Staged Compilation With Two-Level Type Theory

ANDRÁS KOVÁCS, Eötvös Loránd University, Hungary

The aim of staged compilation is to enable metaprogramming in a way such that we have guarantees about the well-formedness of code output, and we can also mix together object-level and meta-level code in a concise and convenient manner. In this work, we observe that two-level type theory (2LTT), a system originally devised for the purpose of developing synthetic homotopy theory, also serves as a system for staged compilation with dependent types. 2LTT has numerous good properties for this use case: it has a concise specification, well-developed algebraic and categorical model theory, and it supports a wide range of language features both at the object and the meta level. First, we give an overview of 2LTT's features and applications in staging. Then, we present a staging algorithm and provide a proof of correctness. Our algorithm is "staging-by-evaluation", analogously to the technique of normalization-by-evaluation, in that staging is given by the evaluation of 2LTT syntax in a semantic domain. The staging algorithm together with its correctness proof constitutes a proof of strong conservativity of 2LLT over the object theory. To our knowledge, this is the first system for staged compilation which supports full dependent types and unrestricted staging for types.

Additional Key Words and Phrases: type theory, two-level type theory, staged compilation

ACM Reference Format:
András Kovács. 2018. Staged Compilation With Two-Level Type Theory. *J. ACM* 37, 4, Article 111 (August 2018), 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The purpose of staged compilation is to write code-generating programs in a safe, ergonomic and expressive way. It is always possible to do ad-hoc code generation, by simply manipulating strings or syntax trees in a sufficiently expressive programming language. However, these approaches tend to suffer from verbosity, non-reusability and lack of safety. In staged compilation, there are certain *restrictions* on which metaprograms are expressible. Usually, staged systems enforce typing disciple, prohibit arbitrary manipulation of object-level scopes, and often they also prohibit accessing the internal structure of object expressions. On the other hand, we get *guarantees* about the well-scoping or well-typing of the code output, and we are also able to use concise syntax for embedding object-level code.

Two-level type theory, or 2LTT in short, was described by Annekov, Capriotti, Kraus and Sattler [1], building on ideas from Vladimir Voevodsky [4]. The motivation was to allow convenient metatheoretical reasoning about a certain mathematical language (homotopy type theory), and to enable concise and modular ways to extend the language with axioms.

It turns out that metamathematical convenience closely corresponds to to metaprogramming convenience: 2LTT can be directly and effectively employed in staged compilation. Moreover, semantic ideas underlying 2LTT are also directly applicable to the theory of staging.

Author's address: András Kovács, kovacsandras@inf.elte.hu, Eötvös Loránd University, Hungary, Budapest.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

1.1 Contributions

- In **??** we present an informal syntax of two-level type theory, a dependent type theory with staging features. We look at basic use-cases involving inlining control, partial evaluation and fusion optimizations. We also describe feature variations, enabling applications in monomorphization and memory layout control.
- In **??**, following [1], we present a formal syntax of 2LTT and the object theory (the target theory of code generation). We recall the standard presheaf model of 2LTT, which lies over the syntactic category of the object theory. We show that the evaluation of 2LTT syntax in the presheaf model yields a staging algorithm.
- In **??** we show correctness of staging, consisting of
  - *Stability:* staging the output of staging has no action.
  - *Soundness:* the output of staging is convertible to the input.
  - *Completeness:* convertible programs produce convertible staging outputs.

  Staging together with its correctness can be viewed as a *strong conservativity* theorem of 2LTT over the object theory. This means that the possible object-level constructions in 2LTT are in bijection with the constructions in the object theory, and staging witnesses that meta-level constructions can be always computed away. This improves on the weak notion of conservativity shown in [2] and [1].
- To our knowledge, this is the first description of a language which supports staging in the presence of full-blown dependent types, with universes and large elimination. Moreover, we allow unrestricted staging for types, so that types can be computed by metaprograms at compile time.

2 A TOUR OF TWO-LEVEL TYPE THEORY

In this section, we provide a short overview of 2LTT and its potential applications in staging. We work in the informal syntax of a dependently typed language which resembles Agda [**?**]. We focus on examples and informal explanations here; the formal details will be presented in Section [**?**].

Notation 1. We use the following notations throughout the paper. $(x : A) \to B$ denotes a dependent function type, where x may occur in B. We use $\lambda\, x.\, t$ for abstraction. A Σ-type is written as $(x : A) \times B$, with pairing as $(t,\, u)$, projections as fst and snd, and we may use pattern matching notation on pairs, e.g. as in $\lambda\,(x,\, y).\, t$. The unit type is $\top$ with element tt. We will also use Agda-style notation for implicit arguments, where $t : \{x : A\} \to B$ implies that the first argument to t is inferred by default, and we can override this by writing a $t\{u\}$ application. We may also implicitly quantify over arguments (in the style of Idris and Haskell), for example when declaring $id : A \to A$ with the assumption that A is universally quantified.

2.1 Rules of 2LTT

Universes. We have universes $\mathsf{U}_{i,j}$, where $i \in \{0, 1\}$, and $j \in \mathbb{N}$. The i index denotes stages, where 0 is the runtime (object-level) stage, and 1 is the compile time (meta-level) stage. The j index denotes universe sizes in the usual sense of type theory. We assume Russell-style universes, with $\mathsf{U}_{i,j} : \mathsf{U}_{i,j+1}$. However, for the sake of brevity we will usually omit the j indices in this section, since sizing is orthogonal to our use-cases and examples.

- U_0 can be viewed as the *universe of object-level or runtime types.* Each closed type $A : \mathsf{U}_0$ can be staged to an actual type in the object language (the language of the staging output).
- U_1 can be viewed as the *universe of meta-level or static types.* If we have $A : \mathsf{U}_1$, then A is guaranteed to be only present at compile time, and will be staged away. Elements $a : A$ are likewise computed away.

Type formers. U_0 and U_1 may be closed under arbitrary type formers, such as functions, Σ-types, identity types or inductive types in general. However, all constructors and eliminators in type formers must stay at the same stage. For example:

- Function domain and codomain types must be at the same stage.
- If we have $\mathsf{Nat}_0 : \mathsf{U}_0$ for the runtime type of natural numbers, we can only map from it to a type in U_0 by recursion or induction.

It is not required that we have the *same* type formers at both stages. We will discuss setups with different languages at different stages in Seection **??**.

Moving between stages. At this point, our system is rather limited, since there is no interaction between the stages. We enable such interaction through the following operations.

- *Lifting:* for $A : \mathsf{U}_0$, we have $\Uparrow A : \mathsf{U}_1$. From the staging point of view, $\Uparrow A$ is the type of metaprograms which compute runtime expressions of type A.
- *Quoting:* for $A : \mathsf{U}_0$ and $t : A$, we have $\langle t \rangle : \Uparrow A$. A quoted term $\langle t \rangle$ represents the metaprogram which immediately yields t.
- *Splicing:* for $A : \mathsf{U}_0$ and $t : \Uparrow A$, we have $\sim t : A$. During staging, the metaprogram in the splice is executed, and the resulting expression is inserted into the output.

  Notation 2. Splicing binds stronger than any operation, including function application. For instance, $\sim f\,x$ is parsed as $(\sim f)\,x$.

- Quoting and splicing are definitional inverses, i.e. we have $\sim\langle t \rangle = t$ and $\langle \sim t \rangle = t$ as definitional equalities.

Note that none of these three operations can be expressed as functions, since function types cannot cross between stages.

Informally, if we have a closed program $t : A$ with $A : \mathsf{U}_0$, *staging* means computing all metaprograms and recursively replacing all splices in t and A with the resulting runtime expressions. The rules of 2LTT ensure that this is possible, and we always get a splice-free object program after staging.

Remark. Why do we use the index 0 for the runtime stage? The reason is that it is not difficult to generalize 2LTT to multi-level type theory, by allowing to lift types from U_i to U_{i+1}. In the semantics, this can be modeled by having a 2LTT whose object theory is once again a 2LTT, and doing this in an iterated fashion. But there must be necessarily a bottom-most object theory; hence our stage indexing scheme. For now though, we leave the multi-level generalization to future work.

Notation 3. We may disambiguate type formers at different stages by using 0 or 1 subscripts. For example, $\mathsf{Nat}_1 : \mathsf{U}_1$ is distinguished from $\mathsf{Nat}_0 : \mathsf{U}_0$, and likewise we may write $\mathsf{zero}_0 : \mathsf{Nat}_0$ and so on. For function and Σ types, the stage is usually easy to infer, so we do not annotate them. For example, the type $\mathsf{Nat}_0 \to \mathsf{Nat}_0$ must be at the runtime stage, since the domain and codomain types are at that stage, and we know that the function type former stays within a single stage. We may also omit stage annotations from λ and pairing.

2.2 Staged Programming in 2LTT

In 2LTT, we may have several different polymorphic identity functions. First, we consider the usual identity functions at each stage:

$$
\begin{aligned}
&id_0 : (A : \mathsf{U}_0) \to A \to A &\qquad& id_1 : (A : \mathsf{U}_1) \to A \to A \\
&id_0 := \lambda\,A\,x.x && id_1 := \lambda\,A\,x.x
\end{aligned}
$$

An id_0 application will simply appear in staging output as it is. In contrast, id_1 can be used as a compile-time evaluated function, because the staging operations allow us to freely apply id_1 to runtime arguments. For example, $id_1\,(\Uparrow \mathsf{Bool}_0)\,\langle \mathsf{true}_0\rangle$ has type $\Uparrow\mathsf{Bool}_0$, therefore $\sim\!(id_1\,(\Uparrow \mathsf{Bool}_0)\,\langle \mathsf{true}_0\rangle)$ has type Bool_0. We can stage this expression as follows:

$$\sim\!(id_1\,(\Uparrow \mathsf{Bool}_0)\,\langle \mathsf{true}_0\rangle) = \sim\!\langle \mathsf{true}_0\rangle = \mathsf{true}_0$$

There is another identity function, which computes at compile time, but which can be only used on runtime arguments:

$$\begin{aligned} &id_{\Uparrow} : (A : \Uparrow\mathsf{U}_0) \to \Uparrow \sim\!A \to \Uparrow \sim\!A \\ &id_{\Uparrow} := \lambda\, A\, x.x \end{aligned}$$

Note that since $A : \Uparrow\mathsf{U}_0$, we have $\sim\!A : \mathsf{U}_0$, hence $\Uparrow \sim\!A : \mathsf{U}_1$. Also, $\Uparrow\mathsf{U}_0 : \mathsf{U}_1$, so all function domain and codomain types in the type of $id_{\Uparrow}$ are at the same stage. Now, we may write $\sim\!(id_{\Uparrow}\,\langle\mathsf{Bool}_0\rangle\,\langle\mathsf{true}_0\rangle)$ for a term which is staged to true_0. In this specific case $id_{\Uparrow}$ has no practical advantage over id_1, but in some cases we really have to quantify over $\Uparrow\mathsf{U}_0$. This brings us to the next example.

Assume $\mathsf{List}_0 : \mathsf{U}_0 \to \mathsf{U}_0$ with $\mathsf{nil}_0 : (A : \mathsf{U}_0) \to \mathsf{List}_0\,A$, $\mathsf{cons}_0 : (A : \mathsf{U}_0) \to A \to \mathsf{List}_0\,A$ and $\mathsf{foldr}_0 : (A\,B : \mathsf{U}_0) \to (A \to B \to B) \to B \to \mathsf{List}_0\,A \to B$. We define a map function which "inlines" its function argument:

$$\begin{aligned} &map : (A\,B : \Uparrow\mathsf{U}_0) \to (\Uparrow \sim\!A \to \Uparrow \sim\!B) \to \Uparrow(\mathsf{List}_0 \sim\!A) \to \Uparrow(\mathsf{List}_0 \sim\!B) \\ &map := \lambda\, A\, B\, f\, as.\, \langle \mathsf{foldr}_0 \sim\!A \sim\!B\,(\lambda\, a\, bs.\, \mathsf{cons}_0 \sim\!B \sim\!(f\,\langle a\rangle)\, bs)\,(\mathsf{nil}_0 \sim\!B) \sim\!as\rangle \end{aligned}$$

This map function can be defined with quantification over $\Uparrow\mathsf{U}_0$ but not over U_1, because List_0 expects type parameters in U_0, and there is no generic way to convert from U_1 to U_0. Now, assuming $- +_0 - : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0$ and $ns : \mathsf{List}_0\,\mathsf{Nat}_0$, we have the following staging behavior:

$$\begin{aligned} &\sim\!(map\,\langle\mathsf{Nat}_0\rangle\,\langle\mathsf{Nat}_0\rangle\,(\lambda\, n.\,\langle \sim\!n +_0 10\rangle)\,\langle ns\rangle) \\ = \;&\sim\!\langle \mathsf{foldr}_0 \sim\!\langle\mathsf{Nat}_0\rangle \sim\!\langle\mathsf{Nat}_0\rangle\,(\lambda\, a\, bs.\, \mathsf{cons}_0 \sim\!B \sim\!\langle \sim\!\langle a\rangle +_0 10\rangle\, bs)\,(\mathsf{nil}_0 \sim\!\langle\mathsf{Nat}_0\rangle) \sim\!\langle ns\rangle\rangle \\ = \;&\mathsf{foldr}_0\,\mathsf{Nat}_0\,\mathsf{Nat}_0\,\,(\lambda\, a\, bs.\, a +_0 10)\,(\mathsf{nil}_0\,\mathsf{Nat}_0)\, ns \end{aligned}$$

By using meta-level functions and lifted types, we already have control over inlining. However, if we want to do more complicated meta-level computation, it is convenient to use recursion or induction on meta-level type formers. A classic example in staged compilation is the power function for natural numbers, which evaluates the exponent at compile time. We assume the iterator function $\mathsf{iter}_1 : \{A : \mathsf{U}_1\} \to \mathsf{Nat}_1 \to (A \to A) \to A \to A$, and runtime multiplication as $- *_0 -$.

$$\begin{aligned} &exp : \mathsf{Nat}_1 \to \Uparrow\mathsf{Nat}_0 \to \Uparrow\mathsf{Nat}_0 \\ &exp := \lambda\, x\, y.\, \mathsf{iter}_1\, x\,(\lambda\, n.\,\langle \sim\!y *_0 \sim\!n\rangle)\,\langle 1\rangle \end{aligned}$$

Now, $\sim\!(exp\,3\,\langle n\rangle)$ stages to $n *_0 n *_0 n *_0 1$ by the computation rules of iter_1 and the staging operations.

We can also stage *types*. Below, we use iteration to compute the type of vectors with static length, as a nested pair type.

$$\begin{aligned} &Vec : \mathsf{Nat}_1 \to \Uparrow\mathsf{U}_0 \to \Uparrow\mathsf{U}_0 \\ &Vec := \lambda\, n\, A.\, \mathsf{iter}_1\, n\,(\lambda\, B.\,\langle \sim\!A \times \sim\!B\rangle)\,\langle \top_0\rangle \end{aligned}$$

With this definition, $\sim\!(Vec\,3\,\langle\mathsf{Nat}_0\rangle)$ stages to $\mathsf{Nat}_0 \times (\mathsf{Nat}_0 \times (\mathsf{Nat}_0 \times \top_0))$. Now, we can use *induction* on Nat_1 to implement a map function. For readability, we use an Agda-style pattern matching definition below (instead of the elimination principle).

$$\begin{aligned}
&\mathit{map} : (n : \mathsf{Nat}_1) \to (\Uparrow{\sim}A \to \Uparrow{\sim}B) \to \Uparrow(\mathit{Vec}\,n\,A) \to \Uparrow(\mathit{Vec}\,n\,B)\\
&\mathit{map}\,\mathsf{zero}_1 \quad f\,\mathit{as} := \langle \mathsf{tt}_0 \rangle\\
&\mathit{map}\,(\mathsf{suc}_1\,n)\,f\,\mathit{as} := \langle ({\sim}(f\,\langle \mathsf{fst}_0\,{\sim}\mathit{as}\rangle),\ \mathit{map}\,n\,f\,\langle \mathsf{snd}_0\,{\sim}\mathit{as}\rangle)\rangle
\end{aligned}$$

This definition inlines the mapping function for each projected element of the vector. For instance, staging ${\sim}(\mathit{map}\,2\,(\lambda\,n.\,\langle{\sim}n +_0 10\rangle)\,\langle \mathit{ns}\rangle)$ yields $(\mathsf{fst}_0\,\mathit{ns} +_0 10,\ (\mathsf{fst}_0(\mathsf{snd}_0\,\mathit{ns}) +_0 10,\ \mathsf{tt}_0))$. Sometimes, we do not want to duplicate the code of the mapping function. In such cases, we can use *let-insertion* [**?**], a standard technique in staged compilation. If we bind a runtime expression to a runtime variable, and only use that variable in subsequent staging, only the variable itself can be duplicated. One solution is to do an ad-hoc let-insertion:

$$\begin{aligned}
&\mathsf{let}_0\,f := \lambda\,n.\,n +_0 10\ \mathsf{in}\ {\sim}(\mathit{map}\,2\,(\lambda\,n.\,\langle f\,{\sim}n\rangle)\,\langle \mathit{ns}\rangle)\\
=\ &\mathsf{let}_0\,f := \lambda\,n.\,n +_0 10\ \mathsf{in}\ (f\,(\mathsf{fst}_0\,\mathit{ns}),\ (f\,(\mathsf{fst}_0(\mathsf{snd}_0\,\mathit{ns})),\ \mathsf{tt}_0))
\end{aligned}$$

Alternatively, we can define *map* so that it performs let-insertion, and we can switch between the two versions as needed.

More generally, we are free to use dependent types at the meta-level, so we can reproduce more complicated staging examples. Any well-typed interpreter can be rephrased as a *partial evaluator*, as long as we have sufficient type formers. For instance, we may write a partial evaluator for a simply typed lambda calculus. We sketch the implementation in the following. First, we inductively define contexts, types and terms:

$$\mathsf{Ty} : \mathsf{U}_1 \qquad \mathsf{Con} : \mathsf{U}_1 \qquad \mathsf{Tm} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{U}_1$$

Then we define the interpretation functions:

$$\begin{aligned}
&\mathsf{EvalTy} \ \ : \mathsf{Ty} \to \Uparrow\mathsf{U}_0\\
&\mathsf{EvalCon} : \mathsf{Con} \to \mathsf{U}_1\\
&\mathsf{EvalTm} \ \ : \mathsf{Tm}\,\Gamma\,A \to \mathsf{EvalCon}\,\Gamma \to \Uparrow{\sim}(\mathsf{EvalTy}\,A)
\end{aligned}$$

Types are necessarily computed to runtime types, e.g. an embedded representation of the natural number type is evaluated to $\langle \mathsf{Nat}_0 \rangle$. Contexts are computed as follows:

$$\begin{aligned}
&\mathsf{EvalCon}\ \mathsf{empty} && := \top_1\\
&\mathsf{EvalCon}\ (\mathsf{extend}\,\Gamma\,A) && := \mathsf{EvalCon}\,\Gamma \times (\Uparrow{\sim}(\mathsf{EvalTy}\,A))
\end{aligned}$$

This is an example for the usage of *partially static data*[**?**]: semantic contexts are *static* lists storing *runtime* expressions. This allows us to completely eliminate environment lookups in the staging output: an embedded lambda expression is staged to the corresponding lambda expression in the runtime language. This is similar to the partial evaluator presented in Idris 1 [**?**]. However, in contrast to 2LTT, Idris 1 does not provide a formal guarantee that partial evaluation does not get stuck.

2.3 Properties of Lifting & Binding Time Improvements

We describe more generally the action of $\Uparrow$ on type formers. First, $\Uparrow$ preserves negative type formers up to definitional isomorphism [1]:

$$\begin{aligned}
\Uparrow((x : A) \to B\,x) &\simeq ((x : \Uparrow A) \to \Uparrow(B\,{\sim}x))\\
\Uparrow((x : A) \times B) &\simeq ((x : \Uparrow A) \times \Uparrow(B\,{\sim}x))\\
\Uparrow\top_0 &\simeq \top_1
\end{aligned}$$

For function types, the preservation maps are the following:

$$\begin{aligned}
&\mathit{pres}_{\to} : {\Uparrow}((x : A) \to B\,x) \to ((x : {\Uparrow}A) \to {\Uparrow}(B\,{\sim}x))\\
&\mathit{pres}_{\to}\,f := \lambda\,x.\ \langle f\,{\sim}x\rangle\\
&\mathit{pres}_{\to}^{-1}\,f := \langle \lambda\,x.\ {\sim}(f\,\langle x\rangle)\rangle
\end{aligned}$$

With this, we have that $\mathit{pres}_{\to}\,(\mathit{pres}_{\to}^{-1}\,f)$ is definitionally equal to f, and also the other way around. Preservation maps for Σ and $\top$ work analogously.

By rewriting a 2LTT program left-to-right along preservation maps, we perform what is termed *binding time improvement* in the partial evaluation literature [?]. Note that the output of $\mathit{pres}_{\to}$ uses a meta-level λ, while going the other way we introduce a runtime binder. Meta-level function and Σ types support more computation during staging, so in many cases it is beneficial to use the improved forms. In some cases though we may want to use unimproved forms, to limit the size of generated code. This is similar to what we have seen with let-insertion. For a minimal example, consider the following unimproved version of $\mathit{id}_{\Uparrow}$:

$$\begin{aligned}
&\mathit{id}_{\Uparrow} : (A : {\Uparrow}\mathsf{U}_0) \to {\Uparrow}({\sim}A \to {\sim}A)\\
&\mathit{id}_{\Uparrow} := \lambda\,A.\ \langle \lambda\,x.\,x\rangle
\end{aligned}$$

This can be used at the runtime stage as ${\sim}(\mathit{id}_{\Uparrow}\,\langle \mathsf{Bool}_0\rangle)\,\mathsf{true}_0$, which is staged to $(\lambda\,x.\,x)\,\mathsf{true}_0$. This introduces a useless β-redex, so in this case the improved version is clearly preferable.

For inductive types in general we only get oplax preservation. For example, we have $\mathsf{Bool}_1 \to {\Uparrow}\mathsf{Bool}_0$, defined as $\lambda\,b.\ \mathsf{if}\ b\ \mathsf{then}\ \langle\mathsf{true}_0\rangle\ \mathsf{then}\ \langle\mathsf{false}_0\rangle$. In the staging literature, this is called "serialization" [?], or "lifting" in the context of Template Haskell [?]. In the other direction, we can only define constant functions from ${\Uparrow}\mathsf{Bool}_0$ to Bool_1.

The lack of elimination principles for ${\Uparrow}A$ means that we cannot inspect the internal structure of expressions. This is called "generativity" in staging [?]. We will briefly discuss non-generative staging in Section **??**.

In particular, we have no serialization map from $\mathsf{Nat}_1 \to \mathsf{Nat}_1$ to ${\Uparrow}(\mathsf{Nat}_0 \to \mathsf{Nat}_0)$. However, when $A : \mathsf{U}_1$ is *finite*, and $B : \mathsf{U}_1$ can be serialized, then $A \to B$ can be serialized, because it is equivalent to a finite product. For instance, $\mathsf{Bool}_1 \to \mathsf{Nat}_1 \simeq \mathsf{Nat}_1 \times \mathsf{Nat}_1$. In 2LTT, A is called *cofibrant* [?]: this means that for each B, $A \to {\Uparrow}B$ is equivalent to ${\Uparrow}C$ for some C. This formalizes the so-called "trick" in partial evaluation, which improves binding times by η-expanding functions out of finite sums [?].

2.3.1 Fusion. Fusion optimizations can be viewed as binding time improvement techniques for general inductive types. The basic idea is that by lambda-encoding an inductive type, it is brought to a form which can be binding-time improved. For instance, consider foldr-build fusion for lists, which is employed in GHC Haskell [?]. Starting from ${\Uparrow}(\mathsf{List}_0\,A)$, we use Böhm-Berarducci encoding [?] under the lifting to get

$$ {\Uparrow}((L : \mathsf{U}_0) \to (A \to L \to L) \to L \to L) $$

which is isomorphic to

$$ (L : {\Uparrow}U_0) \to ({\Uparrow}A \to {\Uparrow}{\sim}L \to {\Uparrow}{\sim}L) \to {\Uparrow}{\sim}L \to {\Uparrow}{\sim}L. $$

Alternatively, for *stream fusion*, we embed $\mathsf{List}\,A$ into the coinductive colists (i.e. the possibly infinite lists), and use a terminal lambda-encoding. The embedding into the "larger" structure enables some staged optimizations which are otherwise not possible, such as fusion for the zip function [?]. However, the price we pay is that converting back to lists from colists is not necessarily total.

We do not detail the implementation of fusion in 2LTT here. In short, 2LTT is a natural setting for a wide range of fusion setups. A major advantage of fusion in 2LTT is the formal guarantee of staging, in contrast to implementations where compile-time computation relies on ad-hoc user annotations and general-purpose optimization passes. For instance, fusion in GHC relies on rewrite rules and inlining annotations which have to be carefully tuned and ordered, and it is quite possible to get pessimized code via failed fusion.

2.3.2 Inferring staging operations. During bidirectional elaboration [**?**], we can use the preservation isomorphisms and the quote-splice isomorphism as a coercive subtyping system. When elaboration needs to compare an inferred and an expected type, it may insert transports along isomorphisms. We implemented this feature in our prototype. It additionally supports Agda-style implicit arguments and pattern unification, so it can elaborate the following definition:

$$\begin{aligned}&\mathit{map} : \{A\,B : \Uparrow\mathsf{U}_0\} \to (\Uparrow A \to \Uparrow B) \to \Uparrow(\mathsf{List}_0\,A) \to \Uparrow(\mathsf{List}_0\,B)\\&\mathit{map} := \lambda\,f\,\mathit{as}.\,\mathsf{foldr}_0(\lambda\,a\,\mathit{bs}.\,\mathsf{cons}_0\,(f\,a)\,\mathit{bs})\,\mathsf{nil}_0\,\mathit{as}\end{aligned}$$

We may go a bit further, and also add the coercive subtyping rule $\mathsf{U}_0 \leq \mathsf{U}_1$, witnessed by $\Uparrow$. Then, the type of *map* can be written as $\{A\,B : \Uparrow\mathsf{U}_0\} \to (A \to B) \to \mathsf{List}_0\,A \to \mathsf{List}_0\,B$. However, here the elaborator has to make a choice, whether to elaborate to improved or unimproved types. In this case, the fully unimproved type would be

$$\{A\,B : \Uparrow\mathsf{U}_0\} \to \Uparrow((\sim A \to \sim B) \to \mathsf{List}_0 \sim A \to \mathsf{List}_0 \sim B).$$

It seems to the author of this paper that improved types are a sensible default, and we can insert explicit lifting when we want to opt for unimproved types. This is also available in our prototype.

2.4 Variations of Object-Level Languages

In the following, we consider variations on object-level languages, with a focus on applications in downstream compilation after staging. Adding restrictions or more distinctions to the object language can make it easier to optimize and compile it.

2.4.1 Monomorphization. In this case, the object language is simply typed, so every type is known statically. This makes it easy to assign different memory layouts to different types, and generate code accordingly for each type. Moving to 2LTT, we still want to abstract over runtime types at compile time, so we use the following setup.

- We have a *jugdment*, written as $A\,\mathsf{type}_0$, for well-formed runtime types. Runtime types may be closed under simple type formers.
- We have a type $\mathsf{Ty}_0 : \mathsf{U}_1$ in lieu of the previous $\Uparrow\mathsf{U}_0$.
- For each $A\,\mathsf{type}_0$, we have $\Uparrow A : \mathsf{Ty}_0$.
- We have quoting and splicing for types and terms. For types, we send $A\,\mathsf{type}_0$ to $\langle A\rangle : \mathsf{Ty}_0$. For terms, we send $t : A$ to $\langle t\rangle : \Uparrow A$.

Despite the restriction to simple types at runtime, we can still write arbitrary higher-rank polymorphic functions in this setup, such as a function with type $((A : \mathsf{Ty}_0) \to \Uparrow \sim A \to \Uparrow\sim A) \to \Uparrow \mathsf{Bool}_0$. This function can be only applied to statically known arguments, so the polymorphism can be staged away. The main restriction that programmers have to keep in mind is that polymorphic functions cannot be stored inside runtime data types.

This setup is fairly similar to the well-known monomorphization models in the C++ and Rust programming languages. The evident difference is that in 2LTT we can use a full type theory at compile time, as opposed to languages which are less expressive and less principled. Additionally, we will see in Section **??** that this setup is compatible with an *induction principle* for Ty_0, so that we can analyze the structure of runtime types.

2.4.2 Memory representation polymorphism. This refines monomorphization, in that types are not directly identified with memory representations, but instead representations are internalized in 2LTT as a meta-level type, and runtime types are indexed over representations.

- We have $\mathsf{Rep} : \mathsf{U}_1$ as the type of memory representations. We have considerable freedom in the specification of Rep. A simple setup may distinguish references from unboxed products, i.e. we have $\mathsf{Ref} : \mathsf{Rep}$ and $\mathsf{Prod} : \mathsf{Rep} \to \mathsf{Rep} \to \mathsf{Rep}$, and additionally we may assume any desired primitive machine representation as a Rep.
- We have Russell-style $\mathsf{U}_{0,j} : \mathsf{Rep} \to \mathsf{U}_{0,j+1}\,r$, where r is some chosen runtime representation for types; often this would mark types as erased at runtime. We leave the meta-level $\mathsf{U}_{1,j}$ hierarchy unchanged.
- We may introduce unboxed Σ types and primitive machine types in the runtime language. For $A : \mathsf{U}_0\,r$ and $B : A \to \mathsf{U}_0\,r'$, we may have $(x : A) \times B\,x : \mathsf{U}_0\,(\mathsf{Prod}\,\mathsf{r}\,\mathsf{r}')$. Thus, we have type dependency, but we do not have dependency in memory representations.

Since Rep is meta-level, there is no way to abstract over it at runtime, and during staging all Rep indices are computed to concrete canonical representations. This is a way to reconcile dependent types with control over memory layouts. The unboxed flavor of Σ ends up with a statically known flat memory representation, computed from the representations of the fields.

In principle, it should be possible to have *dependent memory layouts*; this would be a more advanced variation on type-passing polymorphism [?], where layouts of objects may depend on runtime data. For example, length-prefixed flat arrays would not be a primitive notion, but simply defined as $(n : \mathsf{Nat}) \times \mathsf{Vec}\,n\,A$. However, it seems to be highly challenging to implement runtime systems and precise garbage collection in this setting. See `sixten` [?] for an experimental implementation of dependent memory layouts.

3 FORMAL 2LTT

In this section we switch to a formal description of 2LTT, which we will use in the subsequent sections to define the staging algorithm and prove its correctness. First we describe the metatheory that we work in.

3.1 Metatheory

TODO

3.2 Models and Syntax of 2LTT

We use an algebraic specification for models, and specify the syntax as the initial model. We only handle well-formed syntactic objects and only define operations which respect syntactic definitional equality; in fact, we identify definitional equality with metatheoretic equality. First, we define the structural scaffolding of 2LTT without type formers.

Definition 3.1. A model of **basic 2LTT** consists of the following.

- A category $\mathbb{C}$ with a terminal object. We denote the set of objects as $\mathsf{Con}_{\mathbb{C}} : \mathsf{Set}$ and use capital Greek letters starting from Γ to refer to objects. The set of morphisms is $\mathsf{Sub}_{\mathbb{C}} : \mathsf{Con}_{\mathbb{C}} \to \mathsf{Con}_{\mathbb{C}} \to \mathsf{Set}$, and we use σ, δ and so on to refer to morphisms. The terminal object is written as $\bullet$ with unique morphism $\epsilon : \mathsf{Sub}_{\mathbb{C}}\,\Gamma\,\bullet$. We omit the $\mathbb{C}$ subscript if it is clear from context.
- For each $i \in \{0, 1\}$ and $j \in \mathbb{N}$, we have $\mathsf{Ty}_{i,j} : \mathsf{Con} \to \mathsf{Set}$ and $\mathsf{Tm}_{i,j} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}_{i,j}\,\Gamma \to \mathsf{Set}$, where Ty is a presheaf over $\mathbb{C}$ and $\mathsf{Tm}_{i,j}$ is a presheaf over the category of elements of $\mathsf{Ty}_{i,j}$. This means that both types (Ty) and terms (Tm) can be substituted, and substitution has functorial action. We use A, B, C to refer to types and t, u, v to refer to terms, and use $A[\sigma]$ and $t[\sigma]$ for substituting types and terms. Additionally, for each Γ : Con and

$A : \mathsf{Ty}_{i,j}\,\Gamma$, we have the extended object $\Gamma \triangleright A : \mathsf{Con}$ such that there is a natural isomorphism $\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \simeq ((\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \times \mathsf{Tm}_{i,j}\,\Gamma\,(A[\sigma]))$.
- For each j we have a *lifting structure*, consisting of a natural transformation $\Uparrow : \mathsf{Ty}_{0,j}\,\Gamma \to \mathsf{Ty}_{1,j}\,\Gamma$, and an invertible natural transformation $\langle - \rangle : \mathsf{Tm}_{0,j}\,\Gamma\,A \to \mathsf{Tm}_{1,j}\,\Gamma\,(\Uparrow A)$, with inverse $\sim -$.

In short, for each i and j we have a family structure in the sense of categories-with-families [**?**], such that there is a family morphism from each $(\mathsf{Ty}_{0,j},\ \mathsf{Tm}_{0,j})$ to $(\mathsf{Ty}_{1,j},\ \mathsf{Tm}_{1,j})$ in the sense of [**?**].

The following notions are derivable in any model:

- By moving left-to-right along $\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \simeq ((\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \times \mathsf{Tm}_{i,j}\,\Gamma\,(A[\sigma]))$, and starting from the identity morphism $\mathsf{id} : \mathsf{Sub}\,(\Gamma \triangleright A)\,(\Gamma \triangleright A)$, we recover the *weakening substitution* $\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$ and the *zero variable* $\mathsf{q} : \mathsf{Tm}_{i,j}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$.
- By weakening q, we recover a notion of variables as De Bruijn indices. In general, the n-th De Bruijn index is defined as $\mathsf{q}[\mathsf{p}^n]$, where p^n denotes n-fold composition.
- By moving right-to-left along $\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \simeq ((\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \times \mathsf{Tm}_{i,j}\,\Gamma\,(A[\sigma]))$, we recover the operation which extends a morphism with a term. In the syntax (initial model), this justifies the view of Sub as a type of lists of terms, i.e. a parallel substitutions. We denote the extension operation as $(\sigma,\ t)$ for $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$.

Notation 4. De Bruijn indices are rather hard to read, so we will often use nameful notation for binders and substitutions. For example, we may write $\Gamma \triangleright (x : A) \triangleright (y : B)$ for a context, and subsequently write $B[y \mapsto t]$ for substituting the x variable for some term $t : \mathsf{Tm}_{i,j}\,\Gamma\,A$. Using nameless notation, we would instead have $B : \mathsf{Ty}_{i,j}\,(\Gamma \triangleright A)$ and $B[\mathsf{id},\ t]$; here we recover single substitution by extending the identity substitution $\mathsf{id} : \mathsf{Sub}\,\Gamma\,\Gamma$ with t.

We may also use implicit weakening: if a type or term is in a Γ context, we may use it in an extended $\Gamma \triangleright A$ context without marking the weakening substitution.

Definition 3.2. A **model of 2LTT** is a model of basic 2LTT which supports certain type formers. For the sake of brevity, we only present our results for a small collection of type formers. However, we will argue that our results easily extend to any general notion of inductive type formers. We specify type formers in the following. We assume that all type formers are natural, i.e. stable under substitution.

- *Universes.* For each i and j, we have a Coquand-style universe [**?**] in $\mathsf{Ty}_{i,j}$. This consists of $\mathsf{U}_{i,j} : \mathsf{Ty}_{i,j+1}\,\Gamma$, together with $\mathsf{El} : \mathsf{Tm}_{i,j+1}\,\Gamma\,\mathsf{U}_{i,j} \to \mathsf{Ty}_{i,j}\,\Gamma$ and Code, where Code and El are inverses.
- *Σ-types.* We have $\Sigma\,(x : A)\,B : \mathsf{Ty}_{i,j}\,\Gamma$ for $A : \mathsf{Ty}_{i,j}\,\Gamma$ and $B : \mathsf{Ty}_{i,j}\,(\Gamma \triangleright (x : A))$, together with a natural isomorphism of pairing and projections:
$$\mathsf{Tm}_{i,j}\,\Gamma\,(\Sigma\,A\,B) \simeq ((t : \mathsf{Tm}_{i,j}\,\Gamma\,A) \times \mathsf{Tm}_{i,j}\,\Gamma\,(B[x \mapsto t])$$
We write $(t,\ u)$ for pairing and fst and snd for projections.
- *Function types.* We have $\Pi\,(x : A)\,B : \mathsf{Ty}_{i,j}\,\Gamma$ for $A : \mathsf{Ty}_{i,j}\,\Gamma$ and $B : \mathsf{Ty}_{i,j}\,(\Gamma \triangleright (x : A))$, together with $\mathsf{app} : \mathsf{Tm}_{i,j}\,\Gamma\,(\Pi\,A\,B) \to \mathsf{Tm}_{i,j}\,(\Gamma \triangleright (x : A))\,B$ and its inverse lam.

- *Natural numbers.* We have $\mathsf{Nat}_{i,j} : \mathsf{Ty}_{i,j}\,\Gamma$, $\mathsf{zero}_{i,j} : \mathsf{Tm}_{i,j}\,\Gamma\,\mathsf{Nat}_{i,j}$, and $\mathsf{suc}_{i,j} : \mathsf{Tm}_{i,j}\,\Gamma\,\mathsf{Nat}_{i,j} \to \mathsf{Tm}_{i,j}\,\Gamma\,\mathsf{Nat}_{i,j}$. The eliminator is the following.

$$
\begin{aligned}
\mathsf{NatElim} :\ & (P : \mathsf{Ty}_{i,k}\,(\Gamma \triangleright (n : \mathsf{Nat}_{i,j})) \\
& (z\ : \mathsf{Tm}_{i,k}\,\Gamma\,(P[n \mapsto \mathsf{zero}_{i,j}])) \\
& (s\ : \mathsf{Tm}_{i,k}\,(\Gamma \triangleright (n : \mathsf{Nat}_{i,j}) \triangleright (pn : P[n \mapsto n]))\,(P[n \mapsto \mathsf{suc}_{i,j}\,n])) \\
& (t\ : \mathsf{Tm}_{i,j}\,\Gamma\,\mathsf{Nat}_{i,j}) \\
\to\ & \ \mathsf{Tm}_{i,j}\,\Gamma\,(P[n \mapsto t]))
\end{aligned}
$$

We also have the β-rules:

$$
\begin{aligned}
\mathsf{NatElim}\,P\,z\,s\,\mathsf{zero}_{i,j} &= z \\
\mathsf{NatElim}\,P\,z\,s\,(\mathsf{suc}_{i,j}\,t) &= s[n \mapsto t,\ pn \mapsto \mathsf{NatElim}\,P\,z\,s\,t]
\end{aligned}
$$

Note that we can eliminate from a level j to any level k.

Definition 3.3. The **syntax of 2LTT** is defined to be the initial model of 2LTT. The syntax can be assumed to exist, since the notion of model is algebraic; the specification can be expressed using a suitable notion of algebraic signatures, such as one for essentially algebraic theories [**?**] or as a finitary quotient inductive-inductive signature [**?**]. The initiality of syntax directly yields a notion of *recursion on syntax*, and it can be shown that initiality also implies an *induction principle* [**?**].

The examples in Section **??** can be faithfully represented in the formal syntax. The only notable difference is that the formal syntax uses Coquand-style universes instead of Russell-style ones. We use the former because it is much easier to model in the semantics in Section **??**. We note though that it is straightforward to elaborate from a Russell-style surface syntax to the formal version, by inserting El and Code as required.

3.2.1 Comparison to Annekov et al. Comparing our models to the primary reference on 2LTT [1], the main difference is the handling of "sizing" levels. In ibid. there is a cumulative lifting from $\mathsf{Ty}_{i,j}$ to $\mathsf{Ty}_{i,j+1}$, which we do not assume. Instead, we allow elimination from $\mathsf{Nat}_{i,j}$ into any k level. This means that we can manually define lifting maps from $\mathsf{Nat}_{i,j}$ to $\mathsf{Nat}_{i,j+1}$ by elimination. This is more similar to e.g. Agda, where we do not have cumulativity, but we can define explicit lifting from $\mathsf{Nat}_j : \mathsf{Set}_j$ to $\mathsf{Nat}_k : \mathsf{Set}_k$.

In [1], "two-level type theory" specifically refers to the setup where the object-level is a homotopy type theory and the meta level is an extensional type theory. In contrast, we allow a wider range of setups under the 2LTT umbrella. Annekov et al. also considers a range of additional strengthenings and extension of 2LTT [1, Section 2.4], most of which are of use in synthetic homotopy theory. We do not assume any of these, and stick to the most basic formulation of 2LTT.

3.3 Models and Syntax of the Object Theory

We also need to specify the object theory, which serves as the output language of staging. In general, the object theory corresponding to a particular flavor of 2LTT is simply the type theory that supports only the object-level $\mathsf{Ty}_{0,j}$ hierarchy with its type formers.

Definition 3.4. A **model of the object theory** is a category-with-families, with types and terms indexed over $j \in \mathbb{N}$, supporting Coquand-style universes U_j, type formers Π, Σ and Nat_j, with elimination from Nat_j to any level k.

Definition 3.5. Like before, the **syntax of the object theory** is the initial model.

Notation 5. From now on, by default we use Con, Sub, Ty and Tm to refer to sets in the syntax of 2LTT. We use $\mathbb{O}$ to refer to the object syntax, and $\mathsf{Con}_{\mathbb{O}}$, $\mathsf{Sub}_{\mathbb{O}}$, $\mathsf{Ty}_{\mathbb{O}}$ and $\mathsf{Tm}_{\mathbb{O}}$ to refer to its underlying sets.

Definition 3.6. We have an **embedding** of object syntax syntax into 2LTT syntax, consisting of the following functions:

$$\ulcorner - \urcorner : \mathsf{Con}_{\mathbb{O}} \to \mathsf{Con}$$
$$\ulcorner - \urcorner : \mathsf{Sub}_{\mathbb{O}}\,\Gamma\,\Delta \to \mathsf{Sub}\,\ulcorner\Gamma\urcorner\,\ulcorner\Delta\urcorner$$
$$\ulcorner - \urcorner : \mathsf{Ty}_{\mathbb{O}j}\,\Gamma \to \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$$
$$\ulcorner - \urcorner : \mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,A \to \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,\ulcorner A\urcorner$$

Embedding maps all type and term formers to the corresponding ones in 2LTT, and strictly preserves all structure.

4 THE STAGING ALGORITHM

In this section we specify what we mean by a staging algorithm, then proceed to define one.

Definition 4.1. A **staging algorithm** consists of two functions:

$$\mathsf{Stage} : \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner \to \mathsf{Ty}_{\mathbb{O}j}\,\Gamma$$
$$\mathsf{Stage} : \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,A \to \mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,(\mathsf{Stage}\,A)$$

Note that we can stage open types and terms as long as their contexts are purely object-level. By *closed staging* we mean staging only for closed types and terms.

Definition 4.2. A staging algorithm Stage is **correct** if the following properties hold:

- *Soundness:* $\ulcorner\mathsf{Stage}\,A\urcorner = A$ and $\ulcorner\mathsf{Stage}\,t\urcorner = t$.
- *Completeness:* staging respects definitional equality.
- *Stability:* $\mathsf{Stage}\,\ulcorner A\urcorner = A$ and $\mathsf{Stage}\,\ulcorner t\urcorner = t$.

We make some remarks on correctness. First, we get completeness for free in our algebraic setup, since *all* functions definable in the metatheory must respect definitional equality.

Second, note that soundness and stability together is the statement that embedding is a bijection on types and terms. This is a statement of *conservativity* of 2LTT over the object theory. In [**?**] a significantly weaker conservativity theorem is shown, which expresses that there exists a function from $\mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,\ulcorner A\urcorner$ to $\mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,A$.

Lastly, we note that the correctness terminology is borrowed from previous works on normalization-by-evaluation, where the soundness-completeness-stability trio has been used several times [**?**]. In [**?**], stability is instead called "idempotence".

4.1 The Presheaf Model

We observe that the presheaf model described in [1, Section 2.5.3] immediately yields a closed staging algorithm, by evaluation of 2LTT types and terms in the model. In this model, contexts are presheaves over the syntactic category of the object theory. We denote the components of the model by putting hats on 2LTT components, e.g. as in $\widehat{\mathsf{Con}}$. We summarize the key components in the following.

Notation 6. In this section, we switch to naming elements of $\mathsf{Con}_{\mathbb{O}}$ as a, b and c, and elements of $\mathsf{Sub}_{\mathbb{O}}$ as f, g, and h, to avoid name clashing with contexts and substitutions in the presheaf model.

4.1.1 The syntactic category and the meta-level fragment.

Definition 4.3. $\widehat{\mathbf{Con}} : \mathbf{Set}_{\omega+1}$ is defined as the set of presheaves over $\mathbb{O}$. $\Gamma : \widehat{\mathsf{Con}}$ has an action on objects $|\Gamma| : \mathsf{Con}_{\mathbb{O}} \to \mathsf{Set}_\omega$ and an action on morphisms $-[-] : |\Gamma|\, b \to \mathsf{Sub}_{\mathbb{O}}\, a\, b \to |\Gamma|\, a$, such that $\gamma[\mathsf{id}] = \gamma$ and $\gamma[f \circ g] = \gamma[f][g]$.

Notation 7. Above, we reused the substitution notation $-[-]$ for the action on morphisms. Also, we use lowercase γ and δ to denote elements of $|\Gamma|\, a$ and $|\Delta|\, a$ respectively.

Definition 4.4. $\widehat{\mathbf{Sub}}\, \Gamma\, \Delta : \mathbf{Set}_\omega$ is the set of natural transformations from Γ to Δ. $\sigma : \widehat{\mathsf{Sub}}\, \Gamma\, \Delta$ has action $|\sigma| : \{a : \mathsf{Con}_{\mathbb{O}}\} \to |\Gamma|\, a \to |\Delta|\, a$ such that $|\sigma|\, (\gamma[f]) = (|\sigma|\, \gamma)[f]$.

Definition 4.5. $\widehat{\mathbf{Ty}}_{1,j}\, \Gamma : \mathbf{Set}_\omega$ is the set of *displayed presheaves* over Γ; see e.g. [**?**]. This is equivalent to the set of presheaves over the category of elements of Γ, but it is usually more convenient in calculations. An $A : \widehat{\mathsf{Ty}}\, \Gamma$ has an action on objects $|A| : \{a : \mathsf{Con}_{\mathbb{O}}\} \to |\Gamma|\, a \to \mathsf{Set}_j$ and an action on morphisms $-[-] : |A|\, \gamma \to (f : \mathsf{Sub}_{\mathbb{O}}\, a\, b) \to |A|\, (\gamma[f])$, such that $\alpha[\mathsf{id}] = \alpha$ and $\alpha[f \circ g] = \alpha[f][g]$.

Notation 8. We write α and β respectively for elements of $|A|\, \gamma$ and $|B|\, \gamma$.

Definition 4.6. $\widehat{\mathbf{Tm}}_{1,j}\, \Gamma\, A : \mathbf{Set}_\omega$ is the set of sections of the displayed presheaf A. This can be viewed as a dependently typed analogue of a natural transformation. A $t : \widehat{\mathsf{Tm}}_{1,j}\, \Gamma\, A$ has action $|t| : \{a\} \to (\gamma : |\Gamma|\, a) \to |A|\, \gamma$ such that $|t|\, (\gamma[f]) = (|t|\, \gamma)[f]$.

We also look at he empty context and context extension with meta-level types, as these will appear in subsequent definitions.

Definition 4.7. $\widehat{\bullet} : \widehat{\mathbf{Con}}$ is defined as the presheaf which is constantly $\top$, i.e. $|\widehat{\bullet}|\, \_ = \top$.

Definition 4.8. For $A : \widehat{\mathsf{Ty}}_{1,j}\, \Gamma$, we define $\mathbf{\Gamma} \mathbin{\widehat{\triangleright}} \boldsymbol{A}$ pointwise by $|\Gamma \mathbin{\widehat{\triangleright}} A|\, a := (\gamma : |\Gamma|\, a) \times |A|\, \gamma$ and $(\gamma, \alpha)[f] := (\gamma[f], \alpha[f])$.

Using the above definitions, we can model the syntactic category of 2LTT, and also the meta-level family structure and all meta-level type formers. For an exposition in previous literature, see [**?**] and [**?**].

4.1.2 The object-level fragment. We move on to modeling the object-level syntactic fragment of 2LTT. We make some preliminary definitions. First, note the types in the object theory yield a presheaf, and terms yield a displayed presheaf over them; this immediately follows from the specification of a family structure in a cwf. Hence, we do a bit of a name overloading, and have $\mathsf{Ty}_{\mathbb{O}j} : \widehat{\mathsf{Con}}$ and $\mathsf{Tm}_{\mathbb{O}j} : \widehat{\mathsf{Ty}}\, \mathsf{Ty}_{\mathbb{O}j}$.

Definition 4.9. $\widehat{\mathbf{Ty}}_{0,j}\, \Gamma : \mathbf{Set}_\omega$ is defined as $\widehat{\mathsf{Sub}}\, \Gamma\, \mathsf{Ty}_{\mathbb{O}j}$, and $\widehat{\mathbf{Tm}}_{0,j}\, \Gamma\, \boldsymbol{A} : \mathbf{Set}_\omega$ is defined as $\widehat{\mathsf{Tm}}\, \Gamma\, (\mathsf{Tm}_{\mathbb{O}j}[A])$.

For illustration, if $A : \widehat{\mathsf{Ty}}_{0,j}\, \Gamma$, then $A : \widehat{\mathsf{Sub}}\, \Gamma\, \mathsf{Ty}_{\mathbb{O}j}$, so $|A| : \{a : \mathsf{Con}_{\mathbb{O}}\} \to |\Gamma|\, a \to \mathsf{Ty}_{\mathbb{O}j}\, a$. In other words, the action of A on objects maps a semantic context to a syntactic object-level type. Likewise, for $t : \widehat{\mathsf{Tm}}_{0,j}\, \Gamma\, A$, we have $|t| : (\gamma : |\Gamma|\, a) \to \mathsf{Tm}_{\mathbb{O}j}\, a\, (|A|\, \gamma)$, so we get a syntactic object-level term as output.

Definition 4.10. For $A : \mathsf{Ty}_{0,j}\, \Gamma$, we define $\mathbf{\Gamma} \mathbin{\widehat{\triangleright}} \boldsymbol{A}$ as $|\Gamma \mathbin{\widehat{\triangleright}} A|\, a := (\gamma : |\Gamma|\, a) \times \mathsf{Tm}_{\mathbb{O}j}\, a\, (|A|\, \gamma)$ and $(\gamma, t)[f] := (\gamma[f], t[f])$. Thus, extending a semantic context with an object-level type appends an object-level term.

Using the above definitions and following [?], we can model all type formers in $\widehat{\mathsf{Ty}}_{0,j}$. Intuitively, that is because $\widehat{\mathsf{Ty}}_{0,j}$ and $\widehat{\mathsf{Tm}}_{0,j}$ return types and terms, so we can reuse the type and term formers in the object theory.

4.1.3 Lifting.

Definition 4.11. $\widehat{\Uparrow} A : \widehat{\mathbf{Ty}}_{1,j}\,\Gamma$ is defined as $\mathsf{Tm}_{\mathbb{O}j}\,[A]$. With this, we get that $\widehat{\mathsf{Tm}}_{0,j}\,\Gamma\,A$ is equal to $\widehat{\mathsf{Tm}}_{1,j}\,\Gamma\,(\widehat{\Uparrow} A)$. Hence, we can define both quoting and splicing as identity functions in the model.

4.2 Closed Staging

Definition 4.12. The **evaluation morphism, denoted** $\mathbb{E}$ is a 2LTT model morphism from the syntax of 2LTT to $\hat{\mathbb{O}}$, arising from the initiality of the syntax. In other words, $\mathbb{E}$ is defined by recursion on the syntax and strictly preserves all structure.

Note that $\widehat{\bullet}$ is defined as the terminal presheaf, which is constantly $\top$. This leads us to the following definition.

Definition 4.13. **Closed staging** is defined as follows.

$$\begin{array}{ll} \mathsf{Stage} : \mathsf{Ty}_{0,j}\,\bullet \to \mathsf{Ty}_{\mathbb{O}j}\,\bullet & \mathsf{Stage} : \mathsf{Tm}_{0,j}\,\bullet\,A \to \mathsf{Tm}_{\mathbb{O}j}\,\bullet\,(\mathsf{Stage}\,A) \\ \mathsf{Stage}\,A := |\mathbb{E}\,A|\,\{\bullet\}\,\mathsf{tt} & \mathsf{Stage}\,t := |\mathbb{E}\,t|\,\{\bullet\}\,\mathsf{tt} \end{array}$$

This is well-typed, since $|\mathbb{E}\,A|\{\bullet\} : |\mathbb{E}\,\bullet|\,\bullet \to \mathsf{Ty}_{\mathbb{O}j}\,\bullet$, so $|\mathbb{E}\,A|\{\bullet\} : \top \to \mathsf{Ty}_{\mathbb{O}j}\,\bullet$, and $|\mathbb{E}\,t|\,\{\bullet\} : \top \to \mathsf{Tm}_{\mathbb{O}j}\,\bullet\,(|\mathbb{E}\,A|\,\mathsf{tt})$.

What about general (open) staging though? Given $A : \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$, we get $|\mathbb{E}\,A|\,\{\Gamma\} : |\ulcorner\Gamma\urcorner|\,\Gamma \to \mathsf{Ty}_{\mathbb{O}j}$. We need an element of $|\ulcorner\Gamma\urcorner|\,\Gamma$, in order to extract an object-level type. Such "generic" semantic contexts should be possible to construct, because elements of $|\ulcorner\Gamma\urcorner|\,\Gamma$ are essentially lists of object-level terms in Γ, by Definition **??**, so $|\ulcorner\Gamma\urcorner|\,\Gamma$ should be isomorphic to $\mathsf{Sub}_{\mathbb{O}}\,\Gamma\,\Gamma$. It turns out that this falls out from the stability proof of $\mathbb{E}$.

4.3 Stability and Open Staging

Stability means that staging an object-level construction does nothing. We define a family of functions $-^P$ by induction on object syntax. The induction motives are as follows.

$$\begin{array}{ll} (\Gamma : \mathsf{Con}_{\mathbb{O}})^P & : |\mathbb{E}\,\ulcorner\Gamma\urcorner|\,\Gamma \\ (\sigma : \mathsf{Sub}_{\mathbb{O}}\,\Gamma\,\Delta)^P & : \Delta^P[\sigma] = |\mathbb{E}\,\ulcorner\sigma\urcorner|\,\Gamma^P \\ (A : \mathsf{Ty}_{\mathbb{O}j}\,\Gamma)^P & : A = |\mathbb{E}\,\ulcorner A\urcorner|\,\Gamma^P \\ (t : \mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,A)^P & : t = |\mathbb{E}\,\ulcorner t\urcorner|\,\Gamma^P \end{array}$$

We look at the interpretation of contexts.

- For $\bullet^P$, we need an element of $|\mathbb{E}\,\ulcorner\bullet\urcorner|\,\bullet$, hence an element of $\top$, so we define $\bullet^P$ as tt.
- For $(\Gamma \triangleright A)^P$, we need an element of

$$(\gamma : |\mathbb{E}\ulcorner\Gamma\urcorner|\,(\Gamma \triangleright A)) \times \mathsf{Tm}_{\mathbb{O}j}\,(\Gamma \triangleright A)\,(|\mathbb{E}\ulcorner A\urcorner|\,\gamma).$$

  We have $\Gamma^P : |\mathbb{E}\ulcorner\Gamma\urcorner|\,\Gamma$, which we can weaken as $\Gamma^P[\mathsf{p}] : |\mathbb{E}\ulcorner\Gamma\urcorner|\,(\Gamma \triangleright A)$, so we set the first projection of the result as $\Gamma^P[\mathsf{p}]$. For the second projection, the goal type can be simplified as follows:

$$\begin{aligned} & \mathsf{Tm}_{\mathbb{O}j}\,(\Gamma \triangleright A)\,(|\mathbb{E}\ulcorner A\urcorner|\,(\Gamma^P[\mathsf{p}])) & \\ = \; & \mathsf{Tm}_{\mathbb{O}j}\,(\Gamma \triangleright A)\,((|\mathbb{E}\ulcorner A\urcorner|\,\Gamma^P)[\mathsf{p}]) & \text{by naturality of } \mathbb{E}\ulcorner A\urcorner \\ = \; & \mathsf{Tm}_{\mathbb{O}j}\,(\Gamma \triangleright A)\,(A[\mathsf{p}]) & \text{by } A^P \end{aligned}$$

We have the zero de Bruijn variable $\mathsf{q} : \mathsf{Tm}_{\mathbb{O}j}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$. Hence, we define $(\Gamma \triangleright A)^P$ as $(\Gamma^P[\mathsf{p}],\,\mathsf{q})$.

Thus, a generic semantic context $\Gamma^P : |\mathbb{E}\ulcorner\Gamma\urcorner|\,\Gamma$ is just a list of variables, corresponding to the identity substitution $\mathsf{id} : \mathsf{Sub}_{\mathbb{O}}\,\Gamma\,\Gamma$ which maps each variable to itself.

The rest of the $-^P$ interpretation is straightforward and we omit it here. In particular, preservation of definitional equalities is automatic, since types, terms and substitutions are all interpreted as proof-irrelevant equations.

Definition 4.14. We define **open staging** as follows.

$$\begin{aligned} & \mathsf{Stage} : \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner \to \mathsf{Ty}_{\mathbb{O}j}\,\Gamma & \qquad & \mathsf{Stage} : \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,A \to \mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,(\mathsf{Stage}\,A) \\ & \mathsf{Stage}\,A := |\mathbb{E}\,A|\,\Gamma^P & & \mathsf{Stage}\,t := |\mathbb{E}\,t|\,\Gamma^P \end{aligned}$$

Theorem 4.15. *Open staging is stable.*

Proof. For $A : \mathsf{Ty}_{\mathbb{O}j}$, $\mathsf{Stage}\,\ulcorner A\urcorner$ is by definition $|\mathbb{E}\,\ulcorner A\urcorner|\,\Gamma^P$, hence by A^P it is equal to A. Likewise, $\mathsf{Stage}\,\ulcorner t\urcorner$ is equal to t by t^P. □

4.4 Implementation and Efficiency

We have defined staging in a constructive metatheory, so it is possible to extract an implementation from it. We discuss its operational behavior and potential optimizations in the following. First, consider the computational part of evaluation:

$$\begin{aligned} & |-| \circ \mathbb{E} : \mathsf{Ty}_{1j}\,\Gamma \to \{\Delta : \mathsf{Con}_{\mathbb{O}}\} \to |\mathbb{E}\,\Gamma|\,\Delta \to \mathsf{Set}_j \\ & |-| \circ \mathbb{E} : \mathsf{Ty}_{0j}\,\Gamma \to \{\Delta : \mathsf{Con}_{\mathbb{O}}\} \to |\mathbb{E}\,\Gamma|\,\Delta \to \mathsf{Ty}_{\mathbb{O}j}\,\Delta \\ & |-| \circ \mathbb{E} : \mathsf{Tm}_{1j}\,\Gamma\,A \to \{\Delta : \mathsf{Con}_{\mathbb{O}}\} \to (\gamma : |\mathbb{E}\,\Gamma|\,\Delta) \to |\mathbb{E}\,A|\,\gamma \\ & |-| \circ \mathbb{E} : \mathsf{Tm}_{0j}\,\Gamma\,A \to \{\Delta : \mathsf{Con}_{\mathbb{O}}\} \to (\gamma : |\mathbb{E}\,\Gamma|\,\Delta) \to \mathsf{Tm}_{\mathbb{O}\,j}\,\Delta\,(|\mathbb{E}\,A|\,\gamma) \end{aligned}$$

Operationally, we interpret syntactic types or terms in a semantic environment $\gamma : |\mathbb{E}\,\Gamma|\,\Delta$. These environments are lists containing object-level terms and semantic values. The meta-level semantic values are represented using meta-level inductive types and the function space of the metatheory.

- $|\mathsf{Nat}_1|\,\_$ is simply the set of meta-level natural numbers.
- $|\Sigma_1\,A\,B|\,\gamma$ is simply a set of pairs of values.
- In the non-dependent case, a function type $A \mathrel{\widehat{\to}} B$ is defined as the presheaf exponential [**?**]. The computational part of $|A \mathrel{\widehat{\to}} B|\,\{\Delta\}\,\gamma$ is given by a function with type

  $$(\Theta : \mathsf{Con}_{\mathbb{O}}) \to (\sigma : \mathsf{Sub}_{\mathbb{O}}\,\Theta\,\Delta) \to |A|\,(\gamma[\sigma]) \to |B|\,(\gamma[\sigma])$$

  This may be also familiar as the semantic implication from the Kripke semantics of intuitionistic logics. Whenever we evaluate a function application, we supply an extra $\mathsf{id} : \mathsf{Sub}_{\mathbb{O}}\,\Delta\,\Delta$. This may incur cost via the $\gamma[\mathsf{id}]$ restrictions in a naive implementation, but this is easy to optimize, by introducing a formal representation of id, such that $\gamma[\mathsf{id}]$ immediately computes to γ. The dependent case is analogous operationally.

In summary, in the meta-level fragment of 2LTT, $\mathbb{E}$ yields a reasonably efficient computation of closed values, which reuses functions from the ambient metatheory. Alternatively, instead of using ambient functions, we could use our own implementation of function closures during staging.

In contrast, we have a bit of an efficiency problem in the handling of object-level terms and binders: whenever we go under an object-level binder we have to weaken the current semantic context. In other words, when moving from Δ to $\Delta \triangleright A$, we have to shift $\gamma : |\mathbb{E}\,\Gamma|\,\Delta$ to $\gamma[\mathsf{p}] : |\mathbb{E}\,\Gamma|\,(\Delta \triangleright A)$. This cannot be avoided by an easy optimization trick.

The exact same efficiency issue occurs in formal presheaf-based normalization-by-evaluation [**?**]. However, in practical implementations [**?**] this issue can be fully solved by using De Bruijn *levels* in the semantic domain, thus arranging that weakening has no operational cost on semantic values. This corresponds to Abel's "liftable terms" presentation [**?**]. We can use the same solution in staging, by using de Bruijn levels and delayed weakenings in object-level terms during staging. It is a somewhat unfortunate mismatch that indices are far more convenient in formalization, but levels are more efficient in practice.

In our prototype implementation, we use the above optimization with De Bruijn levels, and we also switch to *syntax-directed* staging. In the formal staging algorithm, since types are tracked, quotes and splices have no operational significance; recall that they are interpreted as identity functions in Definition **??**. However, it seems more efficient to not compute types at all and instead rely on quotes and splices to keep track of the current stage.

4.4.1 Caching. In a production-strength implementation we would need some caching mechanism, to avoid excessive duplication of function code. For example, if we use the staged *map* function multiple times with the same type and function arguments, we do not want to generate separate code for each usage. De-duplicating object-level code with *function types* is usually safe, since function bodies become closed top-level code after closure conversion. We leave this to future work.

5 YONEDA, REPRESENTABILITY AND GENERATIVITY

6 SOUNDNESS OF STAGING

In this section we prove soundness of staging. We build a proof-relevant logical relation between the evaluation morphism $\mathbb{E}$ and a *restriction* morphism, which restricts 2LTT syntax to object-level contexts. The relational interpretations of $\mathsf{Ty}_{0,j}$ and $\mathsf{Tm}_{0,j}$ will yield the soundness property.

6.1 Working in $\hat{\mathbb{O}}$

We have seen that 2LTT can be modeled in the presheaf topos $\hat{\mathbb{O}}$. Additionally, $\hat{\mathbb{O}}$ supports all type formers of extensional type theory [**?**], and certain other structures which are stable under object-level substitution. Since almost all constructions in this section must be stable under object-level substitution, it makes sense to work internally to $\hat{\mathbb{O}}$. This technique has been previously used in normalization proofs [**?**] and also in the metatheory of cubical type theories [**?**].

When we work internally in a model of a type theory, we do not explicitly refer to contexts, types, and substitutions. For example, when working in Agda, we do not refer to Agda's typing contexts. Instead, we only work with terms, and use functions and universes to abstract over types and semantic contexts. Hence, we have to convert along certain isomorphisms when we switch between the internal and external views. In the following, we summarize features in $\hat{\mathbb{O}}$ and also the internal-external conversions.

We write $\widehat{\mathsf{Set}}_j$ for ordinal-indexed Russell-style universes. Formally, we have Coquand-style universes, but for the sake of brevity we omit El and Code from internal syntax. Universes are cumulative, and closed under Π, Σ, extensional identity $- = -$ and inductive types. We use the same conventions as in Notation **??**.

The basic scheme for internalization is as follows:

$$\begin{array}{lll}
\Gamma : \widehat{\mathsf{Con}} & \text{is internalized as} & \Gamma : \widehat{\mathsf{Set}}_\omega \\
\sigma : \widehat{\mathsf{Sub}}\,\Gamma\,\Delta & \text{is internalized as} & \sigma : \Gamma \to \Delta \\
A : \widehat{\mathsf{Ty}}_{1,j}\,\Gamma & \text{is internalized as} & A : \Gamma \to \widehat{\mathsf{Set}}_j \\
t : \widehat{\mathsf{Tm}}_{1,j}\,\Gamma\,A & \text{is internalized as} & t : (\gamma : \Gamma) \to A\,\gamma
\end{array}$$

6.1.1 Object-theoretic syntax. The syntax of the object theory is clearly fully stable under object-theoretic substitution, so we can internalize all of its type and term formers. We internalize object-level types and terms as $\mathsf{Ty}_{\mathbb{O}j} : \widehat{\mathsf{Set}}_0$ and $\mathsf{Tm}_{\mathbb{O}j} : \mathsf{Ty}_{\mathbb{O}j} \to \widehat{\mathsf{Set}}_0$. $\mathsf{Ty}_{\mathbb{O}j}$ is closed under type formers. For instance, we have

$$\mathsf{Nat}_j : \mathsf{Ty}_{\mathbb{O}j} \qquad \mathsf{zero}_j : \mathsf{Tm}_{\mathbb{O}j}\,\mathsf{Nat}_j \qquad \mathsf{suc}_j : \mathsf{Tm}_{\mathbb{O}j}\,\mathsf{Nat}_j \to \mathsf{Tm}_{\mathbb{O}j}\,\mathsf{Nat}_j$$

together with $\mathsf{NatElim}$, and likewise we have all other type formers.

6.1.2 Internal $\mathbb{E}$. We also use an internal view of $\mathbb{E}$ which maps 2LTT syntax to internal values; i.e. we compose $\mathbb{E}$ with internalization. Below, note that the input to $\mathbb{E}$ is *external*, so we mark $\mathbb{E}$ as being parametrized by external input.

$$\begin{array}{ll}
\mathbb{E}\,(\Gamma\,:\,\mathsf{Con}) & : \widehat{\mathsf{Set}}_\omega \\
\mathbb{E}\,(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) & : \mathbb{E}\,\Gamma \to \mathbb{E}\,\Delta \\
\mathbb{E}\,(A : \mathsf{Ty}_{1,j}\,\Gamma) & : \mathbb{E}\,\Gamma \to \widehat{\mathsf{Set}}_j \\
\mathbb{E}\,(t\ : \mathsf{Tm}_{1,j}\,\Gamma) & : (\gamma : \mathbb{E}\,\Gamma) \to \mathbb{E}\,A\,\gamma \\
\mathbb{E}\,(A : \mathsf{Ty}_{0,j}\,\Gamma) & : \mathbb{E}\,\Gamma \to \mathsf{Ty}_{\mathbb{O}_j} \\
\mathbb{E}\,(t\ : \mathsf{Tm}_{0,j}\,\Gamma\,A) & : (\gamma : \mathbb{E}\,\Gamma) \to \mathsf{Tm}_{\mathbb{O}j}\,(\mathbb{E}\,A\,\gamma)
\end{array}$$

6.1.3 Object-level fragment of 2LTT. The purely object-level syntactic fragment of 2LTT can be internalized as follows. We define externally the presheaf of object-level types as $|\mathsf{Ty}_{0,j}|\,a := \mathsf{Ty}_{0,j}\,\ulcorner a \urcorner$, and the displayed presheaf of object-level terms over $\mathsf{Ty}_{0,j}$ as $|\mathsf{Tm}_{0,j}|\,\{a\}\,A := \mathsf{Tm}_{0,j}\,\ulcorner a \urcorner\,A$. Hence, internally we have $\mathsf{Ty}_{0,j} : \widehat{\mathsf{Set}}_0$ and $\mathsf{Tm}_{0,j} : \mathsf{Ty}_{0,j} \to \widehat{\mathsf{Set}}_0$. $\mathsf{Ty}_{0,j}$ is closed under all type formers, analogously as we have seen for $\mathsf{Ty}_{\mathbb{O}j}$.

6.1.4 Embedding. Now, the embedding operation $\ulcorner - \urcorner$ can be also internalized on types and terms, as $\ulcorner - \urcorner : \mathsf{Ty}_{\mathbb{O}j} \to \mathsf{Ty}_{0,j}$, and $\ulcorner - \urcorner : \mathsf{Tm}_{\mathbb{O}\,j}\,A \to \mathsf{Tm}_{0,j}\,\ulcorner A \urcorner$. Embedding strictly preserves all structure.

6.2 The Restriction Morphism

We define a family of functions $\mathbb{R}$ from the 2LTT syntax to objects in $\hat{\mathbb{O}}$. This will be related to the evaluation morphism $\mathbb{E}$ in the relational interpretation. In short, $\mathbb{R}$ restricts 2LTT syntax so that it can only depend on object-level contexts, i.e. contexts given as $\ulcorner \Gamma \urcorner$.

Definition 6.1. We specify the types of the **restriction operations** internally, and the $|-|$ components of the operations externally. The naturality of $|-|$ is straightforward in each case.

$$\begin{array}{ll} \mathbb{R}\,(\Gamma : \mathsf{Con}) : \widehat{\mathsf{Set}}_0 & \mathbb{R}\,(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) : \mathbb{R}\,\Gamma \to \mathbb{R}\,\Delta \\ |\mathbb{R}\,\Gamma|\,\Delta := \mathsf{Sub}\,\ulcorner\Delta\urcorner\,\Gamma & |\mathbb{R}\,\sigma|\,\gamma := \sigma \circ \gamma \end{array}$$

$$\begin{array}{ll} \mathbb{R}\,(A : \mathsf{Ty}_{1,j}\,\Gamma) : \mathbb{R}\,\gamma \to \widehat{\mathsf{Set}}_0 & \mathbb{R}\,(t : \mathsf{Tm}_{1,j}\,\Gamma\,A) : (\gamma : \mathbb{R}\,\Gamma) \to \mathbb{R}\,A\,\gamma \\ |\mathbb{R}\,A|\,\{\Delta\}\,\gamma := \mathsf{Tm}_{1,j}\,\ulcorner\Delta\urcorner\,(A[\gamma]) & |\mathbb{R}\,t|\,\gamma := t[\gamma] \end{array}$$

$$\begin{array}{ll} \mathbb{R}\,(A : \mathsf{Ty}_{0,j}\,\Gamma) : \mathbb{R}\,\Gamma \to \mathsf{Ty}_{0,j} & \mathbb{R}\,(t : \mathsf{Tm}_{0,j}\,\Gamma\,A) : (\gamma : \mathbb{R}\,\Gamma) \to \mathsf{Tm}_{0,j}\,(\mathbb{R}\,A\,\gamma) \\ |\mathbb{R}\,A|\,\gamma := A[\gamma] & |\mathbb{R}\,t|\,\gamma := t[\gamma] \end{array}$$

6.2.1 Preservation properties of $\mathbb{R}$. First, we note that $\mathbb{R}$ strictly preserves id, $- \circ -$ and type/term substitution, and it preserves $\bullet$ and $- \triangleright_1 -$ up to isomorphism. We have the following isomorphisms internally to $\hat{\mathbb{O}}$:

$$\begin{aligned} \mathbb{R}_\bullet\ &: \mathbb{R}\,\bullet \simeq \top \\ \mathbb{R}_{\triangleright_1} &: \mathbb{R}\,(\Gamma \triangleright_1 A) \simeq ((\gamma : \mathbb{R}\,\Gamma) \times \mathbb{R}\,A\,\gamma) \end{aligned}$$

Notation 9. When we have an isomorphism $f : A \simeq B$, we may write f for the function in $A \to B$, and $f^{-1} : B \to A$ for its inverse.

Notation 10. We can use a pattern matching notation on isomorphisms. For example, if $f : A \simeq B$, then we may write $(\lambda\,(f\,a).\,t) : B \to C$, and likewise $(\lambda\,(f^{-1}\,b).\,t) : A \to C$, where the function bodies can refer to the bound $a : A$ and $b : B$ variables.

The above preservation properties mean that $\mathbb{R}$ is a *pseudomorphism* in the sense of [3], between the syntactic cwf given by $(\mathsf{Ty}_{1,j}, \mathsf{Tm}_{1,j})$ and the corresponding cwf structure in $\hat{\mathbb{O}}$. In ibid. there is an analysis of such cwf morphisms, from which we obtain the following additional preservation properties:

- Meta-level Σ types are preserved up to isomorphism, so we have
$$\mathbb{R}_\Sigma : \mathbb{R}\,(\Sigma\,A\,B)\,\gamma \simeq ((\alpha : \mathbb{R}\,A) \times \mathbb{R}\,B\,(\mathbb{R}^{-1}_{\triangleright_1}\,(\gamma,\,\alpha))).$$
The semantic values of $\mathbb{R}\,(\Sigma\,A\,B)\,\gamma$ are 2LTT terms with type $(\Sigma\,A\,B)[\gamma]$, restricted to object-level contexts. We can still perform pairing and projection with such restricted terms; hence the preservation property.
- Meta-level Π types and universes are preserved in a lax way. For Π, we have
$$\mathbb{R}_{\mathsf{app}} : \mathbb{R}\,(\Pi\,A\,B)\,\gamma \to (\alpha : \mathbb{R}\,A\,\gamma) \to \mathbb{R}\,B\,(\mathbb{R}^{-1}_{\triangleright_1}\,(\gamma,\,\alpha))$$
such that $\mathbb{R}_{\mathsf{app}}\,(\mathbb{R}\,t\,\gamma)\,\alpha = \mathbb{R}\,(\mathsf{app}\,t)\,(\mathbb{R}^{-1}_{\triangleright_1}\,(\gamma,\,\alpha))$. In this case, we can apply a restricted term with a Π-type to a restricted term, but we cannot do lambda-abstraction, because that would require extending the context with a meta-level binder. For $\mathsf{U}_{1,j}$, we have
$$\mathbb{R}_{\mathsf{El}} : \mathbb{R}\,\mathsf{U}_{1,j}\,\gamma \to \widehat{\mathsf{Set}}_j$$
such that $\mathbb{R}_{\mathsf{El}}\,(\mathbb{R}\,t\,\gamma) = \mathbb{R}\,(\mathsf{El}\,t)\,\gamma$. Here, we only have lax preservation simply because $\widehat{\mathsf{Set}}_j$ is much larger than the the set of syntactic 2LTT types, so not every semantic $\widehat{\mathsf{Set}}_j$ has a syntactic representation.

- Meta-level positive (inductive) types are preserved in an oplax way. In the case of natural numbers, we have
$$\mathbb{R}_{\mathbb{N}} : \mathbb{N} \to \mathbb{R}\,\mathsf{Nat}_{1,j}\,\gamma.$$
This is a "serialization" map: from a metatheoretic natural number we compute a numeral as a closed 2LTT term. This works analogously for other inductive types, including parameterized ones. For an example, from a semantic list of restricted terms we would get a syntactic term with list type, containing the same restricted terms.

6.2.2 Action on lifting. We have an isomorphism $\mathbb{R}\,(\Uparrow A)\,\gamma \;\simeq\; \mathsf{Tm}_{0,j}\,(\mathbb{R}\,A\,\gamma)$. This is given by quoting and splicing: we convert between restricted meta-level terms with type $\Uparrow A$ and restricted object-level terms with type A. Hence, we write components of this isomorphism as $\sim -$ and $\langle - \rangle$, as internal analogues of the external operations. With this, we also have $\mathbb{R}\,\langle t\rangle\,\gamma = \langle \mathbb{R}\,t\,\gamma\rangle$ and $\mathbb{R} \sim t\,\gamma = \sim(\mathbb{R}\,t\,\gamma)$.

6.2.3 Action on $- \triangleright_0 -$. We have preservation up to isomorphism:
$$\mathbb{R}_{\triangleright_0} : \mathbb{R}\,(\Gamma \triangleright_0 A) \simeq ((\gamma : \mathbb{R}\,\Gamma) \times \mathsf{Tm}_{0,j}\,(\mathbb{R}\,A\,\gamma))$$
This is because substitutions targeting $\Gamma \triangleright_0 A$ are the same as pairs of substitutions and terms, by the specification of $- \triangleright_0 -$.

6.2.4 Action on object-level types and terms. $\mathbb{R}$ preserves all structure in the object-level fragment of 2LTT. This follows from the $\mathbb{R}$ specification: an external type $A : \mathsf{Ty}_{0,j}\,\Gamma$ is directly internalized as an element of $\mathsf{Ty}_{0,j}$, and the same happens for $t : \mathsf{Tm}_{0,j}\,\Gamma\,A$.

6.3 The Logical Relation

Internally to $\hat{\mathbb{O}}$, we define by induction on the syntax of 2LTT a proof-relevant logical relation interpretation, written as $-^{\approx}$. The induction motives are specified as follows.
$$\begin{array}{ll}
(\Gamma\ :\mathsf{Con})^{\approx} & : \mathbb{E}\,\Gamma \to \mathbb{R}\,\Gamma \to \widehat{\mathsf{Set}}_{\omega} \\
(\sigma : \mathsf{Sub}\,\Gamma\,\Delta)^{\approx} & : \Gamma^{\approx}\,\gamma\,\gamma' \to \Delta^{\approx}\,(\mathbb{E}\,\sigma\,\gamma)\,(\mathbb{R}\,\sigma\,\gamma') \\
(A : \mathsf{Ty}_{1,j}\,\Gamma)^{\approx} & : \Gamma^{\approx}\,\gamma\,\gamma' \to \mathbb{E}\,A\,\gamma \to \mathbb{R}\,A\,\gamma' \to \widehat{\mathsf{Set}}_{j} \\
(t\ : \mathsf{Tm}_{1,j}\,\Gamma\,A)^{\approx} & : (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \to A^{\approx}\,\gamma^{\approx}\,(\mathbb{E}\,t\,\gamma)\,(\mathbb{R}\,t\,\gamma') \\
(A : \mathsf{Ty}_{0,j}\,\Gamma)^{\approx} & : \Gamma^{\approx}\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\,A\,\gamma \urcorner = \mathbb{R}\,A\,\gamma' \\
(t\ : \mathsf{Tm}_{0,j}\,\Gamma\,A)^{\approx} & : \Gamma^{\approx}\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\,t\,\gamma \urcorner = \mathbb{R}\,t\,\gamma'
\end{array}$$
For Con, Sub, and meta-level types and terms, this is a fairly standard logical relation interpretation: contexts are mapped to relations, types to dependent relations, and substitutions and terms respect relations. We will only have modest complications in meta-level type formers, because we will need to sometimes use the lax/oplax preservations properties of $\mathbb{R}$. For object-level types and terms, we get soundness statements: evaluation via $\mathbb{E}$ followed by embedding back to 2LTT is the same as restriction to object-level contexts. We summarize the $-^{\approx}$ interpretation in the following.

6.3.1 Syntactic category and terminal object. Here, we simply have $\mathsf{id}^{\approx}\,\gamma^{\approx} := \gamma^{\approx}$ and $(\sigma \circ \delta)^{\approx}\,\gamma^{\approx} := \sigma^{\approx}\,(\delta^{\approx}\,\gamma^{\approx})$. The terminal object is interpreted as $\bullet^{\approx}\,\gamma\,\gamma' := \top$.

6.3.2 Meta-level family structure. We interpret context extension and type/term substitution as follows. Note the usage of the pattern matching notation on the $\mathbb{R}_{\triangleright_1}^{-1}$ isomorphism.

$$
\begin{aligned}
(\Gamma \triangleright_1 A)^{\approx}\,(\gamma,\,\alpha)\,(\mathbb{R}_{\triangleright_1}^{-1}(\gamma',\,\alpha')) &:= (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \times A^{\approx}\,\gamma^{\approx}\,\alpha\,\alpha' \\
(A[\sigma])^{\approx}\,\gamma^{\approx}\,\alpha\,\alpha' &:= A^{\approx}\,(\sigma^{\approx}\,\gamma^{\approx})\,\alpha\,\alpha' \\
(t[\sigma])^{\approx}\,\gamma^{\approx} &:= t^{\approx}\,(\sigma^{\approx}\,\gamma^{\approx})
\end{aligned}
$$

It is enough to specify the $-^{\approx}$ action on extended substitutions $(\sigma,\ t) : \mathsf{Sub}\,\Gamma\,(\Delta \triangleright A)$, $\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$, $\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$ and $\mathsf{q} : \mathsf{Tm}_{1,j}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$. The category-with-families equations hold evidently.

$$
\begin{aligned}
(\sigma,\,t)^{\approx}\,\gamma^{\approx} &:= (\sigma^{\approx}\,\gamma^{\approx},\,t^{\approx}\,\gamma^{\approx}) \\
\mathsf{p}^{\approx}\,(\gamma^{\approx},\,t^{\approx}) &:= \gamma^{\approx} \\
\mathsf{q}^{\approx}\,(\gamma^{\approx},\,t^{\approx}) &:= t^{\approx}
\end{aligned}
$$

6.3.3 Meta-level natural numbers. First, we have to define a relation:

$$
\begin{aligned}
&(\mathsf{Nat}_{1,j})^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma' \to \mathbb{N} \to \mathbb{R}\,\mathsf{Nat}_{1,j}\,\gamma' \to \widehat{\mathsf{Set}}_j \\
&(\mathsf{Nat}_{1,j})^{\approx}\,\gamma^{\approx}\,n\,n' := (\mathbb{R}_{\mathbb{N}}\,n = n')
\end{aligned}
$$

Note that evaluation sends $\mathsf{Nat}_{1,j}$ to the semantic type of natural numbers, i.e. $\mathbb{E}\,\mathsf{Nat}_{1,j} = \mathbb{N}$. We refer to the serialization map $\mathbb{R}_{\mathbb{N}} : \mathbb{N} \to \mathbb{R}\,\mathsf{Nat}_{1,j}\,\gamma'$ above. In short, $(\mathsf{Nat}_{1,j})^{\approx}$ expresses *canonicity*: n' is canonical precisely if it is of the form $\mathbb{R}_{\mathbb{N}}\,n$ for some n, hence a finite successor of zero.

For $\mathsf{zero}^{\approx}$ and $\mathsf{suc}^{\approx}$, we need to show that serialization respects zero and suc respectively, which is evident.

Let us look at elimination. We need to define the following:

$$
(\mathsf{NatElim}\,P\,z\,s\,n)^{\approx}\,\gamma^{\approx} : P^{\approx}\,(\gamma^{\approx},\,n^{\approx}\,\gamma^{\approx})\,(\mathbb{E}\,(\mathsf{NatElim}\,P\,z\,s\,n)\,\gamma)\,(\mathbb{R}\,(\mathsf{NatElim}\,P\,z\,s\,n)\,\gamma')
$$

Unfolding $\mathbb{E}$, we can further compute this to the following:

$$
\begin{aligned}
&(\mathsf{NatElim}\,P\,z\,s\,n)^{\approx}\,\gamma^{\approx} : \\
&\qquad P^{\approx}\,(\gamma^{\approx},\,n^{\approx}\,\gamma^{\approx})\,(\mathsf{NatElim}\,(\lambda\,n.\,\mathbb{E}\,P\,(\gamma,\,n))\,(\mathbb{E}\,z\,\gamma)\,(\lambda\,n\,pn.\,\mathbb{E}\,s\,((\gamma,\,n),\,pn))(\mathbb{E}\,n\,\gamma)) \\
&\qquad\qquad (\mathbb{R}\,(\mathsf{NatElim}\,P\,z\,s\,n)\,\gamma')
\end{aligned}
$$

In short, we need to show that NatElim preserves relations. Here we switch to the external view temporarily. By Definition **??**, we know that

$$
|\mathbb{R}\,(\mathsf{NatElim}\,P\,z\,s\,n)|\,\gamma' = (\mathsf{NatElim}\,P\,z\,s\,n))[\gamma'].
$$

At the same time, we have $n^{\approx}\,\gamma^{\approx} : \mathbb{R}_{\mathbb{N}}\,(\mathbb{E}\,n\,\gamma) = \mathbb{R}\,n\,\gamma'$, hence we know externally that $|\mathbb{E}\,n|\,\gamma = n[\gamma']$. In other words, $n[\gamma']$ is canonical and is obtained as the serialization of $\mathbb{E}\,n\,\gamma$. Therefore, $(\mathsf{NatElim}\,P\,z\,s\,n))[\gamma']$ is definitionally equal to $|\mathbb{E}\,n|\,\gamma$-many applications of s to z, and we can use $|\mathbb{E}\,n|\,\gamma$-many applications of $s^{\approx}$ to $z^{\approx}$ to witness the preservation. The β-rules for NatElim are also respected by this definition.

6.3.4 Meta-level Σ-types. We define relatedness pointwise. Pairing and projection are interpreted as meta-level pairing and projection.

$$
\begin{aligned}
&(\Sigma\,A\,B)^{\approx}\,\gamma^{\approx} : ((\alpha : \mathbb{E}\,A\,\gamma) \times \mathbb{E}\,B\,(\gamma,\,\alpha)) \to \mathbb{R}\,(\Sigma\,A\,B)\,\gamma' \to \widehat{\mathsf{Set}}_j \\
&(\Sigma\,A\,B)^{\approx}\,\gamma^{\approx}\,(\alpha,\,\beta)\,(\mathbb{R}_{\Sigma}^{-1}(\alpha',\,\beta')) := (\alpha^{\approx} : A^{\approx}\,\gamma^{\approx}\,\alpha\,\alpha') \times B^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx})\,\beta\,\beta'
\end{aligned}
$$

6.3.5 Meta-level Π*-types.* We again use a pointwise definition. Note that we need to use $\mathbb{R}_{\mathsf{app}}$ to apply $t' : \mathbb{R}\,(\Pi\,A\,B)\,\gamma'$ to α'.

$$\begin{aligned}&(\Pi\,A\,B)^{\approx}\,\gamma^{\approx} : ((\alpha : \mathbb{E}\,A\,\gamma) \to \mathbb{E}\,B\,(\gamma,\,\alpha)) \to \mathbb{R}\,(\Pi\,A\,B)\,\gamma' \to \widehat{\mathsf{Set}}_j\\&(\Pi\,A\,B)^{\approx}\,\gamma^{\approx}\,t\,t' := (\alpha : \mathbb{E}\,A\,\gamma)(\alpha' : \mathbb{R}\,A\,\gamma')(\alpha^{\approx} : A^{\approx}\,\gamma^{\approx}\,\alpha\,\alpha') \to B^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx})\,(t\,\alpha)\,(\mathbb{R}_{\mathsf{app}}\,t'\,\alpha')\end{aligned}$$

For abstraction and application, we use a curry-uncurry definition:

$$\begin{aligned}(\mathsf{lam}\,t)^{\approx}\,\gamma^{\approx} &:= \lambda\,\alpha\,\alpha'\,\alpha^{\approx}.\,t^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx})\\(\mathsf{app}\,t)^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx}) &:= t^{\approx}\,\gamma^{\approx}\,\alpha\,\alpha'\,\alpha^{\approx}\end{aligned}$$

6.3.6 Meta-level universes. We interpret $\mathsf{U}_{1,j}$ as a semantic relation space:

$$\begin{aligned}&(\mathsf{U}_{1,j})^{\approx}\,\gamma^{\approx} : \widehat{\mathsf{Set}}_j \to \mathbb{R}\,\mathsf{U}_{1,j}\,\gamma' \to \widehat{\mathsf{Set}}_{j+1}\\&(\mathsf{U}_{1,j})^{\approx}\,\gamma^{\approx}\,t\,t' := t \to \mathbb{R}_{\mathsf{El}}\,t' \to \widehat{\mathsf{Set}}_j\end{aligned}$$

Note that we have

$$\begin{aligned}(\mathsf{El}\,t)^{\approx} &: (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \to \mathbb{E}\,t\,\gamma \to \mathbb{R}\,(\mathsf{El}\,t)\,\gamma' \to \widehat{\mathsf{Set}}_j\\(\mathsf{Code}\,t)^{\approx} &: (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \to \mathbb{E}\,t\,\gamma \to \mathbb{R}\,(\mathsf{El}\,t)\,\gamma' \to \widehat{\mathsf{Set}}_j.\end{aligned}$$

The types coincide because of the equation $\mathbb{R}\,(\mathsf{El}\,t)\,\gamma' = \mathbb{R}_{\mathsf{El}}\,(\mathbb{R}\,t\,\gamma')$. Therefore we can interpret El and Code as identity maps, as $(\mathsf{El}\,t)^{\approx} := t^{\approx}$ and $(\mathsf{Code}\,t)^{\approx} := t^{\approx}$.

6.3.7 Object-level family structure. We interpret extended contexts as follows.

$$(\Gamma \triangleright_0 A)^{\approx}\,(\gamma,\,\alpha)\,(\mathbb{R}^{-1}_{\triangleright_1}\,(\gamma',\,\alpha')) := (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \times (\ulcorner\alpha\urcorner = \alpha')$$

Note that $\alpha : \mathsf{Tm}_{\mathbb{O}j}\,(\mathbb{E}\,A\,\gamma)$, so $\ulcorner\alpha\urcorner : \mathsf{Tm}_{0,j}\,\ulcorner\mathbb{E}\,A\,\gamma\urcorner$, but since $A^{\approx}\,\gamma^{\approx} : \ulcorner\mathbb{E}\,A\,\gamma\urcorner = \mathbb{R}\,A\,\gamma'$, we also have $\ulcorner\alpha\urcorner : \mathsf{Tm}_{0,j}\,(\mathbb{R}\,A\,\gamma')$. Thus, the equation $\ulcorner\alpha\urcorner = \alpha'$ is well-typed. For type substitution, we need

$$(A[\sigma])^{\approx}\,\gamma^{\approx} : \ulcorner\mathbb{E}\,(A[\sigma])\,\gamma\urcorner = \mathbb{R}\,(A[\sigma])\,\gamma'.$$

The goal type computes to $\ulcorner\mathbb{E}\,A\,(\mathbb{E}\,\sigma\,\gamma)\urcorner = \mathbb{R}\,A\,(\mathbb{R}\,\sigma\,\gamma')$. This is obtained directly from $A^{\approx}\,(\sigma^{\approx}\,\gamma^{\approx})$. Similarly, we have

$$\begin{aligned}(t[\sigma])^{\approx}\,\gamma^{\approx} &:= t^{\approx}\,(\sigma^{\approx}\,\gamma^{\approx})\\(\sigma,\,t)^{\approx}\,\gamma^{\approx} &:= (\sigma^{\approx}\,\gamma^{\approx},\,t^{\approx}\,\gamma^{\approx})\\\mathsf{p}^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx}) &:= \gamma^{\approx}\\\mathsf{q}^{\approx}\,(\gamma^{\approx},\,\alpha^{\approx}) &:= \alpha^{\approx}.\end{aligned}$$

6.3.8 Lifting structure.

$$\begin{aligned}&(\Uparrow A)^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma' \to \mathsf{Tm}_{\mathbb{O}j}\,(\mathbb{E}\,A\,\gamma) \to \mathbb{R}\,(\Uparrow A)\,\gamma' \to \widehat{\mathsf{Set}}_j\\&(\Uparrow A)^{\approx}\,\gamma^{\approx}\,t\,t' := (\ulcorner t\urcorner = {\sim}t')\end{aligned}$$

This is well-typed by $A^{\approx}\,\gamma^{\approx} : \ulcorner\mathbb{E}\,A\,\gamma\urcorner = \mathbb{R}\,A\,\gamma'$, which implies that $\ulcorner t\urcorner : \mathsf{Tm}_{\mathbb{O}j}\,(\mathbb{R}\,A\,\gamma')$. For $\langle t\rangle$, we need

$$\langle t\rangle^{\approx}\,\gamma^{\approx} : \ulcorner\mathbb{E}\,\langle t\rangle\,\gamma\urcorner = {\sim}(\mathbb{R}\,\langle t\rangle\,\gamma').$$

The goal type can be further computed to $\ulcorner\mathbb{E}\,t\,\gamma\urcorner = \mathbb{R}\,t\,\gamma'$, which we prove by $t^{\approx}\,\gamma^{\approx}$. For splicing, we need

$$({\sim}t)^{\approx}\,\gamma^{\approx} : \ulcorner\mathbb{E}\,{\sim}t\,\gamma\urcorner = \mathbb{R}\,{\sim}t\,\gamma'$$

where the goal type computes to $\ulcorner F\,t\,\gamma\urcorner = {\sim}(\mathbb{R}\,t\,\gamma')$, but this again follows directly from $t^{\approx}\,\gamma^{\approx}$.

6.3.9 Object-level type formers. Lastly, object-level type formers are straightforward. For types, we need $\ulcorner \mathbb{E}\, A\, \gamma \urcorner = \mathbb{R}\, A\, \gamma'$, and for terms, we need $\ulcorner \mathbb{E}\, A\, \gamma \urcorner = \mathbb{R}\, A\, \gamma'$. Note that $\mathbb{E}$ and $\ulcorner - \urcorner$ preserve all structure, and $\mathbb{R}$ preserves all structure on object-level types and terms. Hence, each object-level A and t case in $-^{\approx}$ trivially follows from induction hypotheses.

This concludes the definition of the $-^{\approx}$ interpretation.

6.4 Soundness

Definition 6.2. First, we introduce shorthands for external operations that can be extracted from $-^{\approx}$.

- For $\Gamma : \mathsf{Con}$, we get $|\Gamma^{\approx}| : \{\Delta : \mathsf{Con}_{\mathbb{O}}\} \to |\mathbb{E}\,\Gamma|\,\Delta \to \mathsf{Sub}\,\ulcorner\Delta\urcorner\,\Gamma \to \mathsf{Set}_j$.
- For $A : \mathsf{Ty}_{0,j}\,\Gamma$, we get $|A^{\approx}| : |\Gamma^{\approx}|\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\, A\, \gamma \urcorner = A[\gamma']$.
- For $t : \mathsf{Tm}_{0,j}\,\Gamma\, A$, we get $|t^{\approx}| : |\Gamma^{\approx}|\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\, t\, \gamma \urcorner = t[\gamma']$.

Since $-^{\approx}$ was defined in $\hat{\mathbb{O}}$, we also know that the above are all stable under object-theoretic substitution.

Theorem 6.3 (Soundness for generic contexts). *For each* $\Gamma : \mathsf{Con}_{\mathbb{O}}$, *we have* $\Gamma^{P^{\approx}} : |\Gamma^{\approx}|\,\Gamma^{P}\,\mathsf{id}$.

Proof. We define $-^{P^{\approx}}$ by induction on object-theoretic contexts. $\bullet^{P^{\approx}} : \top$ is defined trivially as tt. For $(\Gamma \triangleright A)^{P^{\approx}}$, we need $(\gamma^{\approx} : |\Gamma|^{\approx}\,(\Gamma^{P}[\mathsf{p}])\,\mathsf{p}) \times (\ulcorner \mathsf{q} \urcorner = \mathsf{q})$. We get $\Gamma^{P^{\approx}} : |\Gamma^{\approx}|\,\Gamma^{P}\,\mathsf{id}$. Because of the naturality of $|\Gamma^{\approx}|$, this can be weakened to $\Gamma^{P^{\approx}}[\mathsf{p}] : |\Gamma^{\approx}|\,(\Gamma^{P}[\mathsf{p}])\,\mathsf{p}$. Also, $\ulcorner \mathsf{q} \urcorner = \mathsf{q}$ holds immediately. □

Theorem 6.4 (Soundness of staging). *The open staging algorithm from Definition* **??** *is sound.*

Proof.

- For $A : \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$, we have that $\ulcorner \mathsf{Stage}\, A \urcorner = \ulcorner |\mathbb{E}\, A|\, \Gamma^{P} \urcorner$ by the definition of Stage, and moreover we have $|A^{\approx}|\,\Gamma^{P^{\approx}} : \ulcorner |\mathbb{E}\, A|\, \Gamma^{P} \urcorner = A[\mathsf{id}]$, hence $\ulcorner \mathsf{Stage}\, A \urcorner = A$.
- For $t : \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\, A$, using $|t^{\approx}|\,\Gamma^{P^{\approx}} : \ulcorner |\mathbb{E}\, t|\, \Gamma^{P} \urcorner = t[\mathsf{id}]$, we likewise have $\ulcorner \mathsf{Stage}\, t \urcorner = \ulcorner |\mathbb{E}\, t|\, \Gamma^{P} \urcorner = t[\mathsf{id}] = t$.

□

Corollary 6.5 (Conservativity of 2LTT). *From the soundness and stability of staging, we get that* $\ulcorner - \urcorner$ *is bijective on types and terms, i.e.* $\mathsf{Ty}_{\mathbb{O}j}\,\Gamma \simeq \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$ *and* $\mathsf{Tm}_{\mathbb{O}j}\,\Gamma\, A \simeq \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,\ulcorner A \urcorner$.

REFERENCES

[1] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2019. Two-Level Type Theory and Applications. *ArXiv e-prints* (may 2019). http://arxiv.org/abs/1705.03307

[2] Paolo Capriotti. 2017. Models of type theory with strict equality. *arXiv preprint arXiv:1702.04912* (2017).

[3] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for Type Theory. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:19. https://doi.org/10.4230/LIPIcs.FSCD.2019.25

[4] Vladimir Voevodsky. 2013. A simple type system with two identity types. (2013). Unpublished note.