Staged Compilation With Two-Level Type Theory

ANONYMOUS AUTHOR(S)

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-behaved 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 prove its soundness and stability. 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. A sound and stable staging algorithm constitutes a proof of strong conservativity of 2LLT over the object theory. To our knowledge, this is the first description of 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:
Anonymous Author(s). 2022. Staged Compilation With Two-Level Type Theory. 1, 1 (June 2022), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 discipline, 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 [6], building on ideas from Vladimir Voevodsky [45]. 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 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.

XXXX-XXXX/2022/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Overview & Contributions

- In Section 2 we present an informal syntax of two-level type theory, a dependent type theory with support for two-stage compilation. 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 Section 3, we specify the formal notion of models and syntax for the variant of 2LTT used in this paper. We mostly work with a high-level algebraic presentation, where the syntax is quotiented by conversion. However, we also explain how to extract functions operating on non-quotiented syntax, by interpreting type-theoretic constructions in a setoid model.
- In Section 4, we review the standard presheaf model of 2LTT [6, Section 2.5.3], which lies over the syntactic category of the object theory. We show that evaluation in this model yields a staging algorithm for closed types and terms. We then extend staging to open types and terms, which may depend on object-level contexts. We show stability of staging, which means that staging is surjective up to conversion, and we also show that staging strictly preserves all type and term formers. Finally, we discuss efficiency and potential practical implementations of staging.
- In Section 5 we show soundness of staging, which roughly means that the output of staging is convertible to its input. Staging together with its stability and soundness 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 strengthens the weak notion of conservativity shown in [10] and [6].
- In Section 6, we discuss possible semantic interpretations of intensional code analysis, i.e. the ability to look into the structure of object-level code.
- We provide a small standalone implementation of elaboration and staging for a two-level type theory, with code samples that expand on the paper.
- To our knowledge, this work is the first formalization and implementation of staged compilation in the presence of full dependent types, with universes and large elimination. In particular, 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 [20]. We focus on examples and informal explanation here; the formal details will be presented in Section 3.

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

2.1.1 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, as 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 of A are likewise computed away.

2.1.2 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 variations of the object-level languages in Section 2.4.

2.1.3 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 $(\sim f)\ x$. We borrow this notation from MetaML [44].

- 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, consider the usual identity function at each stage:

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

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:

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

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 $\mathsf{nil}_0 : (A : \mathsf{U}_0) \to \mathsf{List}_0\, A$, $\mathsf{cons}_0 : (A : \mathsf{U}_0) \to A \to \mathsf{List}_0\, A \to \mathsf{List}_0\, A$ 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 mapping function which inlines its function argument:

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

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:

$$\begin{aligned}
&\sim(map\, \langle \mathsf{Nat}_0 \rangle\, \langle \mathsf{Nat}_0 \rangle\, (\lambda\, n.\, \langle \sim n +_0 10 \rangle)\, \langle ns \rangle) \\
=\ &\sim\langle \mathsf{foldr}_0\, \sim\langle \mathsf{Nat}_0 \rangle\, \sim\langle \mathsf{List}_0\, \sim\langle Nat_0 \rangle \rangle\, (\lambda\, a\, bs.\, \mathsf{cons}_0\, \sim\langle \mathsf{Nat}_0 \rangle\, \sim\langle \sim\langle a \rangle +_0 10 \rangle\, bs)\, (\mathsf{nil}_0\, \sim\langle \mathsf{Nat}_0 \rangle)\, \sim\langle ns \rangle \rangle \\
=\ &\mathsf{foldr}_0\, \mathsf{Nat}_0\, (\mathsf{List}_0\, \mathsf{Nat}_0)\ \ (\lambda\, a\, bs.\, \mathsf{cons}_0\, (a +_0 10)\, bs)\, (\mathsf{nil}_0\, \mathsf{Nat}_0)\, ns
\end{aligned}$$

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 $- *_0 -$.

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

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.

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

With this definition, $\sim(\mathit{Vec}\, 3\, \langle \mathsf{Nat}_0\rangle)$ 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 common 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:

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

We include examples of more sophisticated let-insertion in the supplementary code.

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; the full version can be found in the supplementary code. First, we inductively define types, contexts 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* [28]: 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 object language. This is similar to the partial evaluator presented in Idris 1 [9]. 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 lifting on type formers. While lifting does not have definitional computation rules, it does preserve negative type formers up to definitional isomorphism [6, Section 2.3]:

$$\Uparrow((x : A) \to B\,x) \simeq ((x : \Uparrow A) \to \Uparrow (B \sim x))$$
$$\Uparrow((x : A) \times B\,x) \simeq ((x : \Uparrow A) \times \Uparrow (B \sim x))$$
$$\Uparrow \top_0 \simeq \top_1$$

For function types, the preservation maps are the following:

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

With this, we have that $pres_{\to}\ (pres^{-1}_{\to}\ f)$ 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 [28, Chapter 12]. Note that the output of $pres_{\to}$ uses a meta-level λ, while going the other way introduces a runtime binder. Meta-level function types 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 $id_{\Uparrow}$:

$$id_{\Uparrow} : (A : \Uparrow \mathsf{U}_0) \to \Uparrow(\sim A \to \sim A)$$
$$id_{\Uparrow} := \lambda\, A.\ \langle \lambda\, x.\, x \rangle$$

This can be used at the runtime stage as $\sim(id_{\Uparrow}\ \langle \mathsf{Bool}_0 \rangle)\ \mathsf{true}_0$, 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 do not get full preservation, only maps in one direction. 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" [41, 44]. 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. We will briefly discuss ways to lift this restriction in Section 6.

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* [6, Section 3.4]: this means that for each B, $A \to \Uparrow B$ is equivalent to $\Uparrow C$ for some C. This is the 2LTT formalization of the so-called "trick" in partial evaluation, which improves binding times by η-expanding functions out of finite sums [18].

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 [23], which is employed in GHC Haskell. Starting from $\Uparrow(\mathsf{List}_0\ A)$, we use Böhm-Berarducci encoding [8] 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* [16], 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; a small example can be found in the supplementary code. 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 possible to get pessimized code via failed fusion.

2.3.2 Inferring staging operations. We can extract a coercive subtyping system from the staging operations, which can be used for inference, and in particular for automatically transporting definitions along preservation isomorphisms. One choice is to have $A \leq \Uparrow A$, $\Uparrow A \leq A$, a contravariant-covariant rule for functions and a covariant rule for Σ. During bidirectional elaboration, when we need to compare an inferred and an expected type, we can insert coercions. 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 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.

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, 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 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. Additionally, we will see in Section 6 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, so 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 value of 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; usually we would mark types are erased. 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 $r : \mathsf{Rep}$, $r' : \mathsf{Rep}$, $A : \mathsf{U}_0\, r$ 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 some amount of 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.

3 FORMAL SETUP

In this section we describe our approach to formalizing staging and also the *extraction* of algorithms from the formalization. There is a tension here:

- On one hand, we would like to do formal work at a higher level of abstraction, in order to avoid a deluge of technical details (for which dependent type theories are infamous).
- On the other hand, at some point we need to talk about lower-level operational aspects and extracted algorithms.

In the following, we describe our approach to interfacing between the different levels of abstraction.

3.1 The Algebraic Setting

When defining staging and proving its soundness and stability, we work internally in a constructive type theory, using types and terms that are quotiented by conversion. In this setting, every construction respects syntactic conversion, which is very convenient technically. Here, we use standard definitions and techniques from the categorical and algebraic metatheory of type theories, and reuse prior results which rely on the same style of formalization. Concretely, we use *categories with families* [11] which is a commonly used specification of an *explicit substitution calculus* of types and terms.

In the rest of the paper we will mostly work in this setting, and we will explicitly note when we switch to a different metatheory. We summarize the features of this metatheory in the following.

3.1.1 Metatheory. The metatheory is an intensional type theory with a cumulative universe hierarchy Set_i, where i is ordinal number such that $i \leq \omega + 1$. The transfinite i levels are a convenience feature for modeling typing contexts, where sometimes we will need universes large enough to fit types at arbitrary finite levels. We also assume uniqueness of identity proofs and function extensionality. We will omit transports along identity proofs in the paper, for the sake of readability. The missing transports can be in principle recovered due to Hofmann's conservativity result [25]; see also Winterhalter et al. [46]. We assume Π, Σ, $\top$ and $\mathbb{N}$ types in all universes, and reuse Notation 3. We also assume certain quotient inductive-inductive types, to represent the syntax of 2LTT and the object theory. We will detail these shortly in Sections 3.3 and 3.4.

3.2 Algorithm Extraction

In the algebraic setting we can *only* talk about properties which respect syntactic conversion. However, the main motivation of staged compilation is to improve performance and code abstraction, but these notions do *not* respect conversion. For instance, rewriting programs along β-conversion can improve performance. Hence, at some point we may want to "escape" from quotients.

Fortunately, our specified meta type theory can be interpreted in *setoids*, as most recently explained by Pujet and Tabareau [40]. A setoid is a set together with an equivalence relation on it. We interpret closed types as setoids, and dependent types as setoid fibrations. For inductive base types such Bool or Nat, the equivalence relation is interpreted as identity. Functions are interpreted as equivalence-preserving functions, and two functions are equivalent if they are pointwise equivalent. A quotient type A/R is interpreted by extending the semantic notion of equivalence for A with the given R relation.

Definition 3.1. Assume a closed function $f : A \to B$ defined in the mentioned type theory. The **extraction** of f is the underlying function of f's interpretation in the setoid model.

In particular, from $f : A/R \to A/R$ we can extract a function which acts on the underlying set of A's interpretation. Note that an extracted function can be only viewed as an algorithm if the setoid interpretation is itself defined in a constructive theory, but that is the case in the cited work [40].

In other words, we can view the usage of constructive quotients as a convenient shorthand for working with explicit equivalence relations (an *internal language* of setoids). When we reason about the syntax of a type theory, this provides a bridge between the categorical/algebraic style and the traditional style that uses separate equivalence relations. It makes sense to work with quotients internally to a type theory whenever we only want to talk about constructions and properties which respect conversion. However, at any point we have the option to switch to an external perspective and look at underlying sets and the underlying maps.

3.3 Models and Syntax of 2LTT

We specify the notion of model for 2LTT in this section. The syntax of 2LTT will be defined as the initial model, as a quotient inductive-inductive type, following [4]. This yields an explicit substitution calculus, quotiented by definitional equality. A major advantage of this representation is that it *only* includes well-typed syntax, so we never have to talk about "raw" types and terms.

First, we define the structural scaffolding of 2LTT without type formers, mostly following [6]. The specification is agnostic about the sizes of sets involved, i.e. we can model underlying sorts using any metatheoretic Set_j.

Definition 3.2. A model of **basic 2LTT** consists of the following.

- A category $\mathbb{C}$ with a terminal object. We denote the set of objects as $\mathrm{Con}_{\mathbb{C}} : \mathrm{Set}$ and use capital Greek letters starting from Γ to refer to objects. The set of morphisms is $\mathrm{Sub}_{\mathbb{C}} : \mathrm{Con}_{\mathbb{C}} \to$

$\mathsf{Con}_{\mathbb{C}} \to \mathsf{Set}$, 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. In the initial model, an element of Con is a *typing context*, an element of Sub is a *parallel substitution*, $\bullet$ is the *empty typing context*, and ϵ is the empty list viewed as a parallel substitution. The identity substitution $\mathsf{id} : \mathsf{Sub}\,\Gamma\,\Gamma$ maps each variable to itself, and $\sigma \circ \delta$ can be computed by substituting each term in σ (which is a list of terms) with δ.

- For each $i \in \{0, 1\}$ and $j \in \mathbb{N}$, we have $\mathsf{Ty}_{i,j} : \mathsf{Con} \to \mathsf{Set}$ and $\mathsf{Tm}_{i,j} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}_{i,j}\,\Gamma \to \mathsf{Set}$, as sets of types and terms. Both types and terms can be substituted:

$$-[-] : \mathsf{Ty}_{i,j}\,\Delta \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Ty}_{i,j}\,\Gamma$$
$$-[-] : \mathsf{Tm}_{i,j}\,\Delta\,A \to (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}_{i,j}\,\Gamma\,(A[\sigma])$$

  Additionally, we have $A[\mathsf{id}] = A$ and $A[\sigma \circ \delta] = A[\sigma][\delta]$, and we have the same equations for term substitution as well.
  We also have a *comprehension structure*: for each $\Gamma : \mathsf{Con}$ and $A : \mathsf{Ty}_{i,j}\,\Gamma$, we have the extended context $\Gamma \triangleright A : \mathsf{Con}$ such that there is a natural isomorphism $\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \simeq (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \times \mathsf{Tm}_{i,j}\,\Gamma\,(A[\sigma])$. We will sometimes use $-\triangleright_0-$ and $-\triangleright_1-$ to disambiguate object-level and meta-level context extensions.
- 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 $\sim -$.

The following notions are derivable:

- 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 initial model, this extends a parallel substitution with an extra term, thus justifying the view of substitutions as lists of terms. 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 sometimes 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 3.3. A **model of 2LTT** is a model of basic 2LTT which supports certain type formers. For the sake of brevity, we specify here only a small collection of type formers. However, we will argue later that our results extend to more general notions of inductive types. We specify type formers in the following. We omit substitution rules for type and term formers; all of these can be specified in an evident structural way, e.g. as in $(\Uparrow A)[\sigma] = \Uparrow(A[\sigma])$.

- *Universes.* For each i and j, we have a Coquand-style universe [15] 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:

$$\mathsf{Tm}_{i,j}\,\Gamma\,(\Sigma\,(x : A)\,B) \simeq ((t : \mathsf{Tm}_{i,j}\,\Gamma\,A) \times \mathsf{Tm}_{i,j}\,\Gamma\,(B[x \mapsto t])$$

  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 abstraction and application

$$\mathsf{lam} : \mathsf{Tm}_{i,j}\,(\Gamma \triangleright (x : A))\,B \to \mathsf{Tm}_{i,j}\,\Gamma\,(\Pi\,(x : A)\,B)$$
$$\mathsf{app} : \mathsf{Tm}_{i,j}\,\Gamma\,(\Pi\,(x : A)\,B) \to \mathsf{Tm}_{i,j}\,(\Gamma \triangleright (x : A))\,B$$

  such that lam and app are inverses. This specification of app is equivalent to the "traditional" specification:

$$\mathsf{app}' : (t : \mathsf{Tm}_{i,j}\,\Gamma\,(\Pi\,(x : A)\,B)) \to (u : \mathsf{Tm}_{i,j}\,\Gamma\,A) \to \mathsf{Tm}_{i,j}\,\Gamma\,(B[x \mapsto u]).$$

  The traditional version is definable as $(\mathsf{app}\,t)[\mathsf{id},\, u]$. We use the app-lam isomorphism because it is formally more convenient.
- *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.

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

  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 3.4. The **syntax of 2LTT** is the initial model of 2LTT, defined as a quotient inductive-inductive type [31]. The signature for this type includes four sorts (Con, Sub, Ty, Tm), constructors for contexts, explicit substitutions and type/term formers, and equality constructors for all specified equations. The syntax comes with an *induction principle*, from which we can also derive the *recursion principle* as a non-dependent special case, witnessing the initiality of the syntax.

3.3.1 Comparison to Annekov et al. Comparing our models to the primary reference on 2LTT [6], 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 ibid., "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. Ibid. also considers a range of additional strengthenings and extensions of 2LTT [6, Section 2.4], most of which are useful in synthetic homotopy theory. We do not assume any of these, and stick to the most basic formulation of 2LTT.

3.3.2 Elaborating informal syntax. The justification for the usage of the informal syntax in Section 2 is *elaboration*. Elaboration translates surface notation to the formal syntax (often termed "core syntax" in the implementation of dependently typed programming languages). Elaboration is partial: it may throw error on invalid input. Elaboration also requires *decidability of conversion* for the formal syntax, since checking well-typing requires comparing types for conversion, which in turn requires comparing terms. We do not detail elaboration in this paper, we only make some observations.

- Decidability of conversion can be shown through normalization. Since 2LTT is a minor variation on ordinary MLTT, we expect that existing proofs for MLTT can be adapted without issue [3, 15].
- We need to elaborate the surface-level Russell-style universe notation to Coquand-style universes with explicit El and Code annotations. It is likely that this can be achieved with the use of bidirectional elaboration [22], using two directions (checking and inference) both for types and terms, and inserting El and Code when switching between types and terms.
- The formal complexity of elaboration varies wildly, depending on how explicit the surface syntax is. For example, Agda-style implicit arguments require metavariables and unification, which dramatically increases the difficulty of formalization.

3.3.3 The setoid interpretation of the syntax. We need to show that the 2LTT syntax, as a particular inductive type, can be modeled in setoids. Unfortunately, there is no prior work which presents the setoid interpretation of quotient types in the generality of quotient induction-induction. We sketch it for our specific use case, but we argue that the interpretation is fairly mechanical and could be adapted to any quotient inductive-inductive type. We focus on contexts and types here.

We mutually inductively define $\mathsf{Con} : \mathsf{Set}$, $\mathsf{Con}^\sim : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Prop}$, $\mathsf{Ty}_{i,j} : \mathsf{Con} \to \mathsf{Set}$ and $\mathsf{Ty}^\sim : \mathsf{Ty}_{i,j}\,\Gamma \to \mathsf{Ty}_{i,j}\,\Delta \to \mathsf{Con}^\sim\,\Gamma\,\Delta \to \mathsf{Prop}$, where $\mathsf{Con}^\sim$ and $\mathsf{Ty}^\sim$ are proof-irrelevant relations. Note that $\mathsf{Ty}^\sim$ is *heterogeneous*, since we can relate types in different (but provably convertible) contexts.

We specify $\bullet$ and $- \triangleright -$ in Con. To $\mathsf{Con}^\sim$ we add that $\bullet$ and $- \triangleright -$ are congruences, and also that $\mathsf{Con}^\sim$ is reflexive, symmetric and transitive.

We add all type formation rules to $\mathsf{Ty}_{i,j}$, and add all quotient equations to $\mathsf{Ty}^\sim$ as homogeneously indexed rules. For example, the substitution rule for Nat_0 is specified as $\mathsf{Nat}_0[] : \mathsf{Ty}^\sim\,(\mathsf{Nat}_0[\sigma])\,\mathsf{Nat}_0\,\mathsf{refl}^\sim$, where $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$ and $\mathsf{refl}^\sim : \mathsf{Con}^\sim\,\Gamma\,\Gamma$. We also add all congruences and reflexivity, symmetry and transitivity to $\mathsf{Ty}^\sim$.

We add a *coercion* rule to types, as $\mathsf{coerce} : \mathsf{Con}^\sim\,\Gamma\,\Delta \to \mathsf{Ty}_{i,j}\,\Gamma \to \mathsf{Ty}_{i,j}\,\Delta$. This is probably familiar from the traditional specifications of type theories that use conversion relations. However, we additionally need a *coherence rule* which witnesses $\mathsf{Ty}^\sim\,A\,(\mathsf{coerce}\,p\,A)\,p$ for each $A : \mathsf{Ty}_{i,j}\,\Gamma$ and $p : \mathsf{Con}^\sim\,\Gamma\,\Delta$. In short, coherence says that coercion preserves conversion.

The concise description of this setup is that $(\mathsf{Con}, \mathsf{Con}^\sim)$ is a setoid, and $(\mathsf{Ty}_{i,j}, \mathsf{Ty}^\sim)$ is a *setoid fibration* over $(\mathsf{Con}, \mathsf{Con}^\sim)$, for each i and j. This notion of setoid fibration is equivalent to the ordinary notion of fibration, when setoids are viewed as categories with proof-irrelevant invertible morphisms. This is appropriate, since types in general are interpreted as setoid fibrations in the setoid model of a type theory.

This scheme generalizes to substitutions and terms in a mechanical way. Sub is indexed over two contexts, so it is interpreted as a fibration over the setoid product of Con and Con. Tm is indexed over $(\Gamma : \mathsf{Con}) \times \mathsf{Ty}_{i,j}\,\Gamma$, so it is interpreted as a fibration over a setoid Σ-type. In every conversion relation we take the congruence closure of the specified quotient equations. Finally, we can use the induction principle for the thus specified sets and relations to interpret the internal elimination principle for the 2LTT syntax.

3.4 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 3.5. 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 3.6. Like before, the **syntax of the object theory** is the initial model, given as a quotient inductive-inductive type, and it can be also interpreted in setoids.

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 3.7. We recursively define the **embedding** of object syntax into 2LTT syntax, which preserves all structure strictly and consists of the following functions:

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

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:

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

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. The following properties are of interest when considering a staging algorithm:

(1) *Soundness:* $\ulcorner \mathsf{Stage}\, A \urcorner = A$ and $\ulcorner \mathsf{Stage}\, t \urcorner = t$.
(2) *Stability:* $\mathsf{Stage}\, \ulcorner A \urcorner = A$ and $\mathsf{Stage}\, \ulcorner t \urcorner = t$.
(3) *Strictness:* the function extracted from Stage preserves all type and term formers strictly (not just up to conversion).

Soundness and stability together state that embedding is a bijection on types and terms (up to conversion). This is a statement of *conservativity* of 2LTT over the object theory. In [6] a significantly weaker conservativity theorem is shown, which only expresses that there exists a function from $\mathsf{Tm}_{0,j}\, \ulcorner\Gamma\urcorner\, \ulcorner A\urcorner$ to $\mathsf{Tm}_{\mathbb{O}j}\, \Gamma\, A$.

Strictness tells us that staging does not perform any computation in the object theory; e.g. it does not perform β-reduction or inline any let-definition. This is practically quite important; for instance, *full normalization of 2LTT* is a sound and stable staging algorithm, but it is not strict, and it is practically not useful, since it does not provide any control over inlining and β-reduction in code generation.

Note that staging necessarily preserves type and term formers up to conversion, since it is defined on quotiented types and terms. More generally, staging respects conversion because every function must respect conversion in the meta type theory.

4.1 The Presheaf Model

We review the presheaf model described in [6, Section 2.5.3]. This will directly yield a closed staging algorithm, by recursive evaluation of 2LTT types and terms in the model.

We give a higher-level overview first. Presheaves can be viewed as a generalization of sets, as "variable sets" which may vary over a base category. A set-based model of 2LTT yields a standard interpreter, where every syntactic object is mapped to the obvious semantic counterpart. The presheaf model over the category of contexts and substitutions in $\mathbb{O}$ generalizes this to an interpreter where semantic values may depend on object-theoretic contexts. This allows us to handle object-theoretic *types and terms* during interpretation, since these necessarily depend on their contexts.

In the presheaf model, every semantic construction must be *stable* under object-level substitution, or using alternative terminology, must be *natural* with respect to substitution. Semantic values can be viewed as runtime objects during interpretation, and naturality means that object-theoretic substitutions can be *applied* to such runtime objects, acting on embedded object-theoretic types and terms. Stability underlies the core trade-off in staging:

- On one hand, every syntactic rule and construction in 2LTT must be stable, which restricts the range of metaprograms that can be written in 2LTT. For example, we cannot make decisions based on sizes of the object-theoretic scopes, since these sizes may be changed by substitution.
- On the other hand, if only stable constructions are possible, we never have to prove stability, or even explicitly mention object-theoretic contexts, substitutions and variables. The tedious details of deep embeddings can be replaced with a handful of staging operations and typing rules.

In principle, the base category in a 2LTT can be any category with finite products; it is not required that it is a category of contexts and substitutions. If the base category has more structure, we get more interesting object theories, but at the same time stability becomes more onerous, because we must be stable under *more* morphisms. If the base category is simpler, with fewer morphisms, then the requirement of stability is less restrictive. In Section 6, we will look at a concrete example for this.

We proceed to summarize the key components of the model in a bit more detail. We denote the components of the model by putting hats on 2LTT components, e.g. as in $\widehat{\mathsf{Con}}$.

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. [27, Chapter 1.2]. 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 $-[-] : |A|\,\gamma \to (f : \mathrm{Sub}_{\mathbb{O}}\, a\, b) \to |A|\,(\gamma[f])$, such that $\alpha[\mathrm{id}] = \alpha$ 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. $\widehat{\mathbf{Tm}}_{1,j}\,\boldsymbol{\Gamma}\,\boldsymbol{A} : \mathbf{Set}_{\omega}$ 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 $\boldsymbol{\Gamma}\,\widehat{\triangleright}\,\boldsymbol{A}$ pointwise by $|\Gamma\,\widehat{\triangleright}\,A|\,a := (\gamma : |\Gamma|\,a) \times |A|\,\gamma$ 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 expositions in previous literature, see [27, Chapter 1.2] and [26, Section 4.1].

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. $\widehat{\mathbf{Ty}}_{\mathbf{0},j}\,\boldsymbol{\Gamma} : \mathbf{Set}_{\omega}$ is defined as $\widehat{\mathsf{Sub}}\,\Gamma\,\mathsf{Ty}_{\mathbb{O}j}$, and $\widehat{\mathbf{Tm}}_{\mathbf{0},j}\,\boldsymbol{\Gamma}\,\boldsymbol{A} : \mathbf{Set}_{\omega}$ 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 $|A| : \{a : \mathrm{Con}_{\mathbb{O}}\} \to |\Gamma|\,a \to \mathsf{Ty}_{\mathbb{O}j}\,a$. 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 $\boldsymbol{\Gamma}\,\widehat{\triangleright}\,\boldsymbol{A}$ as $|\Gamma\,\widehat{\triangleright}\,A|\,a := (\gamma : |\Gamma|\,a) \times \mathsf{Tm}_{\mathbb{O}j}\,a\,(|A|\,\gamma)$ 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 [6], we can model all type formers in $\widehat{\mathsf{Ty}}_{0,j}$. Intuitively, this 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. $\widehat{\Uparrow}\boldsymbol{A} : \widehat{\mathbf{Ty}}_{\mathbf{1},j}\,\boldsymbol{\Gamma}$ 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 morphism of 2LTT models, between the syntax of 2LTT and $\hat{\mathbb{O}}$. It is defined by recursion on the syntax (that is, appealing to the initiality of the syntax), and it strictly preserves all structure.

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

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

Note that $\widehat{\bullet}$ is defined as the terminal presheaf, which is constantly $\top$. Also, $|\mathbb{E}\,A|\{\bullet\} : |\mathbb{E}\,\bullet|\,\bullet \to \mathsf{Ty}_{\mathbb{O}j}\,\bullet$, therefore $|\mathbb{E}\,A|\{\bullet\} : \top \to \mathsf{Ty}_{\mathbb{O}j}\,\bullet$ 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 obtain object-level type. Such "generic" semantic environments should be possible to construct, because elements of $|\ulcorner\Gamma\urcorner|\,\Gamma$ are essentially lists of object-level terms in Γ, by Definitions 4.8 and 4.7, 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 Open Staging, Stability and Strictness

We define a family of functions $-^P$ by induction on object syntax, such that the interpretation of a context yields a generic semantic environment. The induction motives are as follows.

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

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.

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

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. □

Theorem 4.16. *Open staging is strict.*

Proof. This is evident from the definition of the presheaf model, where the action of each type and term former in $\widehat{\mathsf{Ty}}_{0,j}$ and $\widehat{\mathsf{Tm}}_{0,j}\,\Gamma\,A$ is defined using the corresponding type or term former in the object syntax. □

4.4 Implementation and Efficiency

We discuss now the operational behavior of the extracted staging algorithm and look at potential adjustments and optimizations that make it more efficient or more convenient to implement. Recall the types of the component functions in $\mathbb{E}$:

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

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

- $|\mathsf{Nat}_1|\,\_$ is simply the set of natural numbers $\mathbb{N}$.
- $|\Sigma_1\,A\,B|\,\gamma$ is simply a set of pairs of values.
- A non-dependent function type $A \mathbin{\widehat{\to}} B$ is defined as the presheaf exponential. The computational part of $|A \mathbin{\widehat{\to}} B|\,\{\Delta\}\,\gamma$ 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 case of dependent functions 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 binders: whenever we go under such binder we have to weaken the current semantic environment. Concretely, 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. For object-level entries in the environment, this is cheap, because we just add an extra explicit weakening, but for semantic values weakening may need to perform deep traversals.

The same efficiency issue arises in formal presheaf-based normalization-by-evaluation, such as in [5] and [15]. 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; see [14], [2] and [1] for implementations in this style. We can use the same solution in 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 drop explicit substitutions from the 2LTT core syntax. In the object-level syntax, we use De Bruijn levels for variables and closures in binders. This lets us completely skip the weakening of environments, which is slightly more efficient than the more faithfully extracted algorithm.

Additionally, we use an untyped tagged representation of semantic values, since in Haskell (the implementation language) we do not have strong enough types to represent typed semantic values. For example, there are distinct data constructors storing semantic function values, natural number literals and quoted expressions.

4.4.1 Caching. In a production-strength implementation we would need some caching mechanism, to avoid excessive duplication of 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 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.

5.1 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. As all constructions in this section must be stable, it makes sense to work internally to $\hat{\mathbb{O}}$. This style has been previously used in normalization proofs [15] and also in the metatheory of cubical type theories [12, 35, 38].

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 1.
The basic scheme for internalization is as follows:

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

5.1.1 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 NatElim, and likewise we have all other type formers.

5.1.2 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.

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

5.1.3 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 $|\mathsf{Ty}_{0,j}|\,\Gamma := \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$, and the displayed presheaf of object-level terms over $\mathsf{Ty}_{0,j}$ as $|\mathsf{Tm}_{0,j}|\,\{\Gamma\}\,A := \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,A$. 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 $\mathsf{Ty}_{\mathbb{O}j}$.

5.1.4 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.

5.2 The Restriction Morphism

We define a family of functions $\mathbb{R}$ from the 2LTT syntax to objects in $\hat{\mathbb{O}}$. We will relate this 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, which are given as $\ulcorner\Gamma\urcorner$.

Definition 5.1. We specify the types of the **restriction operations** internally, and the $|-|$ components of the operations externally. The naturality of $|-|$ is straightforward in each case.

$$\begin{aligned}
&\mathbb{R}\,(\Gamma : \mathsf{Con}) : \widehat{\mathsf{Set}}_0 && \mathbb{R}\,(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) : \mathbb{R}\,\Gamma \to \mathbb{R}\,\Delta\\
&|\mathbb{R}\,\Gamma|\,\Delta := \mathsf{Sub}\,\ulcorner\Delta\urcorner\,\Gamma && |\mathbb{R}\,\sigma|\,\gamma := \sigma \circ \gamma\\
&\\
&\mathbb{R}\,(A : \mathsf{Ty}_{1,j}\,\Gamma) : \mathbb{R}\,\gamma \to \widehat{\mathsf{Set}}_0 && \mathbb{R}\,(t : \mathsf{Tm}_{1,j}\,\Gamma\,A) : (\gamma : \mathbb{R}\,\Gamma) \to \mathbb{R}\,A\,\gamma\\
&|\mathbb{R}\,A|\,\{\Delta\}\,\gamma := \mathsf{Tm}_{1,j}\,\ulcorner\Delta\urcorner\,(A[\gamma]) && |\mathbb{R}\,t|\,\gamma := t[\gamma]\\
&\\
&\mathbb{R}\,(A : \mathsf{Ty}_{0,j}\,\Gamma) : \mathbb{R}\,\Gamma \to \mathsf{Ty}_{0,j} && \mathbb{R}\,(t : \mathsf{Tm}_{0,j}\,\Gamma\,A) : (\gamma : \mathbb{R}\,\Gamma) \to \mathsf{Tm}_{0,j}\,(\mathbb{R}\,A\,\gamma)\\
&|\mathbb{R}\,A|\,\gamma := A[\gamma] && |\mathbb{R}\,t|\,\gamma := t[\gamma]
\end{aligned}$$

5.2.1 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 preservation properties $\mathbb{R}_\bullet$ and $\mathbb{R}_{\triangleright_1}$ mean that $\mathbb{R}$ is a *pseudomorphism* in the sense of [30], 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
$$\mathbb{R}_\Sigma : \mathbb{R}\,(\Sigma\,A\,B)\,\gamma \simeq ((\alpha : \mathbb{R}\,A) \times \mathbb{R}\,B\,(\mathbb{R}_{\triangleright_1}^{-1}\,(\gamma,\ \alpha))).$$
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
$$\mathbb{R}_{\mathsf{app}} : \mathbb{R}\,(\Pi\,A\,B)\,\gamma \to (\alpha : \mathbb{R}\,A\,\gamma) \to \mathbb{R}\,B\,(\mathbb{R}_{\triangleright_1}^{-1}\,(\gamma,\ \alpha))$$
such that $\mathbb{R}_{\mathsf{app}}\,(\mathbb{R}\,t\,\gamma)\,\alpha = \mathbb{R}\,(\mathsf{app}\,t)\,(\mathbb{R}_{\triangleright_1}^{-1}\,(\gamma,\ \alpha))$. 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.

5.2.2 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 $\mathbb{R} \sim t\,\gamma = \sim(\mathbb{R}\,t\,\gamma)$.

5.2.3 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 $- \triangleright_0 -$.

5.2.4 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 $t : \mathsf{Tm}_{0,j}\,\Gamma\,A$.

5.3 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.

$$\begin{aligned}
&(\Gamma : \mathsf{Con})^{\approx} && : \mathbb{E}\,\Gamma \to \mathbb{R}\,\Gamma \to \widehat{\mathsf{Set}}_{\omega}\\
&(\sigma : \mathsf{Sub}\,\Gamma\,\Delta)^{\approx} && : \Gamma^{\approx}\,\gamma\,\gamma' \to \Delta^{\approx}\,(\mathbb{E}\,\sigma\,\gamma)\,(\mathbb{R}\,\sigma\,\gamma')\\
&(A : \mathsf{Ty}_{1,j}\,\Gamma)^{\approx} && : \Gamma^{\approx}\,\gamma\,\gamma' \to \mathbb{E}\,A\,\gamma \to \mathbb{R}\,A\,\gamma' \to \widehat{\mathsf{Set}}_{j}\\
&(t : \mathsf{Tm}_{1,j}\,\Gamma\,A)^{\approx} && : (\gamma^{\approx} : \Gamma^{\approx}\,\gamma\,\gamma') \to A^{\approx}\,\gamma^{\approx}\,(\mathbb{E}\,t\,\gamma)\,(\mathbb{R}\,t\,\gamma')\\
&(A : \mathsf{Ty}_{0,j}\,\Gamma)^{\approx} && : \Gamma^{\approx}\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\,A\,\gamma \urcorner = \mathbb{R}\,A\,\gamma'\\
&(t : \mathsf{Tm}_{0,j}\,\Gamma\,A)^{\approx} && : \Gamma^{\approx}\,\gamma\,\gamma' \to \ulcorner \mathbb{E}\,t\,\gamma \urcorner = \mathbb{R}\,t\,\gamma'
\end{aligned}$$

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 sometimes need to 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 describe the $-^{\approx}$ interpretation in the following.

5.3.1 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 $\bullet^{\approx}\,\gamma\,\gamma' := \top$.

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

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

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.

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

5.3.3 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.

For $\mathsf{zero}^{\approx}$ and $\mathsf{suc}^{\approx}$, we need to show that serialization preserves 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:

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

In short, we need to show that NatElim preserves relations. Here we switch to the external view temporarily. By Definition 5.1, 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 goal type. The β-rules for NatElim are also respected by this definition.

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

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

5.3.5 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:

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

5.3.6 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

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

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 $(\mathsf{Code}\, t)^{\approx} := t^{\approx}$.

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

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

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,

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

5.3.8 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 $t^{\approx}\, \gamma^{\approx}$.

5.3.9 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 likewise for terms. 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.

5.4 Soundness

Definition 5.2. First, we introduce shorthands for external operations that can be obtained from $-^{\approx}$.

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

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

Theorem 5.3 (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 5.4 (Soundness of staging). *The open staging algorithm from Definition 4.14 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 5.5 (Conservativity of 2LTT). *From the soundness and stability of staging, we get that* $\ulcorner - \urcorner$ *is bijective on types and terms, hence* $\mathsf{Ty}_{\mathbb{O}j}\,\Gamma \simeq \mathsf{Ty}_{0,j}\,\ulcorner\Gamma\urcorner$ *and* $\mathsf{Tm}_{\mathbb{O}j}\,\Gamma\,A \simeq \mathsf{Tm}_{0,j}\,\ulcorner\Gamma\urcorner\,\ulcorner A\urcorner$.

5.4.1 Alternative presentations. The above conservativity proof could be rephrased as an instance of more modern techniques which let us implicitly handle stability under 2LTT substitutions as well, not just $\mathbb{O}$ substitutions. This amounts to working in a *modal* internal language, where modalities control the interaction of the different base categories.

- Using *synthetic Tait computability* [42], we work in the internal language of the glued category $\mathsf{Id}_{\hat{\mathbb{O}}} \downarrow \mathbb{R}$.
- Using *relative induction* [7], we work in the modal type theory obtained from the dependent right adjoint functor $\mathbb{R}^* : \widehat{\mathsf{2LTT}} \to \hat{\mathbb{O}}$, where $\widehat{\mathsf{2LTT}}$ denotes presheaves over the syntactic category of 2LTT.

We do not use either of these in this paper, for the following reasons. First, the author of the paper is not sufficiently familiar with the above techniques. Second, the task at hand is not too technically difficult, so using advanced techniques is not essential. Contrast e.g. normalization for cubical type theories, which is not feasible to handle without the more synthetic presentations [43].

6 INTENSIONAL ANALYSIS

We briefly discuss intensional analysis in this section. This means analyzing the internal structure of object-level terms, i.e. values of $\Uparrow A$. Disallowing intensional analysis is a major simplification, which is sometimes enforced in implementations, for example in MetaML [44].

If we have sufficiently expressive inductive types in the meta-level language, it is possible to simply use inductive deeply embedded syntaxes, which can be analyzed; our example in Section 2.2 for embedding a simply typed lambda calculus is like this. However, native metaprogramming features could be more concise and convenient than deep embeddings, similarly to how staged programming is more convenient than code generation with deeply embedded syntaxes.

In the following, we look at the semantics of intensional analysis in the standard presheaf models. We only discuss basic semantics here, and leave practical considerations to future work.

Definition 6.1. The **Yoneda embedding** is a functor from $\mathbb{O}$ to $\hat{\mathbb{O}}$, sending $\Gamma : \mathsf{Con}_{\mathbb{O}}$ to $\mathsf{y}\Gamma : \widehat{\mathsf{Con}}$, such that $|\mathsf{y}\Gamma|\,\Delta = \mathsf{Sub}_{\mathbb{O}}\,\Delta\,\Gamma$. We say that $\Gamma : \widehat{\mathsf{Con}}$ is *representable* if it is isomorphic to a Yoneda-embedded context.

Lemma 6.2 (Yoneda lemma). *We have* $\widehat{\mathsf{Sub}}\,(\mathsf{y}\Gamma)\,\Delta \simeq |\Delta|\,\Gamma$ *[37, Section III.2]. Also, we have* $\widehat{\mathsf{Tm}}_{1,j}\,(\mathsf{y}\Gamma)\,A \simeq |A|\,\mathsf{id}$, *where* $\mathsf{id} : \mathsf{Sub}_{\mathbb{O}}\,\Gamma\,\Gamma$.

The Yoneda lemma restricts possible dependencies on representable contexts. For example, consider the staging behavior of $t : \mathsf{Tm}_1\,(\bullet \triangleright (x : \mathsf{Bool}_0))\,\mathsf{Bool}_1$. Staging t yields essentially a natural transformation $\widehat{\mathsf{Sub}}\,(\mathbb{E}(\bullet \triangleright \mathsf{Bool}_0))\,\mathbb{B}$, where $\mathbb{B}$ is the metatheoretic type with two elements, considered as a constant presheaf. Now, if $\mathbb{E}(\bullet \triangleright \mathsf{Bool}_0)$ is representable, then $\widehat{\mathsf{Sub}}\,(\mathbb{E}(\bullet \triangleright \mathsf{Bool}_0))\,\mathbb{B} \simeq \mathbb{B}$, so t can be staged in at most two different ways.

Which contexts are representable? In the specific 2LTT in this paper, $\ulcorner\Gamma\urcorner$ is always representable in the presheaf model. $|\mathbb{E}\ulcorner\Gamma\urcorner|\,\Delta$ contains lists of object-theoretic terms, hence $\mathbb{E}\ulcorner\Gamma\urcorner \simeq \mathsf{y}\Gamma$. This implies that intensional analysis is *not compatible* with the standard presheaf model, for our 2LTT.

Consider a very basic feature of intensional analysis, *decidability of definitional equality* for object-level Boolean expressions.

$$\mathsf{decEq} : (x\,y : \Uparrow \mathsf{Bool}_0) \to (x =_1 y) +_1 (x \neq_1 y)$$

The Yoneda lemma says that decEq cannot actually decide definitional equality. We expect that decEq lets us define non-constant maps from $\Uparrow \mathsf{Bool}_0$ to Bool_1, but the Yoneda lemma implies that we only have two terms with distinct semantics in $\mathsf{Tm}_1\ (\bullet \triangleright_0 \mathsf{Bool}_0)\ \mathsf{Bool}_1$.

A more direct way to show infeasibility of decEq in $\hat{\mathbb{O}}$ is to note that definitional *inequality* for object-level terms is not stable under substitution, since inequal variables can be mapped to equal terms.

6.0.1 Stability under weakening only. We can move to different 2LTT setups, where stability under substitution is not required. We may have *weakenings* only as morphisms in $\mathbb{O}$. A weakening from Γ to Δ describes how Γ can be obtained by inserting zero or more entries to Δ. A wide range of intensional analysis features are stable under weakenings. For example, decEq now holds in $\hat{\mathbb{O}}$ whenever object-theoretic definitional equality is in fact decidable. We also dodge the Yoneda lemma, since $\mathbb{E}\ulcorner\Gamma\urcorner$ is not necessarily representable anymore: $|\mathbb{E}\ulcorner\Gamma\urcorner|\ \Delta$ is still a set of lists of terms, but weakenings from Δ to Γ are not lists of terms.

As a trade-off, if there is no notion of substitution in the specification of the object theory, it is not possible to specify dependent or polymorphic types there. We need a substitution operation to describe dependent elimination or polymorphic instantiation.

However, this is still sufficient for many practical use cases, for example the monomorphization setup in Section 2.4.1. Here, it makes sense to only have weakening in the object theory, but no $\beta\eta$-rules. We care about the intension of the staging output; presumably we are doing monomorphization because we care about code performance, so in 2LTT we do not want to definitionally equate object-level programs with different performance characteristics.

6.0.2 Closed modalities. Another option is to add a *closed* or *crisp* modality [35] for tracking object-level values which will become closed terms after staging. Since closed terms are not affected by substitution, we are free to analyze them. Such analysis can be viewed as an internalization of canonicity for closed object-level terms. For example, a closed Bool_0 term will be either true_0 or false_0 after staging, so it makes sense to be able to eliminate from it to Bool_1. This can be also viewed as a safe implementation of a *run* function in the style of MetaML [44]. In our case *scope extrusion* would be prevented by the closed modality.

Function pointers in the style of the C programming language could be also an interesting use case, since these must become closed after staging (as they cannot capture closure environments).

7 RELATED WORK

The work Annekov et al. on two-level type theory [6], building on the thesis of Capriotti [10] and ideas from Voevodsky [45], is the foremost inspiration in the current work. Interestingly, the above works do mention metaprogramming as a source of intuition for 2LTT, but they only briefly touch on this aspect, and in [6] the main focus is on extensions of basic 2LTT which do not have staging semantics anymore. Ibid. conjectures the strong conservativity of 2LTT in Proposition 2.18, and suggests categorical gluing as proof strategy. In this work, we do essentially that: our presheaf model and the logical relation model could be merged into a single instance of gluing along the $\mathbb{R}$ morphism; this merges staging and its soundness proof. We split this construction to two parts in order to get a self-contained presentation of the staging algorithm.

The multi-stage calculus of Kawata and Igarashi [32] supports staging operations and type dependencies. However, there are major differences to the current work. First, ibid. does not have large elimination, hence it does not support computing types by staging. Second, it does not support staged compilation in the sense of Definition 4.1; rather, it is a system for runtime code manipulation and evaluation (which we may call staged *computation*). In other words, programs may manipulate

code fragments during execution, in more liberal ways than in 2LTT, but there is no judgment which ensures that a purely "object-level" program can be extracted from a multi-stage program. Similar considerations apply to systems with the S4 modality (see e.g. Davies and Pfenning [19]), where there is no judgment that guarantees elimination of modal operations.

Idris 1 supports compile-time partial evaluation, through `static` annotations on function arguments [9]. This can be used for dependently typed code generation, but Idris 1 does not guarantee that partial evaluation adequately progresses. For example, we can pass a neutral value as static argument in Idris 1, while in 2LTT static function arguments are always canonical during staging.

Our notation for quoting and splicing is borrowed from MetaML [44]. In the following, we compare 2LTT to MetaML, MetaOCaml [33] and typed Template Haskell [47].

2LTT explicitly tracks stages of types, in contrast to the mentioned systems. There, we have type lifting (usually called Code), quoting and splicing, but lifting does not change the universe of types. We can write a function with type $\mathsf{Bool} \to \mathsf{Code}\ \mathsf{Bool}$, and the two Bool occurrences refer to the exact same type.

This causes some operational confusion down the line, and additional disambiguation is needed for stages of binders. For example, in typed Template Haskell, top-level binders behave as runtime binders when used under a quote, but behave as static binders otherwise. Given a top-level definition $b = \mathsf{True}$, if we write a top-level definition $\mathsf{expr} = [|| b ||]$, then b is a *variable* in the staging output, but if we write $\mathsf{expr} = \mathsf{if}\ b\ \mathsf{then}\ [||\ \mathsf{True}\ ||]\ \mathsf{else}\ [||\ \mathsf{False}\ ||]$, then b is considered to be a static value. In contrast, 2LTT does not distinguish top-level and local binders, and in fact it has no syntactic or scope-based restrictions; everything is enforced by typing.

2LTT supports staging for types and dependent types, while to our knowledge no previous system for staged compilation does the same. It appears to the author that the main missing component in previous systems is *dependent types at the meta level*, rather than staging operations; staging operations in 2LTT and MetaML are already quite similar. For example, any interesting operation on the static-length vectors in Section 2.2 requires meta-level dependent types, even if the object language is simply typed.

In MetaML and MetaOCaml, there is a *run* function for evaluating closed object-level terms: as we mentioned in Section 6.0.2, we believe that the proper way to implement it in 2LTT would be to use a closed modality. This use case is closely related to [35]. In ibid. a semantics is sketched for this modality in a presheaf topos, and we could directly adapt it to our presheaf model of 2LTT. That said, working out the practical details of this modality remains future work.

Additionally, 2LTT only supports two stages, while the other noted systems allow countable stages. It remains future work to develop NLTT (N-level type theory), but this extension does appear to be straightforward. In the specification of NLTT, we would simply need to move from $i \in \{0, 1\}$ to $i \in \mathbb{N}$, although in the semantics there could be more subtleties.

Many existing staged systems also support *side effects* at compile time, while our version of 2LTT does not. Here, general considerations about the mixing of dependent types and effects should apply; see e.g. [39]. However, it should be possible to use standard effect systems such as monads at the meta level.

8 CONCLUSIONS AND FUTURE WORK

We developed the foundational metatheory of 2LTT as a system of two-stage compilation. We view the current results as more like a starting point to more thorough investigation into applications and extensions; in this paper we only sketched these.

We emphasize that variants of 2LTT can serve as languages where staging features can be practically implemented, and also as *metatheories*, not only for the formalization for staging techniques, but also more generally for reasoning about constructions in object languages. This

purpose is prominent in the original 2LTT use case in synthetic homotopy theory. The meta-level language in a 2LTT can be expressive enough to express general mathematics, and the meta-level propositional equality $- =_1 -$ can be used to prove $\beta\eta$-equality of object-level types and terms.

For example, we could use object theories as *shallowly embedded* target languages of optimization techniques and program translations. In particular, we may choose a low-level object theory without higher-order functions, and formalize closure conversion as an interpretation of an embedded meta-level syntax into the object theory.

There is much future work in adapting existing staging techniques to 2LTT (which, as we mentioned, can double as formalization of said techniques), or adding staging where it has not been available previously.

- *Let-insertion techniques.* These allow more precise control over positions of insertion, or smarter selection of such positions. In 2LTT, meta-level continuation monads could be employed to adapt some of the features in [29]. The automatic let-floating feature of MetaOCaml [34] requires strengthening of terms, which is not stable under substitution, but it is stable under weakening, so perhaps it could be implemented as an "intensional analysis" feature.
- *Generic programming with dependent types.* There is substantial literature in this area (e.g. [13, 17, 21, 36]), but only in non-staged settings. 2LTT should immediately yield staging for these techniques, assuming that the meta level has sufficient type formers (e.g. induction-recursion). We could also try to adapt previous work on generic treatment of partially static data types [48]; fully internalizing this in 2LTT would also require dependent types.

There are also ways that 2LTT itself could be enhanced.

- *More stages, stage polymorphism.* Currently, there is substantial code duplication if the object-level and meta-level languages are similar. Stage polymorphism could help reduce this. A simple setup could include three stages, the runtime one, the static one, and another one which supports polymorphism over the first two stages.
- *Modalities.* We mentioned the closed modality in Section 6.0.2. More generally, many *multi-modal* [24] type theories could plausibly support staging. 2LTT itself can be viewed as a very simple multimodal theory with $\Uparrow$ as the only modality, which is also degenerate (because it entails no structural restriction on contexts). We could support multiple object theories, with modalities for lifting them to the meta level or for representing translations between them. We could also have modalities for switching between stability under substitution and stability under weakening.
- *Intensional analysis.* We only touched on the most basic semantics of intensional analysis in Section 6. It remains a substantial challenge to work out the practicalities. For good ergonomics, we would need something like a "pattern matching" operation on object-level terms, or some induction principle which is more flexible than the plain assumption of decidable object-level equality.

REFERENCES

[1] Andreas Abel and Thierry Coquand. 2007. Untyped Algorithmic Equality for Martin-Löf's Logical Framework with Surjective Pairs. *Fundam. Informaticae* 77, 4 (2007), 345–395. http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-05

[2] Andreas Abel, Thierry Coquand, and Miguel Pagano. 2011. A Modular Type-checking algorithm for Type Theory with Singleton Types and Proof Irrelevance. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:4)2011

[3] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. https://doi.org/10.1145/3158111

[4] Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 18–29. https://doi.org/10.1145/2837614.2837638

[5] Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by Evaluation for Type Theory, in Type Theory. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017). https://doi.org/10.23638/LMCS-13(4:1)2017

[6] 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

[7] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. 2021. Relative induction principles for type theories. *arXiv preprint arXiv:2102.11649* (2021).

[8] Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *Theor. Comput. Sci.* 39 (1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

[9] Edwin C. Brady and Kevin Hammond. 2010. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 297–308. https://doi.org/10.1145/1863543.1863587

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

[11] Simon Castellan, Pierre Clairambault, and Peter Dybjer. 2019. Categories with Families: Unityped, Simply Typed, and Dependently Typed. *CoRR* abs/1904.00827 (2019). arXiv:1904.00827 http://arxiv.org/abs/1904.00827

[12] Evan Cavallo, Anders Mörtberg, and Andrew W. Swan. 2020. Unifying Cubical Models of Univalent Type Theory. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain (LIPIcs, Vol. 152)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:17. https://doi.org/10.4230/LIPIcs.CSL.2020.14

[13] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. *ACM Sigplan Notices* 45, 9 (2010), 3–14.

[14] Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Sci. Comput. Program.* 26, 1-3 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

[15] Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015

[16] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 315–326. https://doi.org/10.1145/1291151.1291199

[17] Pierre-Évariste Dagand. 2017. The essence of ornaments. *J. Funct. Program.* 27 (2017), e9. https://doi.org/10.1017/S0956796816000356

[18] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. https://doi.org/10.1145/236114.236119

[19] Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

[20] Agda developers. 2022. Agda documentation. https://agda.readthedocs.io/en/v2.6.2.1/

[21] Larry Diehl. 2017. *Fully Generic Programming over Closed Universes of Inductive-Recursive Types.* Ph. D. Dissertation. Portland State University.

[22] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2021), 98:1–98:38. https://doi.org/10.1145/3450952

[23] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. https://doi.org/10.1145/165180.165214

[24] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 492–506. https://doi.org/10.1145/3373718.3394736

[25] Martin Hofmann. 1995. *Extensional concepts in intensional type theory.* University of Edinburgh, Department of Computer Science.

[26] Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation.* Cambridge University Press, 79–130.

[27] Simon Huber. 2016. *Cubical Interpretations of Type Theory.* Ph. D. Dissertation. University of Gothenburg.

[28] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

[29] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the stage - Staging with delimited control. *J. Funct. Program.* 21, 6 (2011), 617–662. https://doi.org/10.1017/S0956796811000256

[30] 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

[31] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.* 3, POPL (2019), 2:1–2:24. https://doi.org/10.1145/3290315

[32] Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11893)*, Anthony Widjaja Lin (Ed.). Springer, 53–72. https://doi.org/10.1007/978-3-030-34175-6_4

[33] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[34] Oleg Kiselyov and Jeremy Yallop. 2022. let (rec) insertion without Effects, Lights or Magic. *CoRR* abs/2201.00495 (2022). arXiv:2201.00495 https://arxiv.org/abs/2201.00495

[35] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22

[36] Andres Löh and José Pedro Magalhães. 2011. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Jaakko Järvi and Shin-Cheng Mu (Eds.). ACM, 1–12. https://doi.org/10.1145/2036918.2036920

[37] Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2nd ed.). Springer. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0387984038

[38] Ian Orton and Andrew M. Pitts. 2016. Axioms for Modelling Cubical Type Theory in a Topos. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:19. https://doi.org/10.4230/LIPIcs.CSL.2016.24

[39] Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.* 4, POPL (2020), 58:1–58:28. https://doi.org/10.1145/3371126

[40] Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. https://doi.org/10.1145/3498693

[41] Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

[42] Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability.* Ph. D. Dissertation. Carnegie Mellon University Pittsburgh, PA.

[43] Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–15. https://doi.org/10.1109/LICS52264.2021.9470719

[44] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

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

[46] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. 2019. Eliminating reflection from type theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 91–103. https://doi.org/10.1145/3293880.3294095

[47] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed Template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498723

[48] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.* 2, ICFP (2018), 100:1–100:30. https://doi.org/10.1145/3236795