \mathfrak{s}$.

$$\frac{\Gamma \vdash p :: \mathbb{P} \qquad \Gamma \vdash q :: \mathbb{P}}{\Gamma \vdash p \equiv q} \qquad \frac{\Gamma \vdash A :: \mathbb{K} \qquad \Gamma \vdash t :: \mathfrak{r} \qquad \Gamma \vdash u :: \mathfrak{s}}{\Gamma \vdash (\lambda(x^{\mathfrak{s}} : A).t)\ u \equiv t[x := u]}$$

$$\frac{\Gamma \vdash t :: \mathbb{T} \qquad \Gamma \vdash P :: \mathbb{K} \qquad \Gamma \vdash p :: \mathfrak{s} \qquad \mathfrak{s} \in \{\mathbb{P}, \mathbb{G}\}}{\Gamma \vdash \mathrm{reveal}(\mathrm{hide}\ t, P, p) \equiv p\ t}$$

Fig. 3. Conversion for GTT (excerpt)

Lemma 2.3. [BasicMetaTheory.castrm_subst] *Cast removal commutes with substitution.* $|t[x := u]| =_\alpha |t|[x := |u|]$.

Lemma 2.4. [BasicMetaTheory.conv_md] *Conversion entails mode equality.* *If* $\Gamma \vdash u \equiv v$ *then* $\mathrm{md}_\Gamma(u) = \mathrm{md}_\Gamma(v)$.

Lemma 2.5 (Substitution). [BasicMetaTheory.typing_subst] *If* $\Gamma, x^{\mathfrak{s}} : A \vdash t : B$ *and* $\Gamma \vdash u : A$ *with* $\Gamma \vdash u :: \mathfrak{s}$ *then we have* $\Gamma \vdash t[x := u] : A[x := u]$.

We also show a variant of *validity* (sometimes called *presupposition*) that states that types are themselves well typed. We additionally conclude that the term is well scoped. Note how this lemma requires the context to be well formed as well since GTT typing derivations do not contain context formation assumptions.

Lemma 2.6 (Validity). [BasicMetaTheory.validity] *If* $\vdash \Gamma$ *and* $\Gamma \vdash t : A$ *then there exists a level* i *and mode* $\mathfrak{s}$ *such that* $\Gamma \vdash A : \mathrm{Sort}_i^{\mathfrak{s}}$ *and* $\Gamma \vdash t :: \mathfrak{s}$.

3 MODEL OF GTT

The intuition behind ghost data is that it can be safely erased from programs without affecting the outcome of computation. We make this intuition formal by defining an *erasure* procedure that removes all ghost data, as well as proofs. There is but one caveat: ghost information (or proofs) can be used to discard inaccessible execution branches by means of exfalso. In programming languages such as OCaml this is often handled using exceptions or the `assert false` idiom.

We thus build a syntactical model [Boulier et al. 2017] of GTT by adapting the proof of Pédrot and Tabareau [2018] giving a model to a type theory with exceptions. Pédrot and Tabareau perform two translations: one that maps all types to pointed types (to interpret the exceptions); together with a parametricity translation that shows the first translation produces *reasonable* terms, *i.e.* terms that do not raise exceptions at top-level. We follow them, introducing erasure (Section 3.1) and parametricity (Section 3.3) but also a third translation in-between to deal with ghost values we call revival (Section 3.2).

Target. [CCAST.cterm, CTyping.ctyping] The target of these translations is a now standard variant of MLTT with a universe of definitional proof-irrelevant propositions and inductive types. We take the presentation of Section 4.5 of the theory presented by Gilbert et al. [2019], meaning one can think of the target either as Coq or as Agda with cumulativity. Essentially, it is a variant of GTT presented in Section 2 with a few differences: (1) Kind and Type are collapsed and there is neither a Ghost universe nor the associated constructions; (2) annotations are limited to $\mathbb{P}$ and $\mathbb{T}$ and they correspond to the Relevant and Irrelevant annotations of Gilbert et al. [2019]; (3) extra inductive types are defined, we will detail them as we need them.

$$
\begin{array}{lcl}
[x^{\mathbb{K}/\mathbb{T}}]_\varepsilon & := & x \\
[\mathrm{Prop}]_\varepsilon & := & \mathrm{tyval}\ \mathrm{unit}\ () \\
[\mathrm{Sort}_i^{\mathfrak{s}}]_\varepsilon & := & \mathrm{tyval}\ \mathrm{ty}_i\ \mathrm{ty}_\emptyset \\
[\forall^{\mathbb{K}/\mathbb{T}/\mathbb{G}}(x^{\mathbb{K}/\mathbb{T}} : A).B]_\varepsilon & := & \mathrm{tyval}\ (\forall(x : [\![A]\!]_\varepsilon).[\![B]\!]_\varepsilon)\ (\lambda x.[B]_\emptyset) \\
[\forall^{\mathbb{G}}(x^{\mathbb{G}} : A).B]_\varepsilon & := & \mathrm{tyval}\ ([\![A]\!]_\varepsilon \to [\![B]\!]_\varepsilon)\ (\lambda\_.[B]_\emptyset) \\
[\forall^{\mathbb{P}}(x^{\mathfrak{s}} : A).B]_\varepsilon & := & () \\
[\forall^{\mathfrak{r}}(x^{\mathfrak{s}} : A).B]_\varepsilon & := & [B]_\varepsilon \\
[\lambda(x^{\mathbb{K}/\mathbb{T}} : A).t^{\mathbb{K}/\mathbb{T}}]_\varepsilon & := & \lambda(x : [\![A]\!]_\varepsilon).[t]_\varepsilon \\
[\lambda(x^{\mathbb{G}/\mathbb{P}} : A).t^{\mathbb{K}/\mathbb{T}}]_\varepsilon & := & [t]_\varepsilon \\
[t^{\mathbb{K}/\mathbb{T}}\ u^{\mathbb{K}/\mathbb{T}}]_\varepsilon & := & [t]_\varepsilon\ [u]_\varepsilon \\
[t^{\mathbb{K}/\mathbb{T}}\ u^{\mathbb{G}/\mathbb{P}}]_\varepsilon & := & [t]_\varepsilon \\
[\mathrm{erased}\ A]_\varepsilon & := & [A]_\varepsilon \\
[\mathrm{Reveal}\ t\ p]_\varepsilon & := & () \\
[u \approx_A v]_\varepsilon & := & () \\
[\mathrm{cast}_{A,u,v}(e, P, t)]_\varepsilon & := & [t]_\varepsilon \\
[\bot]_\varepsilon & := & () \\
[\mathrm{exfalso}_{\mathbb{K}/\mathbb{T}}\ A\ p]_\varepsilon & := & [A]_\emptyset \\
[t^{\mathbb{G}/\mathbb{P}}]_\varepsilon & := & \blacksquare
\end{array}
$$

$$
\begin{array}{lcl}
[\![A]\!]_\varepsilon & := & \mathrm{El}\ [A]_\varepsilon \\
[A]_\emptyset & := & \mathrm{Err}\ [A]_\varepsilon \\
[\![\bullet]\!]_\varepsilon & := & \bullet \\
[\![\Gamma,\ x^{\mathbb{K}/\mathbb{T}} : A]\!]_\varepsilon & := & [\![\Gamma]\!]_\varepsilon,\ x^{\mathbb{T}} : [\![A]\!]_\varepsilon \\
[\![\Gamma,\ x^{\mathbb{G}/\mathbb{P}} : A]\!]_\varepsilon & := & [\![\Gamma]\!]_\varepsilon
\end{array}
$$

Fig. 4. Erasure translation

3.1 Erasure [`Erasure.erase_term`]

Erasure removes all proofs (of propositions) and ghost values from terms while preserving typing. Essentially a term $t : A$ is erased to $[t]_\varepsilon : [\![A]\!]_\varepsilon$ and we have a distinguished inhabitant for each translated type $[A]_\emptyset : [\![A]\!]_\varepsilon$ corresponding to the exception.

To perform the translation, we require two inductive types in the target: the the unit type with inhabitant $() : \mathrm{unit}$; and a data type for representing universes ty_i for each i. ty has one constructor to embed pointed types $\mathrm{tyval} : \forall(A : \mathrm{Type}_i)\ (a : A).\ \mathrm{ty}_i$, and an error constructor $\mathrm{ty}_\emptyset : \mathrm{ty}_i$ to make ty_i itself a pointed type. Using its eliminator we can define two *projections* $\mathrm{El} : \mathrm{ty}_i \to \mathrm{Type}_i$ and $\mathrm{Err} : \forall(T : \mathrm{ty}_i).\mathrm{El}_i\ T$ such that we have the following computation rules (following the construction of Pédrot and Tabareau):

$$\mathrm{El}\ (\mathrm{tyval}\ A\ a) \equiv A \qquad \mathrm{Err}\ (\mathrm{tyval}\ A\ a) \equiv a \qquad \mathrm{El}\ \mathrm{ty}_\emptyset \equiv \mathrm{unit} \qquad \mathrm{Err}\ \mathrm{ty}_\emptyset \equiv ()$$

In Figure 4, we syntactically define erasure with the idea that it should only operate on relevant terms (whose mode is $\mathbb{T}$ or $\mathbb{K}$). In order to handle partiality of the function we return a dummy term $\blacksquare$ in the cases we wish to discard.[7] Herein we write $\mathbb{K}/\mathbb{T}$ to mean $\mathbb{T}$ or $\mathbb{K}$, similarly for $\mathbb{G}/\mathbb{P}$ and so on. We also write $t^{\mathfrak{s}}$ to mean that we match on t such that $\mathrm{md}(t) = \mathfrak{s}$ to avoid cluttering the definition. Note that erasure actually depends on the context, but we omit it for conciseness.

[7] In the formalisation we have chosen $()$ but any closed term would have worked.

Note that the universe of propositions and ghost types as well as propositions and ghost types themselves are not irrelevant so they need to have an erasure as well. Since we don't care about the content of propositions, it is not necessary to erase them to types. We thus erase Prop to unit and propositions to (). It is then easy to provide erasure for Reveal as we can simply use (). In contrast, we do preserve ghost types even if their values are erased. For them, erasure essentially recovers the associated relevant type, but removing dependencies on ghost values since these values are removed from the context. Also notice the translation of exfalso which may not use the proof of $\bot$ given it is erased so instead raises an exception, declaring that the branch is inaccessible.

We now show that erasure is sound through the following lemmas: two syntactical properties of erasure in relation with substitution and cast removal; and the fact that it preserves typing.

Lemma 3.1. [Erasure.erase_subst] *Erasure commutes with substitution: assuming* $\Gamma \vdash u :: \mathfrak{s}$,

- $[t[x := u]]_\varepsilon =_\alpha [t]_\varepsilon[x := [u]_\varepsilon]$, *when* $\mathfrak{s} \in \{\mathbb{T}, \mathbb{K}\}$.
- $[t[x := u]]_\varepsilon =_\alpha [t]_\varepsilon$, *when* $\mathfrak{s} \in \{\mathbb{P}, \mathbb{G}\}$.

Lemma 3.2. [Erasure.erase_castrm] *Erasure ignores casts:* $[|u|]_\varepsilon =_\alpha [u]_\varepsilon$.

Lemma 3.3. [Erasure.erase_typing, Erasure.erase_conv, Erasure.erase_context]

- *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]_\varepsilon$.
- *If* $\Gamma \vdash u \equiv v$ *then* $[\![\Gamma]\!]_\varepsilon \vdash [u]_\varepsilon \equiv [v]_\varepsilon$.
- *If* $\Gamma \vdash t : A$ *and* $\mathrm{md}(t) \in \{\mathbb{T}, \mathbb{K}\}$ *then* $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$.

A direct corollary is that whenever $\Gamma \vdash A : \mathrm{Sort}^{\mathbb{K}/\mathbb{T}/\mathbb{G}}$ and $\mathrm{md}(A) = \mathbb{K}$ then $[\![\Gamma]\!]_\varepsilon \vdash [A]_\emptyset : [\![A]\!]_\varepsilon$; another is that $A \equiv B$ implies $[A]_\emptyset \equiv [B]_\emptyset$. We use these facts in the proof.

Proof. The first item is a corollary of the two others that we both prove by induction on the derivation, using Lemmas 3.2 and 3.3. Note that we make use of the fact that conversion is untyped as well in the target so that $\blacksquare \equiv \blacksquare$ holds. For instance to show that $[\mathrm{reveal}(\mathrm{hide}\ t, P, p)]_\varepsilon \equiv [p\ t]_\varepsilon$ holds, we remark that both sides have mode $\mathbb{P}$ or $\mathbb{G}$ so we know both sides erase to $\blacksquare$. For the congruence rules of conversion, Lemma 2.4 is useful to make sure both sides are indeed erased in the same way so that the induction hypotheses are enough to conclude. □

3.2 Revival [Revival.revive_term]

Erasure preserves ghost types to some extent but not their inhabitants, and the parametricity translation will need to recover this information. We thus need to consider an extra translation we call *revival* to echo *resurrection* of irrelevant variables in a context as introduced by Pfenning [2001]: we *revive* values that have been erased. We define $[\![-]\!]_v$ in Figure 5 with the idea that whenever $\Gamma \vdash t : A$ with $\mathrm{md}(t) = \mathbb{G}$ then $[\![\Gamma]\!]_v \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$ (notice the type is the one from erasure).

We now prove soundness of revival. We first make an observation about context that says that when $\Gamma \vdash t : A$ then $[t]_\varepsilon$ still makes sense in $[\![\Gamma]\!]_v$ (*i.e.* is implicitly lifted to it). The rest is similar to what we did for erasure.

Lemma 3.4. [Revival.type_to_rev] $[\![\Gamma]\!]_\varepsilon$ *is a sub-context of* $[\![\Gamma]\!]_v$

Lemma 3.5. [Revival.revive_subst] *Revival commutes with substitution: assuming* $\Gamma \vdash u :: \mathfrak{s}$,

- $[\![t[x := u]]\!]_v =_\alpha [\![t]\!]_v[x := [u]_\varepsilon]$, *when* $\mathfrak{s} \in \{\mathbb{T}, \mathbb{K}\}$.
- $[\![t[x := u]]\!]_v =_\alpha [\![t]\!]_v[x := [\![u]\!]_v]$, *when* $\mathfrak{s} = \mathbb{G}$.
- $[\![t[x := u]]\!]_v =_\alpha [\![t]\!]_v$, *when* $\mathfrak{s} = \mathbb{P}$.

Lemma 3.6. [Revival.revive_castrm] *Revival ignores casts:* $[\![|u|]\!]_v =_\alpha [\![u]\!]_v$.

Lemma 3.7. [Revival.revive_typing, Revival.revive_conv, Revival.revive_context]

$$
\begin{array}{lcl}
[\![x^{\mathbb{G}}]\!]_v & := & x \\
[\![\lambda(x^{\mathbb{K}/\mathbb{T}/\mathbb{G}} : A).t^{\mathbb{G}}]\!]_v & := & \lambda(x : [\![A]\!]_\varepsilon).[\![t]\!]_v \\
[\![\lambda(x^{\mathbb{P}} : A).t^{\mathbb{G}}]\!]_v & := & [\![t]\!]_v \\
[\![t^{\mathbb{G}}\ u^{\mathbb{K}/\mathbb{T}}]\!]_v & := & [\![t]\!]_v\ [u]_\varepsilon \\
[\![t^{\mathbb{G}}\ u^{\mathbb{G}}]\!]_v & := & [\![t]\!]_v\ [\![u]\!]_v \\
[\![t^{\mathbb{G}}\ u^{\mathbb{P}}]\!]_v & := & [\![t]\!]_v \\
[\![\mathsf{hide}\ t]\!]_v & := & [t]_\varepsilon \\
[\![\mathsf{reveal}\ t\ P\ p^{\mathbb{G}}]\!]_v & := & [\![p]\!]_v\ [\![t]\!]_v \\
[\![\mathsf{cast}_{A,u,v}\ e\ P\ t]\!]_v & := & [\![t]\!]_v \\
[\![\mathsf{exfalso}_{\mathbb{G}}\ A\ p]\!]_v & := & [A]_\emptyset \\
[\![t^{\mathbb{K}/\mathbb{T}/\mathbb{P}}]\!]_v & := & \blacksquare \\
 & & \\
[\![\Gamma,\ x^{\mathbb{K}/\mathbb{T}/\mathbb{G}} : A]\!]_v & := & [\![\Gamma]\!]_v,\ x^{\mathbb{T}} : [\![A]\!]_\varepsilon \\
[\![\Gamma,\ x^{\mathbb{P}} : A]\!]_v & := & [\![\Gamma]\!]_v
\end{array}
$$

Fig. 5. Revival translation

$$\frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma \vdash \|A\| : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{sq}\ t : \|A\|}$$

$$\frac{\Gamma \vdash e : \|A\| \qquad \Gamma \vdash P : \|A\| \to \mathsf{Prop} \qquad \Gamma \vdash t : \forall x.\ P\ (\mathsf{sq}\ x)}{\Gamma \vdash \mathsf{sq\text{-}elim}\ e\ P\ t : P\ e}$$

Fig. 6. Propositional truncation (squash) in the target

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u =_A v : \mathsf{Type}_i} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{refl}_A\ u : u =_A u}$$

$$\frac{\Gamma \vdash e : u =_A v \qquad \Gamma \vdash P : \forall (x : A).\ u =_A x \to s \qquad \Gamma \vdash t : P\ u\ (\mathsf{refl}_A\ u)}{\Gamma \vdash \mathsf{J}\ e\ P\ t : P\ v\ e} \qquad \frac{}{\mathsf{J}(\mathsf{refl}_A\ u, P, t) \equiv t}$$

Fig. 7. Propositional equality in the target

- *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]_v$.
- *If* $\Gamma \vdash u \equiv v$ *then* $[\![\Gamma]\!]_v \vdash [\![u]\!]_v \equiv [\![v]\!]_v$.
- *If* $\Gamma \vdash t : A$ *and* $\mathrm{md}(t) = \mathbb{G}$ *then* $[\![\Gamma]\!]_v \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$.

3.3 Parametricity [Param.param_term]

We are now ready to perform the parametricity translation. We follow Keller and Lasson [2012]—hence the use of Kind—and interpret types by propositional predicates. Propositions are also interpreted as propositions so in particular $u \approx v$ must be interpreted by a proposition. Having propositional equality in a definitional proof-irrelevant Prop is a notorious problem [Abel and Coquand 2020] and while there are options to circumvent the issue [Pujet and Tabareau 2022, 2023], we decide instead to simply squash an equality type that might live in Type. To that end we

assume the target theory features propositional truncation (or squash) described in Figure 6, and (proof-relevant) equality, given in Figure 7.

The parametricity translation $[\![-]\!]_{\mathcal{P}}$ is given in Figure 8 except for the translation of casts which is a bit more involved and which we will now describe. We refer the reader to the formalisation for the full definition. $[\![\mathrm{cast}_{A,u,v}\ e\ P\ t]\!]_{\mathcal{P}}$ depends on the mode of t so we will only consider the case when it is $\mathbb{T}$, the other cases are similar. Let us start by computing the expected type for $[\![\mathrm{cast}\ e\ P\ t]\!]_{\mathcal{P}}$:

$$\begin{aligned}[\![P\ v]\!]_{\mathcal{P}}\ [\mathrm{cast}\ e\ P\ t]_{\varepsilon} &= [\![P]\!]_{\mathcal{P}}\ [\![v]\!]_{v}\ [\![v]\!]_{\mathcal{P}}\ [\mathrm{cast}\ e\ P\ t]_{\varepsilon} \\ &= [\![P]\!]_{\mathcal{P}}\ [\![v]\!]_{v}\ [\![v]\!]_{\mathcal{P}}\ [t]_{\varepsilon}\end{aligned}$$

We also compute the type of $[\![P]\!]_{\mathcal{P}}$:

$$\begin{aligned}[\![A \to \mathrm{Type}]\!]_{\mathcal{P}}\ [P]_{\varepsilon} &= \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![\mathrm{Type}]\!]_{\mathcal{P}}\ [P]_{\varepsilon} \\ &= \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![P]\!]_{\varepsilon} \to \mathrm{Prop}\end{aligned}$$

This confirms that $[\![P]\!]_{\mathcal{P}}\ [\![v]\!]_{v}\ [\![v]\!]_{\mathcal{P}}\ [t]_{\varepsilon}$ is a proposition so we can prove it by eliminating the squash from $[\![e]\!]_{\mathcal{P}}$ which is of type $\|[\![u]\!]_{v} = [\![v]\!]_{v}\|$, and then use J eliminator to also rewrite the equality $[\![u]\!]_{v} = [\![v]\!]_{v}$ in the type of $[\![t]\!]_{\mathcal{P}}$ which is $[\![P]\!]_{\mathcal{P}}\ [\![u]\!]_{v}\ [\![u]\!]_{\mathcal{P}}\ [t]_{\varepsilon}$. We can swap $[\![u]\!]_{\mathcal{P}}$ for $[\![v]\!]_{\mathcal{P}}$ because after rewriting they are both proofs of the same proposition: $[\![A]\!]_{\mathcal{P}}\ [\![v]\!]_{v}$.

Soundness of parametricity is very similar to that of erasure and revival.

Lemma 3.8. [Param.typing_rev_sub_param] *$[\![\Gamma]\!]_{v}$ is a sub-context of $[\![\Gamma]\!]_{\mathcal{P}}$.*

Combined with Lemma 3.4, $[\![\Gamma]\!]_{\varepsilon}$ is thus also a sub-context of $[\![\Gamma]\!]_{\mathcal{P}}$. [Param.typing_er_sub_param]

Lemma 3.9. [Param.param_subst] *Parametricity commutes with substitution: assuming $\Gamma \vdash u :: \mathfrak{s}$*

- *If $\mathfrak{s} \in \{\mathbb{K}, \mathbb{T}\}$ then $[\![t[x := u]]\!]_{\mathcal{P}} =_{\alpha} [\![t]\!]_{\mathcal{P}}[x := [u]_{\varepsilon}, \overline{x} := [\![u]\!]_{\mathcal{P}}]$.*
- *If $\mathfrak{s} = \mathbb{G}$ then $[\![t[x := u]]\!]_{\mathcal{P}} =_{\alpha} [\![t]\!]_{\mathcal{P}}[x := [\![u]\!]_{v}, \overline{x} := [\![u]\!]_{\mathcal{P}}]$.*
- *If $\mathfrak{s} = \mathbb{P}$ then $[\![t[x := u]]\!]_{\mathcal{P}} =_{\alpha} [\![t]\!]_{\mathcal{P}}[x := [\![u]\!]_{\mathcal{P}}]$.*

Similarly to erasure and revival, we show that when two terms are the same up to casts, then their parametricity translations are equal. Before, we proved the translations were syntactically equal, this time we only prove definitional equality as we need to exploit proof irrelevance. Before we do that we thus establish that the parametricity translation produces terms of the right modes.

Lemma 3.10. [Param.param_scoping] *Parametricity produces consistent modes:*

- *If $\Gamma \vdash t :: \mathbb{K}$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash t :: \mathbb{T}$.*
- *If $\Gamma \vdash t :: \mathfrak{s}$ with $\mathfrak{s} \in \{\mathbb{P}, \mathbb{G}, \mathbb{T}\}$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash t :: \mathbb{P}$.*

Lemma 3.11. [Param.param_castrm] *Parametricity ignores casts: if $\Gamma \vdash t :: \mathfrak{s}$ then $[\![|u|]\!]_{\mathcal{P}} \equiv [\![v]\!]_{\mathcal{P}}$.*

Proof. The proof essentially makes repeated use of congruence rules of conversion and uses proof irrelevance to get rid of the translation of casts. □

Lemma 3.12. [Param.param_typing, Param.param_conv, Param.param_context]

- *If $\vdash \Gamma$ then $\vdash [\![\Gamma]\!]_{\mathcal{P}}$.*
- *If $\Gamma \vdash u \equiv v : A$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash [\![u]\!]_{\mathcal{P}} \equiv [\![v]\!]_{\mathcal{P}}$.*
- *If $\mathrm{md}(t) \in \{\mathbb{K}, \mathbb{T}\}$ and $\Gamma \vdash t : A$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash [\![t]\!]_{\mathcal{P}} : [\![A]\!]_{\mathcal{P}}\ [t]_{\varepsilon}$.*
- *If $\mathrm{md}(t) = \mathbb{G}$ and $\Gamma \vdash t : A$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash [\![t]\!]_{\mathcal{P}} : [\![A]\!]_{\mathcal{P}}\ [\![t]\!]_{v}$.*
- *If $\mathrm{md}(t) = \mathbb{P}$ and $\Gamma \vdash t : A$ then $[\![\Gamma]\!]_{\mathcal{P}} \vdash [\![t]\!]_{\mathcal{P}} : [\![A]\!]_{\mathcal{P}}$.*

$$
\begin{array}{lcl}
[\![x^{\mathbb{K}/\mathbb{T}/\mathbb{G}}]\!]_{\mathcal{P}} & := & \overline{x} \\
[\![x^{\mathbb{P}}]\!]_{\mathcal{P}} & := & x \\
[\![\mathrm{Kind}_i]\!]_{\mathcal{P}} & := & \lambda A.\ \mathrm{El}\ A \to \mathrm{Type}_i \\
[\![\mathrm{Sort}_i^{\mathbb{T}/\mathbb{G}}]\!]_{\mathcal{P}} & := & \lambda A.\ \mathrm{El}\ A \to \mathrm{Prop} \\
[\![\mathrm{Prop}]\!]_{\mathcal{P}} & := & \lambda\_.\ \mathrm{Prop} \\
[\![\forall^{\mathbb{K}/\mathbb{T}}(x^{\mathbb{K}/\mathbb{T}} : A).B]\!]_{\mathcal{P}} & := & \lambda f.\ \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}}\ (f\ x) \\
[\![\forall^{\mathbb{K}/\mathbb{T}}(x^{\mathbb{G}} : A).B]\!]_{\mathcal{P}} & := & \lambda f.\ \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}}\ f \\
[\![\forall^{\mathbb{K}/\mathbb{T}}(x^{\mathbb{P}} : A).B]\!]_{\mathcal{P}} & := & \lambda f.\ [\![A]\!]_{\mathcal{P}} \to [\![B]\!]_{\mathcal{P}}\ f \\
[\![\forall^{\mathbb{G}}(x^{\mathbb{K}/\mathbb{T}/\mathbb{G}} : A).B]\!]_{\mathcal{P}} & := & \lambda f.\ \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}}\ (f\ x) \\
[\![\forall^{\mathbb{G}}(x^{\mathbb{P}} : A).B]\!]_{\mathcal{P}} & := & \lambda f.\ \forall (x : [\![A]\!]_{\mathcal{P}}).\ [\![B]\!]_{\mathcal{P}}\ f \\
[\![\forall^{\mathbb{P}}(x^{\mathbb{K}/\mathbb{T}/\mathbb{G}} : A).B]\!]_{\mathcal{P}} & := & \forall (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}} \\
[\![\forall^{\mathbb{P}}(x^{\mathbb{P}} : A).B]\!]_{\mathcal{P}} & := & \forall (x : [\![A]\!]_{\mathcal{P}}).\ [\![B]\!]_{\mathcal{P}} \\
[\![\lambda(x^{\mathbb{K}/\mathbb{T}/\mathbb{G}} : A).t]\!]_{\mathcal{P}} & := & \lambda (x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![t]\!]_{\mathcal{P}} \\
[\![\lambda(x^{\mathbb{P}} : A).t]\!]_{\mathcal{P}} & := & \lambda (x : [\![A]\!]_{\mathcal{P}}).\ [\![t]\!]_{\mathcal{P}} \\
[\![t\ u^{\mathbb{K}/\mathbb{T}}]\!]_{\mathcal{P}} & := & [\![t]\!]_{\mathcal{P}}\ [u]_{\varepsilon}\ [\![u]\!]_{\mathcal{P}} \\
[\![t\ u^{\mathbb{G}}]\!]_{\mathcal{P}} & := & [\![t]\!]_{\mathcal{P}}\ [\![u]\!]_{v}\ [\![u]\!]_{\mathcal{P}} \\
[\![t\ u^{\mathbb{P}}]\!]_{\mathcal{P}} & := & [\![t]\!]_{\mathcal{P}}\ [\![u]\!]_{\mathcal{P}} \\
[\![\mathrm{erased}\ A^{\mathbb{K}}]\!]_{\mathcal{P}} & := & [\![A]\!]_{\mathcal{P}} \\
[\![\mathrm{hide}\ t^{\mathbb{T}}]\!]_{\mathcal{P}} & := & [\![t]\!]_{\mathcal{P}} \\
[\![\mathrm{reveal}\ t\ P\ p^{\mathbb{G}/\mathbb{P}}]\!]_{\mathcal{P}} & := & [\![p]\!]_{\mathcal{P}}\ [\![t]\!]_{v}\ [\![t]\!]_{\mathcal{P}} \\
[\![\mathrm{Reveal}\ t\ p^{\mathbb{K}}]\!]_{\mathcal{P}} & := & [\![p]\!]_{\mathcal{P}}\ [\![t]\!]_{v}\ [\![t]\!]_{\mathcal{P}} \\
[\![\mathrm{toRev}_{t,p}\ u]\!]_{\mathcal{P}} & := & [\![u]\!]_{\mathcal{P}} \\
[\![\mathrm{fromRev}_{t,p}\ u]\!]_{\mathcal{P}} & := & [\![u]\!]_{\mathcal{P}} \\
[\![u \approx_A v]\!]_{\mathcal{P}} & := & \|[\![u]\!]_{v} =_{[\![A]\!]_{\varepsilon}} [\![v]\!]_{v}\| \\
[\![\mathrm{gh\text{-}refl}_A\ u]\!]_{\mathcal{P}} & := & \mathrm{sq}\ (\mathrm{refl}_{[\![A]\!]_{\varepsilon}}\ [\![u]\!]_{v}) \\
[\![\mathrm{cast}_{A,u,v}\ e\ P\ t]\!]_{\mathcal{P}} & := & \text{(explained in the text, given in Coq)} \\
[\![\bot]\!]_{\mathcal{P}} & := & \bot \\
[\![\mathrm{exfalso}_{\mathbb{K}/\mathbb{T}/\mathbb{G}}\ A\ p]\!]_{\mathcal{P}} & := & \mathrm{exfalso}\ ([\![A]\!]_{\mathcal{P}}\ [A]_{\emptyset})\ [\![p]\!]_{\mathcal{P}} \\
[\![\mathrm{exfalso}_{\mathbb{P}}\ A\ p]\!]_{\mathcal{P}} & := & \mathrm{exfalso}\ [\![A]\!]_{\mathcal{P}}\ [\![p]\!]_{\mathcal{P}} \\
[\![\_]\!]_{\mathcal{P}} & := & \blacksquare
\end{array}
$$

$$
\begin{array}{lcl}
[\![\bullet]\!]_{\mathcal{P}} & := & \bullet \\
[\![\Gamma,\ x^{\mathbb{K}} : A]\!]_{\mathcal{P}} & := & [\![\Gamma]\!]_{\mathcal{P}},\ x^{\mathbb{T}} : [\![A]\!]_{\varepsilon},\ \overline{x}^{\mathbb{T}} : [\![A]\!]_{\mathcal{P}}\ x^{\mathbb{T}} \\
[\![\Gamma,\ x^{\mathbb{T}/\mathbb{G}} : A]\!]_{\mathcal{P}} & := & [\![\Gamma]\!]_{\mathcal{P}},\ x^{\mathbb{T}} : [\![A]\!]_{\varepsilon},\ \overline{x}^{\mathbb{P}} : [\![A]\!]_{\mathcal{P}}\ x^{\mathbb{T}} \\
[\![\Gamma,\ x^{\mathbb{P}} : A]\!]_{\mathcal{P}} & := & [\![\Gamma]\!]_{\mathcal{P}},\ x^{\mathbb{P}} : [\![A]\!]_{\mathcal{P}}
\end{array}
$$

Fig. 8. Parametricity translation

4 META-THEORETICAL CONSEQUENCES OF THE MODEL

As we explained before, our target is standard enough to be a subset of Coq or Agda with cumulativity. As such, we argue it is safe to assume it is consistent and enjoys various properties such as type former discrimination (two different type formers are never convertible) or injectivity of constructors. Assuming this, we can lift certain properties back to GTT thanks to our translations.

Mode injectivity. [`Model.sort_mode_inj`] Before we prove relative consistency, we are going to prove that $\Gamma \vdash \mathrm{Sort}_i^{\mathfrak{s}} \equiv \mathrm{Sort}_j^{\mathfrak{r}}$ entails $\mathfrak{s} = \mathfrak{r}$ which combined with Lemma 2.6 will ensure *e.g.* that

a proof of $\bot$ is indeed in mode $\mathbb{P}$. To do this we instrument erasure translation of sorts so that it carries extra information about the mode which it carried before. We do so by adding one extra argument to tyval which is essentially a mode but which is ignored by El and Err so it doesn't affect the proof in any way. Assuming injectivity for the tyval constructor, we are able to deduce the desired property.

Theorem 4.1 (Consistency). [Model.relative_consistency] *GTT is consistent.*

Proof. Assuming there is a proof of $\bot$ in the empty context GTT, we then translate $\vdash p : \bot$ to $\vdash [\![p]\!]_{\mathcal{P}} : [\![\bot]\!]_{\mathcal{P}}$ and since $[\![\bot]\!]_{\mathcal{P}}$ is $\bot$ we get a contradiction in the target. □

Another feature which we advocated for in the introduction was the ability to discriminate type formers, and we can use the model to lift this property back to GTT. It in fact generalises the property we showed about sorts. Note that beyond sorts, we have not formalised this theorem further due to the quadratic number of cases we would have to consider.

Theorem 4.2 (Discrimination of type formers). *GTT can discriminate type formers. Concretely we can disprove conversions involving different type formers at the head, such as:*

- $\forall^{\mathsf{r}}(x^{\mathsf{s}} : A).B \not\equiv \mathrm{Type}$
- $\mathrm{Kind}_i \not\equiv \mathrm{Type}_i$
- $\mathrm{Type}_i \not\equiv \mathrm{Ghost}_i$
- $\mathrm{Type}_i \not\equiv \mathrm{Prop}$
- $X \not\equiv \mathrm{Type}_i$
- ...

Proof. The idea is to remark that such type formers are distinguished in the model, either by using the erasure (Lemma 3.3) or the parametricity (Lemma 3.12) translations, depending on the case. Let us look precisely at some representative cases.

Let us start with $\forall^{\mathbb{K}}(x^{\mathbb{T}} : A).B$. Assuming $\forall^{\mathbb{K}}(x^{\mathbb{T}} : A).B \equiv \mathrm{Type}$, by applying erasure, we get

$$\mathrm{tyval}\ (\forall(x : [\![A]\!]_{\varepsilon}).[\![B]\!]_{\varepsilon})\ (\lambda x.[B]_{\emptyset}) \equiv \mathrm{tyval}\ \mathrm{ty}_i\ \mathrm{ty}_{\emptyset}$$

then by congruence of El and its computation rules we get

$$\forall(x : [\![A]\!]_{\varepsilon}).[\![B]\!]_{\varepsilon} \equiv \mathrm{ty}_i$$

which is not possible because the target discriminates type formers.

Now, for $\forall^{\mathbb{K}}(x^{\mathbb{G}} : A).B$, the above does not work since it is not erased to a dependent function type. There we rely instead on parametricity. So assuming $\forall^{\mathbb{K}}(x^{\mathbb{G}} : A).B \equiv \mathrm{Type}$, we get

$$\lambda f.\ \forall(x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}}\ f \equiv \lambda A.\ \mathrm{El}\ A \to \mathrm{Prop}$$

By weakening, we can add some variable $f : [\![B]\!]_{\varepsilon}$, and use congruence of application and β-reduction to obtain

$$\forall(x : [\![A]\!]_{\varepsilon})\ (\overline{x} : [\![A]\!]_{\mathcal{P}}\ x).\ [\![B]\!]_{\mathcal{P}}\ f \equiv \mathrm{El}\ f \to \mathrm{Prop}$$

which the target also discriminates (two function types against a function type with codomain Prop). Notice how the two sides are not even of the same type to begin with so the proof also exploits the fact that conversion is untyped. □

We argued already that this property is useful in practice: it allows for early returns in conversion algorithms and it helps avoiding cluttering user feedback with nonsensical error messages.

Coherence of modes. One last interesting property we wish to prove is that whenever $t : A$ and $A : \mathrm{Sort}^{\mathfrak{s}}$ then $t :: \mathfrak{s}$. To establish this property we however need to assume injectivity of $\forall$ *in the source*. Unfortunately, we are unable to derive this property from the model, and so it would require a different approach to derive (such as characterising conversion with a reduction relation that is shown confluent, or by using logical relations). We believe our conversion is close enough to that of Gilbert et al. [2019] that we can safely assume it and we leave a formal proof to future work. We in fact need only assume the following.

Conjecture 4.3. *If* $\Gamma \vdash \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A).B \equiv \forall^{\mathfrak{r}}_{i,j}(x^{\mathfrak{s}} : A').B'$ *then* $\Gamma, x^{\mathfrak{s}} : A \vdash B \equiv B'$.

This assumption in turn lets us conclude a form of uniqueness of type: when $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then A and B are convertible up to universe levels and casts. We can then conclude our theorem which justifies the simplified rules presented in the paper, compared to the rules we formalised.

Lemma 4.4 (Mode coherence). [`Model.mode_coherence`]
If $\vdash \Gamma$ *and* $\Gamma \vdash t : A$ *and* $\Gamma \vdash A : \mathrm{Sort}^{\mathfrak{s}}$ *then* $\Gamma \vdash t :: \mathfrak{s}$.

As we shall see in Section 5.1, the model also lets us derive free theorems as is usual with parametricity models.

5 EXTENDING GTT WITH INDUCTIVE TYPES

As we advertised in the introduction, ghost types really shine when used as indices to inductive types. We will thus show how to extend GTT and the translations of Section 3 to natural numbers (Section 5.2) and vectors (Section 5.3). We will however start with booleans in order to showcase how we can derive free theorems (Section 5.1). All three cases have been formalised.

We leave a more general treatment of inductive types for future work, but we have tried to make our treatment as generic as possible. We will see however that there are some surprises along the way, in the case of vectors, and it would be interesting to see how general this can be. It is also important to note that the inductive types we consider in GTT cannot be eliminated to sort Kind—in other words, there is no large elimination. This is a limitation coming from the fact that the parametricity translation produces propositions and that proofs cannot in general be eliminated to produce relevant information which is expected from the translation of Kind. Keller and Lasson [2012, Section 4.4] show how to circumvent this problem for certain inductive types but we consider it out of scope for the current paper.

5.1 A free theorem for booleans

We first show how to translate booleans. Given that they are very basic we can use them to illustrate how we derive free theorems with one example: functions of type erased bool → bool must be constant. In order to ease reading, we present the definition of booleans, their erasure and their parametricity predicate directly in Coq syntax in in Figure 9. The idea is pretty straightforward: $\mathbb{B}$ is erased to `tyval` $\mathbb{B}^\bullet$ $\mathbb{B}_\emptyset$ while `true` and `false` are erased to `true`$^\bullet$ and `false`$^\bullet$. Then the $\mathbb{B}^{\mathcal{P}}$ predicate is only verified by exception-free booleans, *i.e.* those obtained from `true` and `false`. The eliminator is essentially translated using the eliminators of $\mathbb{B}^\bullet$ and $\mathbb{B}^{\mathcal{P}}$ and we refer the reader to either Pédrot and Tabareau [2018] or the formalisation for details.

We now focus on the free theorem. We want to show that our models supports the following proposition: $\forall(f : \mathrm{erased\ bool} \to \mathrm{bool}),\ f\ (\mathrm{hide\ true}) = f\ (\mathrm{hide\ false})$. Because we did not formalise equality, we resort to its impredicative encoding and show we support the following proposition instead:

$$\forall(f : \mathrm{erased\ bool} \to \mathrm{bool})\ (P : \mathrm{bool} \to \mathrm{Prop}),\ P\ (f\ (\mathrm{hide\ true})) \to P\ (f\ (\mathrm{hide\ false})).$$

```
Inductive 𝔹 :=
| true
| false.
```

```
Inductive 𝔹• :=
| true•
| false•
| 𝔹∅.
```

```
Inductive 𝔹^𝒫 : 𝔹• → SProp :=
| true^𝒫 : 𝔹^𝒫 true•
| false^𝒫 : 𝔹^𝒫 false•.
```

Fig. 9. Booleans and their translations

```
Inductive ℕ :=
| O
| S (n : ℕ).
```

```
Inductive ℕ• :=
| O•
| S• (n : ℕ•)
| ℕ∅.
```

```
Inductive ℕ^𝒫 : ℕ• → SProp :=
| O^𝒫 : ℕ^𝒫 O•
| S^𝒫 : ∀ n, ℕ^𝒫 n → ℕ^𝒫 (S• n).
```

Fig. 10. Natural numbers and their translations

$$\frac{\Gamma \vdash n : \mathrm{nat} \qquad \Gamma \vdash P : \mathrm{nat} \to \mathrm{Sort}^{\mathfrak{r}} \qquad \Gamma \vdash z : P\ 0 \qquad \Gamma \vdash s : \forall^{\mathfrak{r}}(x^{\mathbb{T}} : \mathrm{nat}).\ P\ x \to P\ (\mathrm{S}\ x) \qquad \mathfrak{r} \neq \mathbb{K}}{\Gamma \vdash \mathrm{nat\text{-}elim}_{\mathfrak{r}}\ n\ P\ z\ s : P\ n}$$

$$\frac{\Gamma \vdash P :: \mathbb{K} \qquad \Gamma \vdash z :: \mathfrak{r} \qquad \Gamma \vdash s :: \mathfrak{r} \qquad \mathfrak{r} \neq \mathbb{K}}{\Gamma \vdash \mathrm{nat\text{-}elim}_{\mathfrak{r}}\ 0\ P\ z\ s \equiv z}$$

$$\frac{\Gamma \vdash n :: \mathbb{T} \qquad \Gamma \vdash P :: \mathbb{K} \qquad \Gamma \vdash z :: \mathfrak{r} \qquad \Gamma \vdash s :: \mathfrak{r} \qquad \mathfrak{r} \neq \mathbb{K}}{\Gamma \vdash \mathrm{nat\text{-}elim}_{\mathfrak{r}}\ (\mathrm{S}\ n)\ P\ z\ s \equiv s\ n\ (\mathrm{nat\text{-}elim}_{\mathfrak{r}}\ n\ P\ z\ s)}$$

Fig. 11. Eliminator for natural numbers

Since this is a proposition, we don't need to inhabit its erasure, the only thing we need is show its parametricity is inhabited in the model. In Coq syntax, the theorem we have to prove is the following:

```
∀ (fe : 𝔹•) (fP : ∀ (b : 𝔹•) (bP : 𝔹^𝒫 b), 𝔹^𝒫 fe) (P : 𝔹• → unit) (PP : ∀ (b : 𝔹•) (bP : 𝔹^𝒫 b), SProp),
  PP fe (fP true• true^𝒫) → PP fe (fP false• false^𝒫).
```

which is proven trivially [FreeTheorem.constant_free_theorem] by exploiting proof irrelevance.

5.2 Natural numbers

A second example is natural numbers which is also rather straightforward. With them we also see how one can lift usual results on natural numbers to their erased counterparts without having to reprove them. We give the Coq inductive definition of ℕ and its translations in Figure 10.

5.2.1 Translating the eliminator. As for booleans, the only slight difficulty is translating the eliminator. This time we are going to give more details. First, we give the typing and computation rules for the eliminator in Figure 11. They are rather standard except for the scoping conditions; their presence ensures that the parametricity translation of both sides are proof of propositions, letting us use proof irrelevance to model the computation rules for the parametricity translation.

For erasure, we only have to consider the case where $\mathfrak{r} = \mathbb{T}$ as it's the only relevant case.[8] We build the following term using the induction principle of ℕ• (it is mostly a reformulation of it).

```
ℕ_elim• : ∀ (P : ℕ• → ty) (z : El (P O•)) (s : ∀ (n : El ℕ•), El (P n) → El (P (S• n))) (n : El ℕ•), El (P n).
```

[8]Remember, we disallow elimination to Kind.

We then verify it interprets the conversion rules of nat-elim:

```
ℕ_elim• P z s 0• ≡ z
ℕ_elim• P z s (S• n) ≡ s n (ℕ_elim• P z s n)
```

For revival, similarly we only have one case to consider: this time $\mathfrak{r} = \mathbb{G}$. The type we have to inhabit is exactly the same so we simply reuse ℕ_elim•. The equations are obviously also verified.

For parametricity, we can again provide the same translation for both the $\mathbb{T}$ and $\mathbb{G}$ cases, the only difference being that one uses erasure of the branches $[z]_\varepsilon$ and $[s]_\varepsilon$, while the other one uses revival $[\![z]\!]_v$ and $[\![s]\!]_v$. The $\mathbb{P}$ case is similar but simpler. The properties we need to prove are the following, they follow by a simple induction on the parametricity proof of type $\mathbb{N}^{\mathcal{P}}$ n.

```
∀ (Pe : ℕ• → ty) (PP : ∀ n (nP : ℕ^P n), El (Pe n) → SProp)
 (ze : El (Pe 0•)) (zP : PP 0• 0^P ze)
 (se : ∀ n, El (Pe n) → El (Pe (S• n)))
 (sP : ∀ n nP (h : El (Pe n)) (hP : PP n nP h), PP (S• n) (S^P n nP) (se n h))
 n (nP : ℕ^P n),
 PP n nP (ℕ_elim• Pe ze se n).
```

```
∀ (Pe : ℕ• → unit) (PP : ∀ n (nP : ℕ^P n), SProp)
 (z : PP 0• 0^P)
 (s : ∀ n nP (h : PP n nP), PP (S• n) (S^P n nP))
 n (nP : ℕ^P n),
 PP n nP.
```

As mentioned before, we need not worry about the computation rules since these are strict propositions which enjoy definitional proof irrelevance.

This concludes the proof that the model supports natural numbers. In the formalisation we assume that the constants we have proven in Coq (they are found in the `TransNat` module) exist in the target and we use them to formalise the proof.

Before we move on to vectors, we will show a few examples that will prove useful later, and that already illustrate what can be achieved with erased natural numbers.

5.2.2 Erased successor. We can define a wrapper around the successor function for erased natural numbers by using reveal.

```
gS  :  erased nat → erased nat
gS  :=  λn. reveal n (λ_. erased nat) (λm. hide (S m))
```

If we were to use a style closer to pattern matching we would directly write:

```
gS           :  erased nat → erased nat
gS (hide n)  :=  hide (S m)
```

In fact, we have reason to believe that in practice hide and reveal can be inferred—as is already the case in F*—and a user would then write the following.

```
gS    :  erased nat → erased nat
gS n  :=  S m
```

In the same fashion we could define addition for erased natural numbers by lifting regular addition and lift the usual properties by the appropriate uses of reveal. Once again, F* demonstrates that it works in practice as the SMT solver can leverage equational reasoning on natural numbers to rewrite in erased natural numbers seamlessly.

$$t, u, A, B ::= \cdots \mid \mathrm{vec}\ A\ n \mid \mathrm{vnil}_A \mid \mathrm{vcons}\ a\ n\ v \mid \text{vec-elim}_{\mathfrak{s}}\ v\ P\ z\ s$$

$$\mathrm{md}(\mathrm{vec}\ A\ n) = \mathbb{K} \qquad \mathrm{md}(\mathrm{vnil}_A) = \mathrm{md}(\mathrm{vcons}\ a\ n\ v) = \mathbb{T} \qquad \mathrm{md}(\text{vec-elim}_{\mathfrak{s}}\ v\ P\ z\ s) = \mathfrak{s}$$

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \qquad \Gamma \vdash n : \mathrm{erased\ nat}}{\Gamma \vdash \mathrm{vec}\ A\ n : \mathrm{Type}_i} \qquad \frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \mathrm{vnil}_A : \mathrm{vec}\ A\ (\mathrm{hide}\ 0)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash v : \mathrm{vec}\ A\ n}{\Gamma \vdash \mathrm{vcons}\ a\ n\ v : \mathrm{vec}\ A\ (\mathrm{gS}\ n)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathrm{vec}\ A\ n \qquad \Gamma \vdash P : \forall (m : \mathrm{erased\ nat}).\ \mathrm{vec}\ A\ m \to \mathrm{Sort}^{\mathfrak{s}} \\ \Gamma \vdash z : P\ (\mathrm{hide}\ 0)\ \mathrm{vnil} \qquad \Gamma \vdash s : \forall a\ m\ w.\ P\ m\ w \to P\ (\mathrm{gS}\ m)\ (\mathrm{vcons}\ a\ m\ w) \qquad \mathfrak{s} \neq \mathbb{K}\end{array}}{\Gamma \vdash \text{vec-elim}_{\mathfrak{s}}\ v\ P\ z\ s : P\ n\ v}$$

$$\frac{\Gamma \vdash A :: \mathbb{K} \qquad \Gamma \vdash P :: \mathbb{K} \qquad \Gamma \vdash z :: \mathfrak{s} \qquad \Gamma \vdash s :: \mathfrak{s} \qquad \mathfrak{s} \neq \mathbb{K}}{\Gamma \vdash \text{vec-elim}_{\mathfrak{s}}\ (\mathrm{vnil}_A)\ P\ z,\ s \equiv z}$$

$$\frac{\Gamma \vdash a :: \mathbb{T} \qquad \Gamma \vdash n :: \mathbb{G} \qquad \Gamma \vdash v :: \mathbb{T} \qquad \Gamma \vdash P :: \mathbb{K} \qquad \Gamma \vdash z :: \mathfrak{s} \qquad \Gamma \vdash s :: \mathfrak{s} \qquad \mathfrak{s} \neq \mathbb{K}}{\Gamma \vdash \text{vec-elim}_{\mathfrak{s}}\ (\mathrm{vcons}\ a\ n\ v)\ P\ z\ s \equiv s\ a\ (\mathrm{glength}\ v)\ v\ (\text{vec-elim}_{\mathfrak{s}}\ v\ P\ z\ s)}$$

Fig. 12. Vectors

5.2.3 Discriminating erased natural numbers. This example shows the power of ghost types: even though they cannot be used for computation, one can still distinguish values, something which cannot be done in Prop. The key is to use Reveal to define a discrimination proposition discrP such that discrP (hide 0) is provable but discrP (gS n) is equivalent to $\bot$.

$$\begin{array}{lcl}\mathrm{discrP} & : & \mathrm{erased\ nat} \to \mathrm{Prop} \\ \mathrm{discrP} & := & \lambda n.\ \mathrm{Reveal}\ n\ (\lambda x.\ \text{nat-elim}\ x\ (\lambda\_.\ \mathrm{Prop})\ \top\ (\lambda\_.\ \bot))\end{array}$$

toRev $\star$ is then a proof[9] of discrP (hide 0) whereas fromRev turns discrP (gS n) into a proof of $\bot$. We can then use it to define the following discriminator:

$$\begin{array}{lcl}\mathrm{discr} & : & \forall (n : \mathrm{erased\ nat}).\ \mathrm{hide}\ 0 \approx \mathrm{gS}\ n \to \bot \\ \mathrm{discr} & := & \lambda n\ e.\ \mathrm{fromRev}(\mathrm{cast}\ e\ \mathrm{discrP}\ (\mathrm{toRev}\ \star))\end{array}$$

5.3 Vectors

We now extend GTT with vectors where the length index is an erased natural number. This time we give their syntax and typing rules in details in Figure 12. Indeed, this example is the only one where ghosts play a major role. This means that hide and even gS appear in the typing rules, but the biggest surprise is probably the appearance of glength in the computation rule for vec-elim.

The most natural rule would have been to use n from the left-hand side instead of relying on another function. In fact it seems necessary to ensure that both sides of the equation have the same type. First, note that we consider untyped conversion for a reason: both sides may have been annotated with casts that were necessary for typing, but those are removed at conversion time; it is thus not an expectation of our conversion rules. The second, more important reason is

[9]Assuming $\star : \top$, which we can encode as $\lambda x.x : \bot \to \bot$.

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅.
```

```
Inductive vec^P (A : ty) (AP : El A → SProp) :
 ∀ n (nP : ℕ^P n), vec• A → SProp :=
| vnil^P : vec^P A AP 0• 0^P vnil•
| vcons^P a (aP : AP a) n nP v :
   vec^P A AP n nP v →
   vec^P A AP (S• n) (S^P n nP) (vcons• a v).
```

Fig. 13. Erasure and parametricity inductive types for vectors

that such a rule would be able to extract the natural number stored in the vcons constructor, but this information is gone after erasure which would jeopardise revival as we shall see. Instead we reconstruct this information from the vector itself.

The glength function is defined using the eliminator and computes the *ghost* length by iterating gS as shown below. We could have lifted the regular length function to erased natural numbers but we believe the current presentation is less ad-hoc as it amounts to iterating the building blocks of the indices of vec. Furthermore, the rule is no longer so surprising if we simply see the last two arguments of s as recursive calls on v. How this generalises to other inductive types remains to be investigated.

$$\text{glength } v := \text{vec-elim } v\ (\lambda\_\ \_.\ \text{erased nat})\ (\text{hide } 0)\ (\lambda a\ m\ w\ r.\ \text{gS } r)$$

In other words, glength vnil ≡ hide 0 and glength (vcons a n v) ≡ gS (glength v).

5.3.1 Modelling vectors. To build the model, we proceed as we did for natural numbers by showing how one implements the various translations in Coq. The corresponding inductive types are found in Figure 13. This time it is more involved so we will go step by step.

First, erasure no longer consists in simply adding a special constructor for errors: the natural number argument has disappeared both from the type and from the vcons• constructor. The type vec• A is thus isomorphic to `list` (El A) which matches well the intuition that vectors are lists with an extra invariant constraining its length. This length reappears in the vec$^{\mathcal{P}}$ A AP n nP v predicate. Ignoring AP and nP for now, it says that v is an exception-free list of length n. The nP argument ensures that n itself is free of exceptions whereas AP is a predicate verified by all the elements of the list. Before we turn our attention towards the eliminators, remark that the parametricity translation of vectors is the first time we really make use of revival: without it there would be no way to recover the length to put inside the predicate: $[\![\text{vec } A\ n]\!]_{\mathcal{P}} := \text{vec}^{\mathcal{P}}\ [A]_{\varepsilon}\ [\![A]\!]_{\mathcal{P}}\ [\![n]\!]_{v}\ [\![n]\!]_{\mathcal{P}}$.

For erasure, we only need to consider the $\mathbb{T}$ case as usual. We require a constant as follows that we implement by using the eliminator of vec•.

```
vec_elim• :
 ∀ (A : ty) (P : vec• A → ty)
   (z : El (P vnil•)) (s : ∀ (a : El A) (v : vec• A), El (P v) → El (P (vcons• a v))) (v : vec• A),
   El (P v).
```

We verify the following two equations hold definitionally:

```
vec_elim• A P z s vnil• ≡ z
vec_elim• A P z s (vcons• a v) ≡ s a v (vec_elim• A P z s v)
```

For revival (and thus the $\mathbb{G}$ case), we can reuse vec_elim• but some care needs to be taken as the type we need to inhabit a slightly different type as s now mentions some natural number:

```
∀ (A : ty) (P : vec• A → ty)
 (z : El (P vnil•)) (s : ∀ (a : El A) (n : ℕ•) (v : vec• A), El (P v) → El (P (vcons• a v))) (v : vec• A),
 El (P v).
```

This might seem harmless as we can inhabit this type by feeding any natural number to s. Only, we need to interpret the computation rule for vec-elim. This is not just a technicality due to our definition of vec_elim•, the natural number is simply not stored in the vec• data type! This solves the mystery of why we need to use glength in the computation rule: to have a proper computation rule, we need to recompute the natural number index from v. Once we have made this observation it suffices to use a length• function in the model, corresponding to the erasure of glength in the s branch of vec_elim• and it verifies the expected equations.

For parametricity, we have three cases to consider and this time they are all different. Only the $\mathbb{G}$ case is not straightforward so we will focus on this one and refer the reader to the formalisation [TransVec.pm_vec_elim, TransVec.pm_vec_elim_Prop] for details about the others. The property we need to prove is as follows.

```
∀ A (AP : El A → SProp)
 (Pe : vec• A → ty)
 (PP : ∀ n nP (v : vec• A) (vP : vec^P A AP n nP v), El (Pe v) → SProp)
 (ze : El (Pe vnil•)) (zP : PP 0• 0^P vnil• vnil^P ze)
 (se : ∀ (a : El A) (n : err_ℕ) (v : vec• A), El (Pe v) → El (Pe (vcons• a v)))
 (sP :
  ∀ a aP n nP v vP (h : El (Pe v)) (hP : PP n nP v vP h),
   PP (S• n) (S^P n nP) (vcons• a v) (vcons^P a aP n nP v vP) (se a n v h)
 )
 n nP v vP,
 PP n nP v vP (vec_elim• A Pe ze (λ a v, se a (err_length v) v) v).
```

Here, a basic induction on vP is not enough (as it was up till now). In the recursive case there is indeed a mismatch between n and length• v. We solve this problem by remarking that they are propositionally equal[10] which we also prove by induction on vP.

```
Lemma err_length_eq : ∀ A AP n nP v (vP : vec^P A AP n nP v), ‖ length• v = n ‖.
```

This concludes our construction of the model. This translation echoes the alternative representation of vectors as a refinement type with a condition on the length:

$$\mathrm{vec}\ A\ n \approx \{l : \mathrm{list}\ A \mid \mathrm{length}\, l = n\}.$$

There is a trade-off between the two versions: the inductive one makes it easier in theory to produce correct-by-construction results, but relevant terms end up being polluted by proofs, which can be tedious to write and manage (for instance, they might get in the way of equality or computation). We argue that our setting allows one to get the best of both worlds, as we will illustrate in Section 5.3.2 and as this will become even clearer in Section 6.

5.3.2 Examples. We start by showing that even though the length index is erased, we can still take advantage of it to define the usual head and tail total functions. To do it, we have to rely on exfalso but also on the discriminator discr introduced in Section 5.2.

$$\begin{aligned} \mathrm{head} &: \mathrm{vec}\ A\ (\mathrm{gS}\ n) \to A \\ \mathrm{tail} &: \mathrm{vec}\ A\ (\mathrm{gS}\ n) \to \mathrm{vec}\ A\ n \end{aligned}$$

[10] We need a squash to go from Prop to SProp.

head is defined below:

$$\begin{array}{lcl} P & := & \lambda m\ w.\ \forall (k : \mathsf{erased\ nat}).\ m \approx \mathrm{gS}\ k \to A \\ z & := & \lambda k\ (h : \mathsf{hide}\ 0 \approx \mathrm{gS}\ k).\ \mathsf{exfalso}\ A\ (\mathrm{discr}\ k\ h) \\ s & := & \lambda a\ n\ v\ \_\ k\ h.\ a \\ \mathrm{head} & := & \lambda (v : \mathsf{vec}\ A\ (\mathrm{gS}\ n)).\ (\mathsf{vec\text{-}elim}\ v\ P\ z\ s)\ n\ (\mathsf{gh\text{-}refl}\ (\mathrm{gS}\ n)) \end{array}$$

tail has a similar definition but we also use a cast to rewrite in the length index of the vector.

$$\begin{array}{lcl} P & := & \lambda m\ w.\ \forall (k : \mathsf{erased\ nat}).\ m \approx \mathrm{gS}\ k \to \mathsf{vec}\ A\ k \\ z & := & \lambda k\ h.\ \mathsf{exfalso}\ (\mathsf{vec}\ A\ k)\ (\mathrm{discr}\ k\ h) \\ s & := & \lambda a\ n\ v\ \_\ k\ h.\ \mathsf{cast}\ e\ (\mathsf{vec}\ A)\ v \\ \mathrm{tail} & := & \lambda (v : \mathsf{vec}\ A\ (\mathrm{gS}\ n)).\ (\mathsf{vec\text{-}elim}\ v\ P\ z\ s)\ n\ (\mathsf{gh\text{-}refl}\ (\mathrm{gS}\ n)) \end{array}$$

where $e : n \approx k$ is obtained from $h : \mathrm{gS}\ n \approx \mathrm{gS}\, k$. The two functions erase as one would do in OCaml by raising an exception when the list is empty.

Finally, we bring closure to our initial example of reversal of vectors and show how we define it in GTT,[11] assuming $\oplus$ is lifting the usual addition on nat to erased nat.

$$\begin{array}{l} \mathrm{rev} : \forall\ n\ m.\ \mathsf{vec}\ A\ n \to \mathsf{vec}\ A\ m \to \mathsf{vec}\ A\ (n \oplus m) \\ \mathrm{rev}\ (\mathsf{hide}\ 0)\ m\ \mathsf{vnil}\ acc := \mathsf{cast}\ e_0\ (\mathsf{vec}\ A)\ acc \\ \mathrm{rev}\ (\mathrm{gS}\ k)\ m\ (\mathsf{vcons}\ a\ k\ w)\ acc := \mathsf{cast}\ e_1\ (\mathsf{vec}\ A)\ (\mathrm{rev}\ k\ (\mathrm{gS}\ m)\ w\ (\mathsf{vcons}\ a\ m\ acc)) \end{array}$$

where $e_0 : (\mathsf{hide}\ 0) \oplus m \approx m$ and $e_1 : k \oplus (\mathrm{gS}\ m) \approx (\mathrm{gS}\ k) \oplus m$ are standard, modulo reveal. In the next section we will show how the ideal version without casts can also be supported. However, we can already argue that the cast is no more than a crutch for the type checker and that it can safely be ignored for any subsequent proofs about rev. Equational reasoning is highly simplified when compared to the version with a transport over proof-relevant equality.

6 GHOST REFLECTION

As stated in the introduction, one of the main motivations for this work is to be able to obtain a nicer notion of extensional type theory using ghost types. We now introduce GRTT (ghost reflection type theory) as a variant of GTT presented in Section 2. We will show how GRTT is conservative over GTT, allowing us to lift the properties we established for GTT in Section 4 to GRTT. We can do it because GTT has been set up *just right* so that it can support extensionality. In a sense, this section is going to be a justification for the rules of GTT. GRTT also has the advantage of showing how one could GTT in practice by letting some oracle insert the casts for the user who doesn't have to worry about them.

6.1 Definition of GRTT [RTyping.grtyping]

GRTT is a variant of GTT where we remove cast from syntax as well as its typing rule, and where we change $\Gamma \vdash |A| \equiv |B|$ for $\Gamma \vdash A \equiv B$ in the conversion rule (both rules are highlighted in blue in Figure 2). Note that we reuse the conversion from GTT here. We annotate the turnstile of GRTT judgements with an x to differentiate them from GTT judgments.

Instead of casts, ghost equality is eliminated through a special *ghost equality reflection* rule that let us convert between types which have equal ghost values inside.

$$\frac{\Gamma \vdash_x e : u \approx_A v \qquad \Gamma \vdash_x P : A \to \mathrm{Sort}^{\mathfrak{s}} \qquad \Gamma \vdash_x t : P\ u \qquad \mathfrak{s} \neq \mathbb{K}}{\Gamma \vdash_x t : P\ v}$$

[11] The version using eliminators explicitly is given in Figure 14 in Appendix A.

This rule mimics the cast typing rule and has the same limitation that it cannot be used at sort Kind. This way, conservativity of GRTT over GTT essentially states that casts can indeed be safely ignored when computing. The reader may find it surprising that we limit reflection to applicative contexts instead of the usual generality of ETT; in other words we cannot use reflection of an equality that is only well formed under binders. The main reason is that it corresponds exactly to the GTT model we give. If we wanted to allow for reflection of ghost equalities under binders we would need to extend GTT with extra extensionality principles that we conjecture would in turn require at least function extensionality in the target of the parametricity translation. For now we chose to limit ourselves to a more agnostic notion of equality that we in any case believe is enough for most applications and leave the investigation of other approaches to future work.

GRTT can thus be seen as a restriction of the regular ETT where reflection happens only at certain positions such as the length index of vectors. Our running example of rev can be expressed in GRTT just as it was in Section 1.

6.2 Potential translations

We adapt—and mostly simplify—the proof of Oury [2005]; Winterhalter et al. [2019] to translate *derivations* in GRTT to derivations in GTT. The main idea being replacing the use of reflection with casts in the target. As Oury; Winterhalter et al. already discuss in their case, this means *a priori* that the same term (resp. context or type) can have several valid translations and typically vary in which casts appear in the term. Our own proof can become much simpler thanks to the fact that ghost casts are irrelevant for conversion. We can thus prove that all translations of the same term are definitionally equal.

We start making this fact precise by defining *potential translations* of a GRTT term t as GTT terms t' such that $|t'| =_\alpha t$. Thus, two potential translations of a same term are definitionally equal in GTT.

We define potential translations of judgments as follows. [Potential.tr_ctx, Potential.tr_ty]

(1) $\vdash \Gamma' \in [\![\vdash_x \Gamma]\!]_x$ when $\vdash \Gamma'$ is derivable and $|\Gamma'| =_\alpha \Gamma$ (pointwise);
(2) $\Gamma' \vdash t' : A' \in [\![\Gamma \vdash_x t : A]\!]_x$ when $\Gamma' \vdash t' : A'$ is derivable and $|t'| =_\alpha t$ and $|A'| =_\alpha A$.

Note that we do not need to provide a translation to conversion since it is purely syntactic and ignores casts, meaning we can use conversion derivation *as is*.

We will now define the two crucial lemmas for conducting the translation. They essentially state that the translation does not get in the way of typing.

Lemma 6.1 (Choice of type). [Potential.tr_choice]
If $\Gamma' \in [\![\vdash_x \Gamma]\!]_x$ *and* $\Gamma' \vdash t' : A' \in [\![\Gamma \vdash_x t : A]\!]_x$ *and* $\Gamma' \vdash A'' : \mathrm{Sort}_i^{\mathfrak{s}} \in [\![\Gamma \vdash_x A : \mathrm{Sort}_i^{\mathfrak{s}}]\!]_x$ *with* $\Gamma \vdash t :: \mathfrak{s}$ *then we also have* $\Gamma' \vdash t' : A'' \in [\![\Gamma \vdash_x t : A]\!]_x$.

Proof. We apply the conversion rule by remarking that $|A'| \equiv |A''|$ holds since $|A'| =_\alpha |A''|$. □

Lemma 6.2 (Preservation of type formers). [Potential.tr_sort_eq, Potential.tr_bot_eq]
If $\Gamma' \vdash t' : A' \in [\![\Gamma \vdash_x t : A]\!]_x$ *where* A *has a type former in its head then* A' *has the same head as* A.

Proof. Since $|A'| =_\alpha A$ we know that $A' =_\alpha \mathrm{cast}\ e_1\ P_1\ (\dots(\mathrm{cast}\ e_n\ P_n\ A'')\dots)$ for some A'' with the same head as A. By Lemma 2.6 we also know that $\Gamma' \vdash A' : \mathrm{Sort}$. Using Lemma 4.4 we thus get that $\Gamma' \vdash A' :: \mathbb{K}$. By inversion of typing on the cast rule, we would however get that this mode cannot be $\mathbb{K}$. Hence, $n = 0$ (no casts were applied) and $A' =_\alpha A''$. □

6.3 Eliminating ghost reflection

We now define a translation from GRTT derivations to GTT derivations by showing that whenever a judgment $\mathcal{J}$ is derivable in GRTT, then $[\![\mathcal{J}]\!]_x$ is inhabited, *i.e.* $\mathcal{J}$ has a translation in GTT. Furthermore, this translation is constructive meaning we could extract an algorithm from it.

Theorem 6.3 (Translation from GRTT to GTT). [ElimReflection.elim_reflection,elim_ctx]
(1) If $\vdash_x \Gamma$, *then there exists* $\vdash \Gamma' \in [\![\vdash_x \Gamma]\!]_x$.
(2) If $\Gamma \vdash_x t : A$, *then for all* $\vdash \Gamma' \in [\![\vdash_x \Gamma]\!]_x$, *there exists* $\Gamma' \vdash t' : A' \in [\![\Gamma \vdash_x t : A]\!]_x$.

Proof. We prove the assertions above by induction on the derivations and freely use Lemmas 6.1 and 6.2 to align the various translations. We show a few representative cases.

The most interesting case is probably the ghost reflection rule. By induction hypothesis we have $\Gamma' \vdash e' : u' \approx_{A'} v' \in [\![\Gamma \vdash_x e : u \approx_A v]\!]_x$ and $\Gamma' \vdash P' : A' \to s \in [\![\Gamma \vdash_x P : A \to \mathrm{Sort}^{\mathfrak{s}}]\!]_x$ and $\Gamma' \vdash t' : P'\ u' \in [\![\Gamma \vdash_x t : P\ u]\!]_x$ with $\mathfrak{s} \neq \mathbb{K}$. We can then easily conclude that $\Gamma' \vdash \mathrm{cast}\ e'\ P'\ t' : P'\ v' \in [\![\Gamma \vdash_x t : P\ v]\!]_x$. □

6.4 Conservativity and meta-theoretical results

We now establish some meta-theoretical consequences of the GRTT to GTT translation. We show that GRTT is conservative over GTT, meaning that GRTT captures *exactly* what can be proven in GTT, they have the same logical power.

Theorem 6.4 (Conservativity). [ElimReflection.conservativity] *Whenever we have valid type in GTT* $\vdash A : \mathrm{Sort}^{\mathfrak{s}}$ *such that its cast-free version is inhabited in GRTT* $\vdash_x t : |A|$ *with* $t :: \mathfrak{s}$ *then it is already inhabited in GTT,* i.e. *there is some* t' *such that* $|t'| =_\alpha t$ *and* $\vdash t' : A$.

Proof. Assume $\vdash A : s$ and $\vdash_x t : |A|$, then by applying Theorem 6.3 we have some $\vdash t' : A' \in [\![\vdash_x t : |A|]\!]_x$. Besides, we know that $\vdash A : \mathrm{Sort}^{\mathfrak{s}} \in [\![\vdash_x |A| : \mathrm{Sort}^{\mathfrak{s}}]\!]_x$ so by Lemma 6.1 we get $\vdash t' : A \in [\![\vdash_x t : |A|]\!]_x$. □

There is one caveat: the term t in GRTT needs to be in the right mode, something which we inherit from the conversion rule. As stated earlier we conjecture that this requirement is redundant but we leave it to future work.

Theorem 6.5 (Consistency of GRTT). [ElimReflection.consistency] *GRTT is consistent.*

Proof. Assuming $\vdash_x t : \bot$, then by Theorem 6.3 and Lemma 6.1 we get some $\vdash t' : \bot$ which would disprove consistency of GTT (Theorem 4.1). □

Properties like discrimination of type formers are inherited trivially from GTT since GRTT uses the same conversion. Of course, this works mainly because we did not consider reflection as part of definitional equality but as a rule in itself. We argue that this is enough and that GRTT serves rather as a way to demonstrate that GTT could be used in practice. Conservativity ensures that there is no risk to forgetting about casts (or even hiding them to a user), and furthermore the resulting proof term is essentially the same (up to casts). Amongst prospects, this means that equations such as associativity of vector concatenation which features two terms of different types could be rewritten with, without worrying about casts on either side.

7 RELATED WORK

Ghost types, erasure and shape irrelevance with dependent types. Shape irrelevance was introduced by Abel et al. [2017b] for sized types to be able to say that the size argument in a type does not affect its shape (*e.g.* one cannot match on a size to build a type that would be either nat or bool).

Shape irrelevance was later refined by Nuyts and Devriese [2018] in the more general framework of modalities, from which one may extract a notion of heterogeneous equality. It shares similarities with the *erased modality* in that both are used to represent data that is irrelevant for computation but still holds meaning as a specification. Both are implemented in Agda, and in fact Agda's implementation of the erased modality is a special case of Quantitative type theory (QTT) [Atkey 2018] which is also implemented in Idris 2 [Brady 2021]. It tracks variable usage in judgements, separating occurrences in the term from the ones in the type. Compared to GTT, all of these cases are not type-based—although we believe that our proof should be adaptable to modalities—and to the best of our knowledge do not consider internal principles like we did. The ability to distinguish relevance in the term and in the type however is convenient to be able to consider *e.g.* the length arguments in vector *terms* to be irrelevant. For us, it is only achieved in GRTT for *propositionally equal* sizes, in other words for terms of the same type (which should not be a restriction in practice).

Ghost types in F* [Swamy et al. 2016] served as an inspiration for our Ghost universe, but there remains several differences. Indeed, in F*, *ghost* is an effect rather than a universe and it comes with implicit lifts to it so that ghost computations can be eliminated into any *non-informative* type. In contrast to GTT, it considers the universe to be non-informative, meaning that ghost computations can be examined not only to produce propositions but also any type. This is possible because the erasure they consider gets rid of all types, unlike what we do. However there is also an important similarity in how both F* and GRTT handle equality reflection: they both cannot freely go under binders.

Brady et al. [2003] show that inductive types need not store their indices, a case that we also make for vectors in the introduction. They erase arguments to constructors but do not seem to allow to carry this information with typing while we can expose in the type of rev that the natural numbers it takes as arguments are erased. Their work focus mainly on memory optimisation and we believe is essentially orthogonal to what we propose.

Miquel [2001] introduced the Implicit Calculus of Constructions (ICC) which features a special dependent product type whose arguments are not materialised and which behaves as an intersection type. There is no need to erase these arguments because they are not there in the first place. Like GRTT, ICC has undecidable type checking and Barras and Bernardo [2008] thus introduce a variant, called ICC*, with more annotations to recover decidability of checking. We conjecture it is also the case for GTT and in that sense the two systems are similar. Another similarity is that, in ICC*, conversion also compares so-called extracted terms. Both ICC and ICC* cannot distinguish constructors because inductive types are only impredicative encodings. While they can encode vectors this way, the head and tail function are outside of their scope, while it is an important part of ours.

Cedille [Stump 2017; Stump and Jenkins 2018] proposes dependent intersection types relying on a primitive notion of erasure for terms that discards types. Like ICC, it offers a notion of erasure that is different than that of GTT, with different kinds of applications, perhaps more prominently for impredicative encodings. We instead focused on reasoning principles about erased data, which is what casts (or reflection) give us, in addition to free theorems obtained through parametricity.

Ghost code in verification. Filliâtre et al. [2016] introduce ghost code to the Why3 tool for program verification [Filliâtre and Paskevich 2013]. What we propose is close to a dependent version of their work, which lets us additionally *internalise* the fact that some data is ghost. Like in their case, we can erase ghost code to benefit from better extraction and to ensure that ghost code does not *interfere* with computations. Their type system features other similarities with our work in that variables are syntactically annotated with their ghost status and like us they state that those annotations can be inferred by the typing rules and need not be supplied by the user. One main

difference is that Why3 supports effects which we do not, as Coq or Agda do not. In that direction connecting our work with F* would allow for a better comparison and we leave it for future work.

8 CONCLUSION AND FUTURE WORK

The various translations we presented in the paper provide a model both for GTT and GRTT that we can view as two variants of the same system. GTT is better suited to serve as a kernel to a proof assistant while GRTT might be better as a practical system, despite the fact that type checking is obviously undecidable (as it is for type systems with equality reflection). Indeed, there are many ways to make this system practical, be it with definitional equality handlers *à la* Andromeda 1 [Bauer et al. 2016], or by delegating such proof obligations to an SMT solver *à la* F* [Swamy et al. 2016] or even via the use of (user-defined) rewrite rules. A general take-away from this is that automation confined to erased positions remains powerful and useful but crucially does not get in the way of equational reasoning or of extracted programs. The latter point is reinforced by the erasure translation we propose that gets rid of both proofs and ghost values and could as such serve as a first step before extraction.

We thus believe that this and future work in that direction could help justify and inform the design and meta-theory of tools like F* or expand the capabilities of other proof assistants such as Coq or Agda. There are still several challenges to tackle before implementation. Perhaps the most prominent and achievable one would be to extend G(R)TT to support general inductive types. As these proofs are usually tricky, we would highly benefit from a formalisation such as that of MetaCoq [Sozeau et al. 2020a,b] which deals with most features of Coq. A general treatment of indices would be of particular interest, especially given the fact that some dependencies are erased even after revival. We also plan to investigate how to accommodate for large elimination of at least a subset of inductive types. We could then hope to tackle other properties of GTT than the ones we establish with the current model. One of them is termination. We conjecture that GTT—and even GRTT (as opposed to ETT in general)—have a proper notion of computation and that they are in fact strongly normalising. We could prove this fact from our model by showing that it is in fact a simulation for reduction. In case this would prove insufficient to prove decidability of type checking—this time only for GTT—we could have a look at other methods such as logical relations [Abel et al. 2017a]. While decidability of type checking is out of reach for GRTT, we believe that what is decidable is to determine which proof obligations are required to make a term type-check. In other words we could decide where casts are necessary and type former discrimination would ensure that the obligations make sense and are typically about indices of inductive types, or constraints of refinement types.

Another point to sort out before implementation is the way we deal with binders for ghost reflection. As stated in the paper, the ghost reflection rule does not give rise directly to an implementation. Indeed, it should rather be merged with conversion to properly work. This means for one keeping track of the context, but also making a distinction between which variables were introduced by congruence so that they are not available in the context of the proof obligation. Alternatively, we could allow going under binder at the cost of—presumably—function extensionality.

Finally, it would be interesting to extend GTT to support accessibility predicates as ghost types. Indeed, in Coq they are currently considered as inductive types in Prop, but not in SProp. In other words, it does not enjoy definitional proof irrelevance, but it is still erased during extraction, which makes it a perfect candidate for our Ghost universe. Such an extension would require more investigation as the target cannot readily accommodate for potentially non-terminating definitions that we would only show terminating after the fact (typically through the parametricity translation).

REFERENCES

Andreas Abel and Thierry Coquand. 2020. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. *Logical Methods in Computer Science* 16 (2020).

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017a. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017b. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–30.

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending homotopy type theory with strict equality. *arXiv preprint arXiv:1604.03799* (2016).

Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 56–65.

Bruno Barras and Bruno Bernardo. 2008. The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computational Structures: 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 11*. Springer, 365–379.

Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Chris Stone. 2016. The 'Andromeda' prover. http://www.andromeda-prover.org/

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Certified Programs and Proofs – CPP 2017*. 182–194.

Edwin Brady. 2021. Idris 2: Quantitative type theory in practice. *arXiv preprint arXiv:2104.00480* (2021).

Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive families need not store their indices. In *International Workshop on Types for Proofs and Programs*. Springer, 115–129.

Jesper Cockx and Andreas Abel. 2016. Sprinkles of extensionality for your vanilla type theory. In *Types for Proofs and Programs (TYPES)*.

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages* (2021). https://doi.org/10.1145/3434341

Robert L. Constable and Joseph L. Bates. 2014. The NuPrl system, PRL project. http://www.nuprl.org/

Coq development team. 2023. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.17.

Adrian Dapprich and Andrej Dudenhefner. 2021. Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml. *Bachelor thesis, Saarland University* (2021).

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The spirit of ghost code. *Formal Methods in System Design* 48 (2016), 152–174.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 125–128.

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

Martin Hofmann. 1995. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 153–164.

Chantal Keller and Marc Lasson. 2012. Parametricity in an impredicative sort. *arXiv preprint arXiv:1209.6336* (2012).

Yann Leray. 2022. *Formalisation et implémentation des propositions strictes dans MetaCoq*. Technical Report. Inria Rennes - Bretagne Atlantique ; LS2N-Nantes Université. 17 pages. https://inria.hal.science/hal-04433492

Alexandre Miquel. 2001. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 344–359.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

Andreas Nuyts and Dominique Devriese. 2018. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 779–788.

Nicolas Oury. 2005. Extensionality in the calculus of constructions. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 278–293.

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming (LNCS, Vol. 10801)*. Springer, Thessaloniki, Greece, 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230.

Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–27.

Loïc Pujet and Nicolas Tabareau. 2023. Impredicative Observational Equality. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 2171–2196.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020a. The MetaCoq Project. *Journal of Automated Reasoning* (Feb. 2020). https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020b. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* (Jan. 2020), 1–28. https://doi.org/10.1145/3371076

Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 166–180.

Pierre-Yves Strub. 2010. Coq modulo theory. In *Computer Science Logic: 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings 24*. Springer, 529–543.

Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14.

Aaron Stump and Christopher Jenkins. 2018. Syntax and semantics of Cedille. *arXiv preprint arXiv:1806.04709* (2018).

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Théo Winterhalter. 2020. *Formalisation and meta-theory of type theory*. Ph. D. Dissertation. Université de Nantes.

Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. 2019. Eliminating Reflection from Type Theory. In *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Lisbonne, Portugal, 91–103. https://doi.org/10.1145/3293880.3294095

A DEFINITION OF REVERSAL OF VECTORS IN GTT

We provide here the definition of rev (Figure 14) using eliminators. We still omit the proofs e_0 and e_1.

$$
\begin{array}{lcl}
\mathrm{rev} & : & \forall\ n\ m.\ \mathrm{vec}\ A\ n \to \mathrm{vec}\ A\ m \to \mathrm{vec}\ A\ (n \oplus m) \\
\mathrm{rev} & := & \lambda n\ m\ v.\ (\text{vec-elim}\ v\ (\lambda\ k\ w.\ \forall m.\ \mathrm{vec}\ A\ m \to \mathrm{vec}\ A\ (k \oplus m))\ b_{\mathrm{vnil}}\ b_{\mathrm{vcons}})\ m \\
b_{\mathrm{vnil}} & := & \lambda\ m\ acc.\ \mathrm{cast}\ e_0\ (\mathrm{vec}\ A)\ acc \\
b_{\mathrm{vcons}} & := & \lambda\ a\ k\ w\ rec\ m\ acc.\ \mathrm{cast}\ e_1\ (\mathrm{vec}\ A)\ (rec\ (\mathrm{gS}\ m)\ (\mathrm{vcons}\ a\ m\ acc)) \\
e_0 & : & (\mathrm{hide}\ 0) \oplus m \approx m \\
e_1 & : & k \oplus (\mathrm{gS}\ m) \approx (\mathrm{gS}\ k) \oplus m
\end{array}
$$

Fig. 14. Definition of rev from Section 5.3.2.