Introduction to Staged Compilation and Two-Level Type Theory

Yulong Liu Youzhang Sun

1 Introduction to Staged Compilation

The purpose of staged compilation is to write expressive metaprograms that generate code with the guarantees that the generated code is well-formed. To justify the well-formedness of the code output, the model of two-level type theory (2LTT) [Ann+19] is employed as a formal typing system for staged compilation. While languages such as MetaML [TS97] supports metaprogramming, 2LTT additionally supports dependent types. Therefore, 2LTT contributes to the field of programming language and type theory by introducing dependent types to metaprogramming.

In this paper, we focus on metaprogramming with two stages. We index these stages as stage 0 and stage 1. Each stage has a language which we will further formalise into a type system. We use the term *staging algorithm* to refer the process of transforming (i.e. staging) a metaprogram to a program that only uses the stage 0 language. A metaprogram is a term with stage 0 type but uses type/terms from the stage 1 language through staging annotations. To explain how a metaprogram can use stage 1 language despite being at stage 0, we first describe the interaction between these two stages.

1.1 Interaction between Stages

While 2LTT provides safety about the output code staged from a metaprogram, the only way to interact between stages is through these three staging operations: lifting, quoting, and splicing.

- If we have a stage 0 type A_0, we can *lift* A_0, denoted as $\Uparrow A_0$. The type $\Uparrow A_0$ describes the type of metaprograms that compute a stage 0 expression of type A_0.
- If we have a stage 0 term t of type A_0, we can *quote* t, denoted as $\langle t \rangle$. The term $\langle t \rangle$ describes a metaprogram that immediately yields t.
- If we have a stage 1 term s of type $\Uparrow A_0$, we can *splice* s, denoted as $\sim s$. The term $\sim s$ describes a term that is to be executed during staging with the resulting expression substituted back to the output code.

The combinations of these staging operations allows us to lift stage 0 terms and types to stage 1, and bring lifted terms back to stage 0.

1.2 Example of A Staged Program

In this section, we start with an example of a program without staging annotations; then we will convert this program into a metaprogram to demonstrate the usage of staging operations. After introducing inference rules of 2LTT in later sections, we will revisit this example by type-checking and staging it according to the inference rules.

For now, let's consider a programming language that provides addition for natural numbers. We can implement a multiplication algorithm using the provided addition function recursively. Thus, we can write a program, called $\mathsf{double} : \mathsf{Nat} \to \mathsf{Nat}$ that fixes the first argument of mul:

$$
\begin{aligned}
&\mathsf{zero} : \mathsf{Nat} \\
&\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat} \\
&\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{mul} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
&\mathsf{mul}\,\mathsf{zero}\,x = \mathsf{zero} \\
&\mathsf{mul}\,(\mathsf{suc}\,n)\,x = \mathsf{add}\,x\,(\mathsf{mul}\,n\,x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{double} : \mathsf{Nat} \to \mathsf{Nat} \\
&\mathsf{double} := \lambda\, x.\, \mathsf{mul}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}))\,x
\end{aligned}
$$

Currently, there is no distinction between the stage of our function mul and add. For the purpose of demonstrating the effect of staging a metaprogram, let us lift the function mul to stage 1 and keep add at stage 0. The staging algorithm will stage double into a program with no occurrences of mul. In other words, we can stage the code $\lambda\, x.\, \mathsf{mul}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}))\,x$ into $\lambda\, x.\, \mathsf{add}\,x\,(\mathsf{add}\,x\,\mathsf{zero})$ through substitution with its definition during staging.

Therefore, we rewrite these function as mul_1 and add_0 to indicate their stage. Because both functions add_0 and mul_1 take a natural number as argument but compute in different stages, we also need each stage to have a type for describing naturals. Thus to distinguish between naturals in these two stages, we index the type by its stage. We define Nat_0 to be the type of natural numbers at stage 0, and Nat_1 to be the type of natural numbers at stage 1. We treat these two types as separate types, thus the constructors of these naturals are also indexed: zero_1 and suc_1 for stage 1, and zero_0 and suc_0 for stage 0.

$$
\begin{aligned}
&\mathsf{zero}_0 : \mathsf{Nat}_0 \\
&\mathsf{suc}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\
&\mathsf{add}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{zero}_1 : \mathsf{Nat}_1 \\
&\mathsf{suc}_1 : \mathsf{Nat}_1 \to \mathsf{Nat}_1
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{mul}_1 : \mathsf{Nat}_1 \to \Uparrow\mathsf{Nat}_0 \to \Uparrow\mathsf{Nat}_0 \\
&\mathsf{mul}_1\,\mathsf{zero}_1\,t = \langle \mathsf{zero}_0 \rangle \\
&\mathsf{mul}_1\,(\mathsf{suc}_1\,n)\,t = \langle \mathsf{add}_0\,{\sim}t\,{\sim}(\mathsf{mul}_1\,n\,t) \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{double}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\
&\mathsf{double}_0 := \lambda\, x.\, {\sim}(\mathsf{mul}_1\,(\mathsf{suc}_1\,(\mathsf{suc}_1\,\mathsf{zero}_1))\,\langle x \rangle)
\end{aligned}
$$

We also rewrite our original double program as $\mathsf{double}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0$ since it is the program we want to stage into the output code. However, the argument of the function double_0, namely $x : \mathsf{Nat}_0$, is passed to mul_1, a function at stage 1. Therefore, we need to quote this argument as $\langle x \rangle : \Uparrow\mathsf{Nat}_0$ to lift x to stage 1 for the metaprogram mul_1 to compute.

Therefore, our program double_0 takes a $x : \mathsf{Nat}_0$ as a stage 0 term, quotes it as $\langle x \rangle : \Uparrow A$ to pass it to the metaprogram $\mathsf{mul}_1\,(\mathsf{suc}_1\,(\mathsf{suc}_1\,\mathsf{zero}_1))$, which would gives us a lifted term in stage 1. Then, we can splice the output of mul_1 to get back a stage 0 term.

For the metaprogram mul_1, we follow its recursive definition with respect to the first argument, a Nat_1 from stage 1. Since mul_1 is a metaprogram that generates a lifted code for stage 0, namely with type $\Uparrow\mathsf{Nat}_0$, we splice the output inside double_0 to bring the term back to stage 0. Therefore, we quote the base case as $\langle \mathsf{zero}_0 \rangle$ and the recursive step as $\langle \mathsf{add}_0\ \cdots \rangle$ to yield these terms during staging. In the recursive step, as t is passed as a $\Uparrow\mathsf{Nat}_0$, namely $\langle x \rangle$ from double_0, we splice it to get $x : \mathsf{Nat}_0$ and pass it to the stage 0 function add_0.

Once we feed this program to the staging algorithm, we will get a well-typed program where the occurrence of mul_1 inside double_0 is substituted by its compile time definition during staging:

$$\begin{aligned}
\mathsf{zero}_0 &: \mathsf{Nat}_0 \\
\mathsf{suc}_0 &: \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\
\mathsf{add}_0 &: \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0
\end{aligned}$$

$$\begin{aligned}
\mathsf{double}_0 &: \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\
\mathsf{double}_0 &:= \lambda\, x.\, \mathsf{add}_0\, x\, (\mathsf{add}_0\, x\, \mathsf{zero}_0)
\end{aligned}$$

1.3 Soundness of Staging

The result of staging double_0 was a well-typed function that only uses terms and constructors native to stage 0. We would like to generalize so any metaprogram should be staged to a program that only uses terms and constructors from stage 0. We use the term "sound" to describe such staging algorithm.

Definition 1.3.1 (Soundness, Well-staged, Well-formedness)**.** A staging algorithm is *sound* if for all input program with stage 0 type, the algorithm outputs a well-typed program containing only stage 0 terms / constructors, and no lift, quote, or splice.

A well-typed output program that contains only stage 0 terms / constructors, and no lift, quote, or splice, is called *well-staged* or *well-formed*.

To justify the soundness of staging algorithm, we use two-level type theory, which the remaining paper will explore.

2 Introduction to Two-Level Type Theory

Two-level type theory (2LTT), as the name suggests, extends type theory to two levels, which in our case, are the stages 0 and 1. 2LTT is useful since we can extend the language of stage 0 as the stage 1 language to derive useful properties that are not expressable within stage 0; namely, the soundness of a staging algorithm.

To apply 2LTT in staged programming, we consider the languages used in the two stages as separate type systems. This separation provides support for a wide range of languages even with completely different syntax. For instance, staged compilation with 2LTT is applicable to domain specific languages whose implementations are in different languages (e.g. staging LINQ expressions to C# method calls [Dev23]).

Since 2LTT formalises our metaprograms and output code into type theories, we provide an example of a 2LTT model that consists of universe hierarchies, type formers with dependent types, and formers/eliminators for natural numbers. We will also present a portion of the inference rules for the purpose of formalising the staging operations and working with natural numbers. Next, we will revisit the double_0 metaprogram for type-checking and applying the staging algorithm with these inference rules. We will then round off this paper with some discussions and conclusion.

2.1 Universes and Type Formers

The types that we construct in both stages can be treated as terms as well; a type whose terms are also types is called a universe. As universes themselves are types, a hierarchy of universes is formed for each stage. We denote these universes as $\mathsf{U}_{i,j}$ for $i \in \{0, 1\}$ and $j \in \mathbb{N}$ where i indicates the stage and j for the universe level, that is $\mathsf{U}_{i,j}$ is a term of $\mathsf{U}_{i,j+1}$.

In staged programming, the universes $\mathsf{U}_{0,j}$ are inhabited by types from stage 0 and $\mathsf{U}_{1,j}$ by types from stage 1. For instance, Nat_0 as a term has the type $\mathsf{U}_{0,0}$ whereas Nat_1 as a term has the type $\mathsf{U}_{1,0}$. As mentioned before, we can lift a stage 0 type to stage 1, thus for any $A : \mathsf{U}_{0,j}$, we have $\Uparrow A : \mathsf{U}_{1,j}$.

To construct new types, there are type formers (also known as type constructors) for each universe and stage. Type formers takes types as arguments and construct a new type in the same stage. For instance, the Π-type constructor combines a domain type $A : \mathsf{U}_{i,j}$ and a codomain type $B : \mathsf{U}_{i,j}$ to form the function type $\Pi\, A\, B : \mathsf{U}_{i,j}$ which describes the type of a function that maps an A to a B. We notice here A and B are from the same stage. All type formers require the input types to be from the same stage, and outputs a type of the same stage.

Up until now, we have given a informal description of the natural number type and staging operations to introduce staged compilation through examples. In the next sections, we formalise these ideas through inference rules.

3 Inference Rules of 2LTT

In this section, we analyze the inference rules of 2LTT. As 2LTT is an extension of Martin-Löf type theory, many concepts involve to dependent functions. Therefore, to denote the type of a dependent function, we use the notation $(x : A) \to B$

where the term x may occur in the type B. We also use the alternative notation $(x : A) \to B\,x$, which clarifies that B is a type dependent on x.

3.1 Judgments

We start by listing the forms of judgments defined for 2LTT relevant to our use. We will then explain the meaning of each judgment presented.

Because there are two different stages in 2LTT, a type in one stage is often not available in the other stage. As 2LTT supports the hierarchy of universes, we need to take into consideration the level we are at when making a judgment. We assume index $i \in \{0, 1\}$ to indicate the stage, and $j \in \mathbb{N}$ for the universe level.

Notation 3.1.1. "$\Gamma \vdash_{i,j} \ldots$" is the notation for making the judgment in stage i and universe level j.

We also use the following convention:

- Uppercase Greek letters Γ, Δ for context
- Lowercase Greek letters σ, δ for context substitution
- Uppercase alphabet A, B, C for types
- Lowercase alphabet t, u, v for terms

Definition 3.1.1 (Judgments of 2LTT)**.**

$\Gamma \vdash$	*context formation*
$\Gamma \vdash \sigma : \Delta$	*explicit substitution formation, assuming* $\Gamma \vdash$ *and* $\Delta \vdash$
$\Gamma \vdash_{i,j} A$	*type formation, assuming* $\Gamma \vdash$
$\Gamma \vdash_{i,j} t : A$	*term formation, assuming* $\Gamma \vdash_{i,j} A$
$\Gamma \vdash_{i,j} A = B$	*type equality, assuming* $\Gamma \vdash_{i,j} A$ *and* $\Gamma \vdash_{i,j} B$
$\Gamma \vdash_{i,j} t = u : A$	*term equality, assuming* $\Gamma \vdash_{i,j} t : A$ *and* $\Gamma \vdash_{i,j} u : A$

We now explain each judgment in detail.

Definition 3.1.2 (Form of Judgment Context Formation)**.**

$$\Gamma \vdash \text{ means "}\Gamma \text{ is a context"}$$

We always have the empty context, denoted as $\bullet$. Thus, the judgment "$\bullet \vdash$" would be a believable judgment.

Definition 3.1.3 (Substitution and De Bruijn Indices)**.**

$$\Gamma \vdash \sigma : \Delta \text{ means "under the context } \Gamma, \sigma \text{ is a substitution from } \Delta\text{"}$$

In short, a substitution is a mapping of terms and types from one context to another. In particular, $\Gamma \vdash \sigma : \Delta$ means σ maps terms and types from the context Δ to the context Γ. If $\Delta \vdash_{i,j} A$, that is, A is a type under the context Δ, then we can apply the substitution to derive that $\Gamma \vdash_{i,j} A[\sigma]$, that is, $A[\sigma]$ is a type under the context Γ. We will discuss applying substitions in detail through inference rules in later sections.

For this paper, we only work with the *weakening substitution* p for the purpose of employing De Bruijn indices. The De Bruijn index is a convenient way of refering to the terms in a context without putting them in it explicitly. Again, we will provide examples in the section of inference rules.

Definition 3.1.4 (Forms of Judgment for Type and Term Formation)**.**

$\Gamma \vdash_{i,j} A$	means "A is a type of stage i , universe level j"
$\Gamma \vdash_{i,j} t : A$	means "t is a term of type A in stage i, universe level j "

Example 3.1.1. The expression "$\Gamma \vdash_{1,10} A$" means "A is a type in stage 1 and universe level 10"

Example 3.1.2. Consider the two Nat types for the two stages: Nat_0 and Nat_1. While the type Nat_0 belong strictly in stage 0, the type Nat_1 is in stage 1. So $\Gamma \vdash_{0,j} \mathsf{Nat}_0$ would be a believable judgment, but $\Gamma \vdash_{1,j} \mathsf{Nat}_0$ is not believable.

The judgment "$\Gamma \vdash_{i,j} A$" makes the presupposition that Γ is a valid context ($\Gamma \vdash$). The judgment "$\Gamma \vdash_{i,j} t : A$" assumes A is a type in stage i universe level j ($\Gamma \vdash_{i,j} A$)

Definition 3.1.5 (Forms of Judgment for Type and Term Equality)**.**

$\Gamma \vdash_{i,j} A = B$	means "A and B are the same type in stage i, universe level j"
$\Gamma \vdash_{i,j} t = u : A$	means "t and u are the same term of type A in stage i, universe level j"

Judgmental equality also require specifying the stage and universe level. They also come with the presupposition that the two expressions in comparison are from the same stage and universe.

Example 3.1.3 (Ill-formed Judgment)**.** $\Gamma \vdash_{0,0} \mathsf{Nat}_0 = \mathsf{Nat}_1$ is not well formed. Although $\Gamma \vdash_{0,0} \mathsf{Nat}_0$ is believable, $\Gamma \vdash_{0,0} \mathsf{Nat}_1$ is not believable. So we cannot judge the equality of the two types.

3.2 Familiar Inference Rules in the Context of 2LTT

Here we look at some inference rules that are common in many type theories, but now they involve stage and universe level.

3.2.1 Context

We always have access to the empty context, as represented by the axiom:

$$\textsc{empty-cxt}\ \frac{}{\bullet \vdash}$$

If a type A can be derived from a context Γ, we can extend Γ with the type A

$$\textsc{cxt-extension}\ \frac{\Gamma \vdash \qquad \Gamma \vdash_{i,j} A}{\Gamma \triangleright A \vdash}$$

An interesting observation is that types from all stages and universe levels are treated equally with extending context. A type in stage 0 extends the context in the same way as a type in stage 1.

3.2.2 Substitutions

As mentioned in in the previous section on judgments, we will only employ the weakening substitution for De Bruijn indices. Nonetheless, it is important to mention that substitutions have an identity before covering weakening substitutions:

$$\textsc{identity-sub}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{id} : \Gamma} \qquad \textsc{type-id-sub}\ \frac{\Gamma \vdash_{i,j} A}{\Gamma \vdash_{i,j} A[\mathsf{id}] = A} \qquad \textsc{term-id-sub}\ \frac{\Gamma \vdash_{i,j} t : A}{\Gamma \vdash_{i,j} t[\mathsf{id}] = t : A}$$

Since we only extend a type to the context through CXT-EXTENSION rather than a type-term pair, we need to refer to the terms in our context. To do so, we can derive the *zero variable* q and the *weakening substitution* p when extending a context with an arbitrary type:

$$\textsc{sub-first-proj}\ \frac{\Gamma \vdash_{i,j} A}{\Gamma \triangleright A \vdash \mathsf{p} : \Gamma} \qquad \textsc{sub-second-proj}\ \frac{\Gamma \vdash_{i,j} A}{\Gamma \triangleright A \vdash \mathsf{q} : A[\mathsf{p}]}$$

From the rule SUB-SECOND-PROJ, the term q refers to a term in the context $\Gamma \triangleright A$, in particular, with type A that we have just extended. In general, the term q always points to the most recently extended type. Similarly, the weakening substitution p that is applied to the type $A[\mathsf{p}]$ under the context $\Gamma \triangleright A$ means that $A[\mathsf{p}]$ refers to the type inside the context (namely, the most recently extended one) rather than a newly formed type. Therefore, if we continue to extend more types to the context, we need to shift the De Bruijn index q with these substitution rules:

$$\textsc{type-sub}\ \frac{\Delta \vdash_{i,j} A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{i,j} A[\sigma]} \qquad \textsc{term-sub}\ \frac{\Delta \vdash_{i,j} t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{i,j} t[\sigma] : A[\sigma]}$$

For example, suppose we can form the type A under context Γ and the type B under $\Gamma \triangleright A$, then we can derive the following De Bruijn indices, which we will use colors to match with their type in the context:

$$\dfrac{\dfrac{\Gamma \vdash A}{\Gamma \triangleright A \vdash \mathsf{q} : A[\mathsf{p}]}\,\textsc{sub-second-proj} \quad \dfrac{\Gamma \triangleright A \vdash B}{\Gamma \triangleright A \triangleright B \vdash \mathsf{p} : \Gamma \triangleright A}\,\textsc{sub-first-proj}}{\Gamma \triangleright A \triangleright B \vdash \mathsf{q}[\mathsf{p}] : A[\mathsf{p}][\mathsf{p}]}\,\textsc{term-sub} \qquad \dfrac{\Gamma \triangleright A \vdash B}{\Gamma \triangleright A \triangleright B \vdash \mathsf{q} : B[\mathsf{p}]}\,\textsc{sub-second-proj}$$

Next, as substitutions are mappings from one context to another, they can be composed. In addition, applying the composition of two substitutions is judgmentally equal to applying the first, then applying the second. Despite the fact that there are other properties on substitution composition such as associativity and composing substitutions on terms, they won't be used for the purpose of this paper. Nonetheless, we have these inference rules:

$$\textsc{sub-composition}\ \dfrac{\Delta \vdash \sigma : \Theta \qquad \Gamma \vdash \delta : \Delta}{\Gamma \vdash \sigma \circ \delta : \Theta} \qquad \textsc{type-comp-sub}\ \dfrac{\Theta \vdash_{i,j} A \qquad \Delta \vdash \delta : \Theta \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{i,j} A[\sigma \circ \delta] = A[\sigma][\delta]}$$

Lastly, substitutions can be extended with a term, and applying an extended substitution to a term is analogous to function application. For instance, if a funtion f is defined as $\lambda\, x.\mathrm{body}$ for some term body where x might appear in, then the function application $f\ a$ results the same expression as the substitution $\mathrm{body}[\mathsf{id}, a]$ where (id, a) is the identity substitution extended with the term a. Extending a substitution is rarely used in this paper other than for the elimination of naturals, but for completeness, we have these inference rules:

$$\textsc{sub-extension}\ \dfrac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash t : A[\sigma]}{\Gamma \vdash (\sigma, t) : \Delta \triangleright A}$$

As we revisit the double_0 example for type-checking and staging, we will intensively apply these inference rules on substitution along with the three staging operations.

3.2.3 Universes

Both stage 0 and stage 1 supports hierarchy of universes. 2LTT chooses the Russell-style universes as the framework for implementing hierarchy.

Definition 3.2.1 (Russell-style Universe)**.** *Russell-style universes* have types as terms of universes. For example, it has $\Gamma \vdash A : \mathsf{U}$ where A is a type.

The rule UNIVERSE describes the hierarchy:

$$\textsc{universe}\ \dfrac{}{\Gamma \vdash_{i,j+1} \mathsf{U}_j}$$

Notice the index shift by 1 between the type we are judging and the universe we are making judgment in. The rule states that regardless of the stage, a universe is a term of the next largest universe. This is a reflection of the Russell-style universes.

3.2.4 Natural Numbers (Nat)

Within the context of staged compilation 2LTT, we assume all universes across both stages have access to Nat. Nat is implemented through Peano Arithmetic, meaning we have term zero and the term former suc at our disposal.

$$\textsc{nat-formation}\ \dfrac{}{\Gamma \vdash_{i,j} \mathsf{Nat}} \qquad \textsc{zero}\ \dfrac{}{\Gamma \vdash_{i,j} \mathsf{zero} : \mathsf{Nat}} \qquad \textsc{suc}\ \dfrac{\Gamma \vdash_{i,j} t : \mathsf{Nat}}{\Gamma \vdash_{i,j} \mathsf{suc}\, t : \mathsf{Nat}}$$

We emphasis that although Nat is a type in any stage and universe, two Nats are not the same when they are from different stage or universe by our judgments. Type and term equality require the presupposition of the two types coming from the same stage and universe level. We use the notation Nat_0 and Nat_1 to extinguish Nat from different stages.

We also have eliminator for Nat in the form of NatElim. This eliminator also comes with β reductions as inference rules so we can apply substitutions on a given target.

$$\textsc{nat-elim}\ \frac{\Gamma \triangleright \mathsf{Nat} \vdash_{i,k} P \quad \Gamma \vdash_{i,k} z : P[\mathsf{id}, \mathsf{zero}] \quad \Gamma \triangleright \mathsf{Nat} \triangleright P \vdash_{i,k} s : P[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}\,(\mathsf{q}[\mathsf{p}])] \quad \Gamma \vdash_{i,j} t : \mathsf{Nat}}{\Gamma \vdash_{i,k} \mathsf{NatElim}\, P\, z\, s\, t : P[\mathsf{id}, t]}$$

$$\textsc{zero-}\beta\ \frac{}{\Gamma \vdash_{i,j} \mathsf{NatElim}\, P\, z\, s\, \mathsf{zero} = z : P[\mathsf{id}, \mathsf{zero}]} \qquad \textsc{suc-}\beta\ \frac{}{\Gamma \vdash_{i,j} \mathsf{NatElim}\, P\, z\, s\, (\mathsf{suc}\, t) = s[\mathsf{id}, t, \mathsf{NatElim}\, P\, z\, s\, t] : P[\mathsf{id}, \mathsf{suc}\, t]}$$

While the premises of NAT-ELIM involve some menacing substitutions on the type P, the substitutions on the type Nat and its term formers are rather simple. Again, we describe them as inference rules:

$$\textsc{nat-sub}\ \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{i,j} \mathsf{Nat}[\sigma] = \mathsf{Nat}} \qquad \textsc{zero-sub}\ \frac{\Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{i,j} \mathsf{zero}[\sigma] = \mathsf{zero} : \mathsf{Nat}} \qquad \textsc{suc-sub}\ \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash_{i,j} t : \mathsf{Nat}}{\Gamma \vdash_{i,j} (\mathsf{suc}\, t)[\sigma] = \mathsf{suc}\,(t[\sigma]) : \mathsf{Nat}}$$

As described in the section on De Bruijn indices, substitutions can be thought as function applications. Therefore, by applying the notation on dependently typed functions, we can the eliminator of Nat as follows:

$$\begin{aligned}\mathsf{NatElim} : {} & (P : \mathsf{Nat} \to \mathsf{U}_{i,j}) \\ & \to P\,\mathsf{zero} \\ & \to ((n : \mathsf{Nat}) \to P\, n \to P\,(\mathsf{suc}\, n)) \\ & \to (t : \mathsf{Nat}) \\ & \to P\, t\end{aligned}$$

Therefore, NatElim takes a motive $P : \mathsf{Nat} \to \mathsf{U}_{i,j}$, a base $z : P\,\mathsf{zero}$, a step function $s : (n : \mathsf{Nat}) \to P\, n \to P\,(\mathsf{suc}\, n)$, and a target $t : \mathsf{Nat}$ as arguments; then it eliminates the target to $\mathsf{Nat}\, P\, z\, s\, t : P\, t$. The rules of elimination is described as the β-reductions, which we can rephrase as:

$$\begin{aligned}\mathsf{zero}\text{-}\beta &: \mathsf{NatElim}\, P\, z\, s\, \mathsf{zero} = z \\ \mathsf{suc}\text{-}\beta &: \mathsf{NatElim}\, P\, z\, s\, (\mathsf{suc}\, n) = s\, n\, (\mathsf{NatElim}\, P\, z\, s\, n)\end{aligned}$$

Example 3.2.1 (Addition and Multiplication)**.** We can implement addition, $\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat}$ between two naturals $a, b : \mathsf{Nat}$ by eliminating a with NatElim. More specifically, on each suc-β rule, we wrap b (i.e. the base) with suc:

$$\begin{aligned}&\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \\ &\mathsf{add} : \lambda\, a\, b.\, \mathsf{NatElim}\, (\lambda\, \_.\, \mathsf{Nat})\, b\, (\lambda\, \_\, n.\, \mathsf{suc}\, n)\, a\end{aligned}$$

With add implemented using the NatElim from our 2LTT model, we can implement multiplication $\mathsf{mul}\, a\, b$ in a similar manner, by eliminating a with zero as base case, and perform add b on each SUC-β reduction. Therefore, we can rewrite the example from section 1.1 into well-typed metaprogram in coherenece with our inference rules on Nats and lifting:

$$\begin{aligned}&\mathsf{add}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\ &\mathsf{add}_0 := \lambda\, a\, b.\, \mathsf{NatElim}_0\, (\lambda\, \_.\, \mathsf{Nat}_0)\, b\, (\lambda\, \_\, n.\, \mathsf{suc}_0\, n)\, a\end{aligned}$$

$$\begin{aligned}&\mathsf{mul}_1 : \mathsf{Nat}_1 \to \Uparrow\mathsf{Nat}_0 \to \Uparrow\mathsf{Nat}_0 \\ &\mathsf{mul}_1 := \lambda\, a\, b.\, \mathsf{NatElim}_1\, (\lambda\, \_.\, \Uparrow\mathsf{Nat}_0)\, \langle \mathsf{zero}_0 \rangle\, (\lambda\, \_\, n.\, \langle \mathsf{add}_0 \sim b \sim n \rangle)\, a \\ &\mathsf{two}_1 := \mathsf{suc}_1\, (\mathsf{suc}_1\, \mathsf{zero}_1)\end{aligned}$$

$$\begin{aligned}&\mathsf{double}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \\ &\mathsf{double}_0 := \lambda\, x.\, \sim(\mathsf{mul}_1\, \mathsf{two}_1\, \langle x \rangle)\end{aligned}$$

In a later section, we will stage this metaprogram rigorously through inference rules and get the same result from the introduction.

3.3 Lifting, Quoting, and Splicing

We now move on to the new addition 2LTT brings to the table, *lifting*, *quoting*, and *splicing*. Of all the inference rules we introduced so far, the premise and conclusion does not modify the index i that represents the stage. We are going to address that by introducing inference rules that allows that.

3.3.1 Lifting

We start with *lifting*. Syntactically, we define the annotation "$\Uparrow -$" ("$-$" is placeholder for specific type). Stage 0 types are the only valid arguments that can be used in place of "$-$". The behaviour of lifting is as follows:

$$\textsc{lift}\ \frac{\Gamma \vdash_{0,j} A}{\Gamma \vdash_{1,j} \Uparrow A} \qquad\qquad \textsc{lift-sub}\ \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash_{0,j} A}{\Gamma \vdash_{1,j} (\Uparrow A)[\sigma] = \Uparrow (A[\sigma])}$$

We see that given a type A in stage 0, universe level j, applying lift to A creates a new type $\Uparrow A$ in stage 1 of the same universe level.

Example 3.3.1 (Examples of Lifted Types)**.** $\Uparrow \mathsf{Nat}_0, \Uparrow (\Pi\ \mathsf{Nat}_0\ \mathsf{Nat}_0)$ are examples of types that are in stage 1.

Remark 3.3.1 (Lift and Native types are not the same). Both stage 0 and 1 has the type Nat, denoted $\mathsf{Nat}_0, \mathsf{Nat}_1$. And we have

$$\frac{\Gamma \vdash_{0,0} \mathsf{Nat}_0}{\Gamma \vdash_{1,0} \Uparrow \mathsf{Nat}_0} \qquad\qquad \frac{}{\Gamma \vdash_{1,0} \mathsf{Nat}_1}$$

However, $\Uparrow \mathsf{Nat}_0$ is *not* the same type as Nat_1

Example 3.3.2 (Not Lifted Type)**.** The type $(\Pi \Uparrow \mathsf{Nat}_0 \Uparrow \mathsf{Nat}_0)$ is not a lifted type. It is a Π-type with domain and co-domain being lifted types. We demonstrate this with the inference tree below:

$$\dfrac{\textsc{Lift}\ \dfrac{\textsc{Nat}\ \dfrac{}{\bullet \vdash_{0,0} \mathsf{Nat}_0}}{\bullet \vdash_{1,0} \Uparrow \mathsf{Nat}_0} \qquad \dfrac{\textsc{Lift}\ \dfrac{\textsc{nat}\ \dfrac{}{\bullet \vdash_{0,0} \mathsf{Nat}_0}}{\bullet \vdash_{1,0} \Uparrow \mathsf{Nat}_0}}{\bullet \triangleright \Uparrow \mathsf{Nat}_0 \vdash}\ \textsc{cxt-extension} \qquad \dfrac{}{\bullet \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{0,0} \mathsf{Nat}_0}\ \textsc{nat}}{\bullet \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{1,0} \Uparrow \mathsf{Nat}_0}\ \textsc{lift}}{\bullet \vdash_{1,0} \Pi \Uparrow \mathsf{Nat}_0 \Uparrow \mathsf{Nat}_0}\ \Pi$$

As we see at the bottom of the inference tree, the final step of inference is not Lift, so the type $\Pi, \Uparrow \mathsf{Nat}_0 \Uparrow \mathsf{Nat}_0$ is not a lifted type, but a stage 1 Π type.

Remark 3.3.2. We *do not* believe the following judgment is true:

$$\Gamma \vdash_{1,j} \Uparrow (\Pi\ A\ B) = \Pi \Uparrow A \Uparrow B$$

However, as we will show in later section, there is a way to transform a term of type $\Pi \Uparrow A \Uparrow B$ into a lifted type.

Remark 3.3.3 (Lift v.s. Stage Index). Let A be a type that exists in both stages, denoted A_0 and A_1. We emphasis $\Uparrow A_0 \neq A_1$.

In another word, a lifted type A from stage 0 is not equal to the equivalent to the same type A native to stage 1. This is a small but important detail.

Remark 3.3.4 (Lifted Types Do Not Have Eliminator). There is *no* general elimination rule for $\Uparrow A$.

Example 3.3.3. Take Bool_1 and $\Uparrow \mathsf{Bool}_0$. Where $\mathsf{Bool}_{1/0}$ are the respective boolean type of stage 1 and 0

Unlike Bool_1, $\Uparrow \mathsf{Bool}_0$ does not have eliminator. As a result, a function of type $\Pi \Uparrow \mathsf{Bool}_0\ \mathsf{Bool}_1$ would have to be a constant function, because there are no stage 1 eliminator that can inspect the internal of a term of type $\Uparrow \mathsf{Bool}_0$, and all eliminator eliminates a term to another term in the same stage.

One might attempt to counter with the following function:

$$f := \lambda b.\ \mathsf{if}\ {\sim}b\ \mathsf{then}\ \langle \mathsf{True}_0 \rangle\ \mathsf{else}\ \langle \mathsf{False}_0 \rangle$$

However, this function outputs terms of type $\Uparrow \mathsf{Bool}_0$, which as we mentioned in *remark* 3.3.2, is not the same type as Bool_1.

No eliminator for $\Uparrow A$ will restrict terms we can produce, and a trade-off for the program to behave in a predictable manner.

3.3.2 Quoting

We now introduce *quoting*, used to create a term of lifted type. Quoting is assigned the annotation $\langle - \rangle$, ("$-$" is placeholder for specific term). Its inference rule is as followed

$$\textsc{quote}\ \frac{\Gamma \vdash_{0,j} t : A}{\Gamma \vdash_{1,j} \langle t \rangle : \Uparrow A} \qquad \textsc{quote-sub}\ \frac{\Delta \vdash_{0,j} t : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash_{1,j} \langle t \rangle[\sigma] = \langle t[\sigma] \rangle : \Uparrow (A[\sigma])}$$

Given a term t of type A in stage 0, quoting t to $\langle t \rangle$ gives us a term of type $\Uparrow A$ in stage 1.

Example 3.3.4 (Examples of Quoted Terms)**.** Some examples of quoted terms are

- $\langle \mathsf{zero}_0 \rangle$ has type $\Uparrow \mathsf{Nat}_0$
- $\langle \lambda x.\mathsf{suc}\, x \rangle$ has type $\Uparrow (\mathsf{Nat}_0 \to \mathsf{Nat}_0)$

3.3.3 Splicing

Lastly we have *splicing*. Splicing has the annotation $\sim -$ ("$-$" is placeholder for specific term). Splice also has the highest precedence, even higher than function application. So $\sim f\, x$ parses to $(\sim f)\, x$. Splicing comes with the inference rule:

$$\textsc{splice}\ \frac{\Gamma \vdash_{1,j} t : \Uparrow A}{\Gamma \vdash_{0,j} \sim t : A}$$

Splice is used to bring a stage 0 term that has been lifted to stage 1 back down to 0.

Remark 3.3.5. We can quote any term t of type A if A is a type in stage 0. However, not every term s of type B in stage 1 can be spliced. For example, zero_1 cannot be spliced because zero_1 is of type Nat_1, and Nat_1 is not a lifted type.

3.3.4 Interaction Between Quoting and Splicing

There are two more inference rules that tells us $\langle - \rangle$ and $\sim -$ are inverses of each other:

$$\textsc{quote-splice}\ \frac{\Gamma \vdash_{1,j} t : \Uparrow A}{\Gamma \vdash_{1,j} \langle \sim t \rangle = t : \Uparrow A} \qquad \textsc{splice-quote}\ \frac{\Gamma \vdash_{0,j} t : A}{\Gamma \vdash_{0,j} \sim \langle t \rangle = t : A}$$

Example 3.3.5 (Elimination)**.** The term $\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}_0)$ is judgmentally the same Nat_0 as $\sim\langle \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}_0)\rangle$

This makes intuitive sense. If no computation is done in between lifting and splicing, we should expect to get back the same term.

One may ask why we have two annotations $\langle - \rangle$ and $\sim$ for terms, which are inverses of each other, but only one annotation, the $\Uparrow$ for types? To answer this, we consider the origin of 2LTT where it was intended to make meta-theoretical results of homotopy type theory internal [Ann+19]. We now understand stage 0 being the object-level and stage 1 being the meta-level. There is usually a way of representing object or concepts of the object-level in meta-level. However, with object-level usually being simpler, there are concepts in the meta-level which cannot be represented in the object-level. To make an analogy, $\mathbb{N}$ in meta-level might be defined as a subset of $\mathbb{R}$, while $\mathbb{N}$ in object-level is defined using Peano Arithmetic. In such situation, we cannot guarantee the correctness of bringing $\mathbb{N}$ from meta-level down to object level.

In fact, $\langle \sim - \rangle$ is not the identity function on all terms t in stage 1, but only terms like $\langle s \rangle$ that were lifted from stage 0. The inventors of 2LTT phrased it as such:

> One intuition for the two levels is as follows: from a type in HoTT [Homotopy Type Theory, the inner type theory], we can extract a statement that can be phrased in the meta-theory. From a meta-theoretical statement *about* HoTT, it is not always possible to construct a type. Thus, we can convert inner types into outer one, but not always vice versa [Ann+19].

4 Applying Judgments and Inference Rules

In this section, we provide examples that employ the inference rules as presented in the previous section. First, we apply the inference rules to derive the type of mul_1 under the assumption that add_0 has type $\mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0$. The type of add_0 is derived in the same manner since mul_1 and add_0 are both defined with NatElim. Second, we use the inference rules for lifting to prove the isomorphism between lifted function types and function types where the domain and codomain are lifted. This isomorphism property is useful as it can optimise the implementation of the 2LTT model.

4.1 Type Derivation for mul_1

Back to the definition of mul_1:

$$\mathsf{mul}_1 := \lambda\, a\, b.\, \mathsf{NatElim}_1\ (\lambda\, \_.\, \Uparrow \mathsf{Nat}_0)\ \langle \mathsf{zero}_0 \rangle\ (\lambda\, \_\ n.\, \langle \mathsf{add}_0\ {\sim}b\ {\sim}n \rangle)\ a$$

The ultimate goal for this section is to derive the judgment that mul_1 has the type $\mathsf{Nat}_1 \to \Uparrow \mathsf{Nat}_0 \to \Uparrow \mathsf{Nat}_0$ from the empty context. Since the definition for mul_1 uses the function add_0, we will assume the type of add_0. In other words, let's add the following inference rule.

$$\text{ADD}\ \frac{\Gamma \vdash_{00} a : \mathsf{Nat}_0 \qquad \Gamma \vdash_{00} b : \mathsf{Nat}_0}{\Gamma \vdash_{00} \mathsf{add}_0\ a\ b : \mathsf{Nat}_0}$$

Now, we have enough inference rules to derive the type of mul_1 from the empty context. To begin, we will derive the arguments for the function mul_1 through context extensions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet \vdash_{10} \mathsf{Nat}_1}\,\text{NAT-FORMATION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash}\,\text{CXT-EXTENSION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash_{00} \mathsf{Nat}_0}\,\text{NAT-FORMATION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0}\,\text{LIFT}}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash p : \bullet \triangleright \mathsf{Nat}_1}\,\text{SUB-FIRST-PROJ}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet \vdash_{10} \mathsf{Nat}_1}\,\text{NAT-FORMATION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash_{10} q : \mathsf{Nat}_1[p]}\,\text{SUB-SECOND-PROJ} \qquad \bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash p : \bullet \triangleright \mathsf{Nat}_1}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} q[p] : \mathsf{Nat}_1[p][p]}\,\text{TERM-SUB}}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} q[p] : \mathsf{Nat}_1[p \circ p]}\,\text{TYPE-COMP-SUB}}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} q[p] : \mathsf{Nat}_1}\,\text{NAT-SUB}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet \vdash_{10} \mathsf{Nat}_1}\,\text{NAT-FORMATION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash}\,\text{CXT-EXTENSION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash_{00} \mathsf{Nat}_0}\,\text{NAT-FORMATION}}{\bullet \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0}\,\text{LIFT}}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash q : \Uparrow \mathsf{Nat}_0}\,\text{SUB-SECOND-PROJ}$$

From the two derivations above, we obtain the terms $q[p] : \mathsf{Nat}_1$ and the term $q : \Uparrow \mathsf{Nat}_0$ under the context $\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0$. These will become the arguments to our function mul_1 in the form of De Bruijn indices, with first argument, $q[p] : \mathsf{Nat}_1$, being the target of our natural elimination. Notice our derivation also implies that $\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0$ is a context, thus we will denote this context as Γ for the rest of the section for brevity.

As per the definition of mul_1, it consists of terms to be passed to NatElim as arguments. We start with the derivation of $(\lambda\, \_.\Uparrow \mathsf{Nat}_0)$, which we will refer to as the motive.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash}{\Gamma \vdash_{10} \mathsf{Nat}_1}\,\text{NAT-FORMATION}}{\Gamma \triangleright \mathsf{Nat}_1 \vdash}\,\text{CXT-EXTENSION}}{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{00} \mathsf{Nat}_0}\,\text{NAT-FORMATION}}{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0}\,\text{LIFT}$$

Since we have derived $\Uparrow \mathsf{Nat}_0$ as our motive under the context $\Gamma \triangleright \mathsf{Nat}_1$, both the base and step should produce a $\Uparrow \mathsf{Nat}_0$ with some substitutions according to the inference rule NAT-ELIM. Thus, we continue our derivation with the base, $\langle \mathsf{zero}_0 \rangle$. We will reuse previously derived judgments as assumptions and apply equality judgments directly to our expressions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash}{\Gamma \vdash \mathsf{id} : \Gamma}\,\textsc{identity-sub} \qquad \dfrac{\dfrac{\Gamma \vdash}{\Gamma \vdash_{10} : \mathsf{zero}_1 : \mathsf{Nat}_1}\,\textsc{zero}}{\Gamma \vdash_{10} \mathsf{zero}_1 : \mathsf{Nat}_1[\mathsf{id}]}\,\textsc{type-id-sub}}{\Gamma \vdash (\mathsf{id}, \mathsf{zero}_1) : \Gamma \triangleright \mathsf{Nat}_1}\,\textsc{sub-extension} \qquad \dfrac{\dfrac{\Gamma \triangleright \mathsf{Nat}_1 \vdash}{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{00} \mathsf{zero}_0 : \mathsf{Nat}_0}\,\textsc{zero}}{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{10} \langle \mathsf{zero}_0 \rangle : \Uparrow \mathsf{Nat}_0}\,\textsc{lift}}{\Gamma \vdash_{10} \langle \mathsf{zero}_0 \rangle[\mathsf{id}, \mathsf{zero}_1] : (\Uparrow \mathsf{Nat}_0)[\mathsf{id}, \mathsf{zero}_1]}\,\textsc{term-sub}}{\Gamma \vdash_{10} \langle \mathsf{zero}_0[\mathsf{id}, \mathsf{zero}_1] \rangle : (\Uparrow \mathsf{Nat}_0)[\mathsf{id}, \mathsf{zero}_1]}\,\textsc{quote-sub}}{\Gamma \vdash_{10} : \langle \mathsf{zero}_0 \rangle : (\Uparrow \mathsf{Nat}_0)[\mathsf{id}, \mathsf{zero}_1]}\,\textsc{zero-sub}$$

When our target of elimination is $\mathsf{zero}_1 : \mathsf{Nat}_1$, the multiplication function should return $\langle \mathsf{zero}_0 \rangle$. Otherwise, the target is be nested with suc_1. Thus, to perform multiplication with our argument $\mathsf{q} : \Uparrow \mathsf{Nat}_0$, we add q on each step.

Since the output of the step function is a $(\Uparrow \mathsf{Nat}_0)[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])]$ as per the inference rule NAT-ELIM, we first simplify this type by deriving $\mathsf{Nat}_0[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])] = \mathsf{Nat}_0$. Next, as the step function uses add_0 according to the definition of mul_1, we derive the first argument of add_0 as $\mathsf{q} : \Uparrow \mathsf{Nat}_0$ in our first derivation. However, we need to shift its De Bruijn index as the context is extended for the step function. Lastly, we apply the inference rule ADD as described in the beginning of the section.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash \mathsf{p} : \Gamma \triangleright \mathsf{Nat}_1}\,\textsc{sub-first-proj} \quad \dfrac{\Gamma \vdash \mathsf{Nat}_1}{\Gamma \triangleright \mathsf{Nat}_1 \vdash \mathsf{p} : \Gamma}\,\textsc{sub-first-proj}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash \mathsf{p} \circ \mathsf{p} : \Gamma}\,\textsc{sub-composition} \quad \Gamma \vdash_{10} \mathsf{q} : \Uparrow \mathsf{Nat}_0}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q}[\mathsf{p} \circ \mathsf{p}] : (\Uparrow \mathsf{Nat}_0)[\mathsf{p} \circ \mathsf{p}]}\,\textsc{term-sub}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q}[\mathsf{p} \circ \mathsf{p}] : \Uparrow(\mathsf{Nat}_0[\mathsf{p} \circ \mathsf{p}])}\,\textsc{lift-sub}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q}[\mathsf{p} \circ \mathsf{p}] : \Uparrow \mathsf{Nat}_0}\,\textsc{nat-sub}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{00} \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) : \mathsf{Nat}_0}\,\textsc{splice}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q} : (\Uparrow \mathsf{Nat}_0)[\mathsf{p}]}\,\textsc{sub-second-proj}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q} : \Uparrow(\mathsf{Nat}_0[\mathsf{p}])}\,\textsc{lift-sub}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{q} : \Uparrow \mathsf{Nat}_0}\,\textsc{nat-sub}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{00} \sim\mathsf{q} : \mathsf{Nat}_0}\,\textsc{splice} \quad \Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{00} \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) : \mathsf{Nat}_0}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{00} \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} : \mathsf{Nat}_0}\,\textsc{add}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{00} \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} : \mathsf{Nat}_0[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])]}\,(1)}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle : \Uparrow(\mathsf{Nat}_0[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])])}\,\textsc{quote}}{\Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle : (\Uparrow \mathsf{Nat}_0)[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])]}\,\textsc{lift-sub}$$

Now we have derived all the arguments for NatElim, we follow the definition of mul_1 and derive the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \triangleright \mathsf{Nat}_1 \vdash_{10} \Uparrow \mathsf{Nat}_0 \quad \Gamma \vdash_{10} : \langle \mathsf{zero}_0 \rangle : (\Uparrow \mathsf{Nat}_0)[\mathsf{id}, \mathsf{zero}_1] \quad \Gamma \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle : (\Uparrow \mathsf{Nat}_0)[\mathsf{p} \circ \mathsf{p}, \mathsf{suc}(\mathsf{q}[\mathsf{p}])] \quad \Gamma \vdash_{10} \mathsf{q}[\mathsf{p}] : \mathsf{Nat}_1}{\Gamma \vdash_{10} \mathsf{NatElim} \Uparrow \mathsf{Nat}_0 \langle \mathsf{zero}_0 \rangle \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle \mathsf{q}[\mathsf{p}] : (\Uparrow \mathsf{Nat}_0)[\mathsf{id}, \mathsf{q}[\mathsf{p}]]}\,\textsc{nat-elim}}{\Gamma \vdash_{10} \mathsf{NatElim} \Uparrow \mathsf{Nat}_0 \langle \mathsf{zero}_0 \rangle \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle \mathsf{q}[\mathsf{p}] : \Uparrow(\mathsf{Nat}_0[\mathsf{id}, \mathsf{q}[\mathsf{p}]])}\,\textsc{lift-sub}}{\Gamma \vdash_{10} \mathsf{NatElim} \Uparrow \mathsf{Nat}_0 \langle \mathsf{zero}_0 \rangle \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle \mathsf{q}[\mathsf{p}] : \Uparrow \mathsf{Nat}_0}\,\textsc{nat-sub}}{\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0 \vdash_{10} \mathsf{NatElim} \Uparrow \mathsf{Nat}_0 \langle \mathsf{zero}_0 \rangle \langle \mathsf{add}_0 \sim(\mathsf{q}[\mathsf{p} \circ \mathsf{p}]) \sim\mathsf{q} \rangle \mathsf{q}[\mathsf{p}] : \Uparrow \mathsf{Nat}_0}$$

In the last step of the derivation above, we apply the assumption that the context Γ denotes $\bullet \triangleright \mathsf{Nat}_1 \triangleright \Uparrow \mathsf{Nat}_0$ since the beginning of the section. The end result of our derivation shows that, for an $a : \mathsf{Nat}_1$ and $b : \Uparrow \mathsf{Nat}_0$, the expression $\mathsf{NatElim} \Uparrow \mathsf{Nat}_0 \langle \mathsf{zero}_0 \rangle \langle \mathsf{add}_0 \sim b \sim\mathsf{q} \rangle \rangle\ a$ has the type $\Uparrow \mathsf{Nat}_0$, where q is an argument in the step function. In other words, the step function is $\lambda \_. \lambda\, n. \langle \mathsf{add}_0 \sim b \sim n \rangle$. Ultimately, our derived expression matches the definition of mul_1, thus mul_1 has the

type $\mathsf{Nat}_1 \to \Uparrow\mathsf{Nat}_0 \to \Uparrow\mathsf{Nat}_0$ as wanted. Therefore, we can now use the derivation to stage the metaprogram $\mathsf{double}_0 := \lambda\, x.\sim(\mathsf{mul}_1\ (\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1))\ \langle x\rangle)$ through the equality judgments of NatElim, which will result the same output as described in the introduction.

4.2 Staging double_0 Formally

Bringing back the definition of double_0:

$$\begin{aligned}&\mathsf{double}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0\\ &\mathsf{double}_0 := \lambda\, x.\sim(\mathsf{mul}_1\ (\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1))\ \langle x\rangle)\end{aligned}$$

Based on the definition above, we substitute $\mathsf{q}[\mathsf{p}]$ with the term $\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1) : \mathsf{Nat}_1$ and $\mathsf{q}[\mathsf{p}\circ\mathsf{p}]$ with $\langle x\rangle : \Uparrow\mathsf{Nat}_0$. Thus, applying the inference rules, we stage the program as follows:

$$\begin{aligned}
&\lambda\, x.\sim(\mathsf{mul}_1\ (\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1))\ \langle x\rangle) && \\
=\ &\lambda\, x.\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\sim\langle x\rangle\sim\mathsf{q}\rangle\ (\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1))) && (\text{definition of } \mathsf{mul}_1)\\
=\ &\lambda\, x.\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\ x\sim\mathsf{q}\rangle\ (\mathsf{suc}_1\ (\mathsf{suc}_1\ \mathsf{zero}_1))) && (\textsc{splice-quote})\\
=\ &\lambda\, x.\sim\langle\mathsf{add}_0\ x\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\ x\sim\mathsf{q}\rangle\ (\mathsf{suc}_1\ \mathsf{zero}_1))\rangle && (\textsc{suc}\text{-}\beta)\\
=\ &\lambda\, x.\mathsf{add}_0\ x\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\ x\sim\mathsf{q}\rangle\ (\mathsf{suc}_1\ \mathsf{zero}_1)) && (\textsc{splice-quote})\\
=\ &\lambda\, x.\mathsf{add}_0\ x\sim\langle\mathsf{add}_0\ x\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\ x\sim\mathsf{q}\rangle\ \mathsf{zero}_1)\rangle && (\textsc{suc}\text{-}\beta)\\
=\ &\lambda\, x.\mathsf{add}_0\ x\ (\mathsf{add}_0\ x\sim(\mathsf{NatElim}\ \Uparrow\mathsf{Nat}_0\ \langle\mathsf{zero}_0\rangle\ \langle\mathsf{add}_0\ x\sim\mathsf{q}\rangle\ \mathsf{zero}_1)) && (\textsc{splice-quote})\\
=\ &\lambda\, x.\mathsf{add}_0\ x\ (\mathsf{add}_0\ x\sim\langle\mathsf{zero}_0\rangle) && (\textsc{zero}\text{-}\beta)\\
=\ &\lambda\, x.\mathsf{add}_0\ x\ (\mathsf{add}_0\ x\ \mathsf{zero}_0) && (\textsc{splice-quote})
\end{aligned}$$

5 Discussion

We have completed our main objective of applying and type-checking our metaprogram double_0. Now we will discuss some properties of 2LTT that can serve as direction for further exploration.

5.1 Isomorphism Property of Lifting and Quoting

In mathematics, *isomorphism* describes a bijective function that preserves "properties". The properties an isomorphism preserves can differ from context to context.

Definition 5.1.1 (Isomorphic)**.** If there is an isomorphism between two sets A, B, we call these two sets *isomorphic*.

We do not believe $\Uparrow(\Pi\, A\, B)$ is the same type as $\Pi\ (\Uparrow A)\ (\Uparrow B)$. However, it is possible to show that there two types are similar enough that we can consider them to be isomorphic.

To set up the demonstration, we first define two functions.

Definition 5.1.2 (Transformation between $\Uparrow((x : A) \to B\, x)$ and $(x : \Uparrow A) \to \Uparrow(B \sim x)$)**.** We define two functions:

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

Remark 5.1.1 ($pres_{\to}$ and $pres_{\to}^{-1}$ don't modify the input function). We now demonstrate $pres_{\to}$ and $pres_{\to}^{-1}$ does not modify the input function's behaviour. We do so by showing it is possible to recover the original function through an inverse operation.

Let $f : (x : \Uparrow A) \to \Uparrow(B \sim x)$. Consider the following reduction:

$$
\begin{aligned}
pres_{\to}(pres_{\to}^{-1} f) &= (\lambda f.\ \lambda x.\ \langle \sim f\ \sim x\rangle)\ ((\lambda f.\ \langle \lambda x.\ \sim(f\ \langle x\rangle)\rangle)\ f) \\
&= (\lambda f.\ \lambda x.\ \langle \sim f\ \sim x\rangle)\ \langle \lambda x.\ \sim(f\ \langle x\rangle)\rangle && (\beta\text{-reduction}) \\
&= \lambda x.\ \langle \sim\langle \lambda x.\ \sim(f\ \langle x\rangle)\rangle\ \sim x\rangle && (\beta\text{-reduction}) \\
&= \lambda x.\ \langle ((\lambda x.\ \sim(f\ \langle x\rangle))\ \sim x\rangle && (\textsc{splice-quote}) \\
&= \lambda x.\ \langle \sim(f\ \langle \sim x\rangle)\rangle && (\beta\text{-reduction}) \\
&= \lambda x.\ \langle \sim(f\ x)\rangle && (\textsc{quote-splice}) \\
&= \lambda x.\ (f\ x) && (\textsc{quote-splice}) \\
&= f && (\text{Function } \eta\text{-reduction})
\end{aligned}
$$

Now we are ready to demonstrate that the lifting process possesses isomorphic behaviour. For those more comfortable with mathematics, and isomorphism ϕ often satisfies the following structure:

$$\phi(f\,a) = \phi(f)\phi(a)$$

Ideally, we would define for all x, $\phi(x) = \langle x\rangle$, and show $\langle f\ a\rangle = \langle b\rangle = \langle f\rangle\ \langle a\rangle$ to match the structure. However, because the expression $(\langle f\rangle\ \langle a\rangle)$ is undefined due to type mismatch, we achieve a similar structure by $pres_{\to}$ to $\langle f\rangle$ first, which doesn't change f's behaviour as shown. With that, we will show the lifting process possesses properties of isomorphism.

Example 5.1.1 (The effect of lifting)**.** Let $f : (x : A) \to B\,x$, and $a : A, b : (B\,a)$ such that $f\ a = b$, then we have the following reduction:

$$
\begin{aligned}
(pres_{\to}\langle f\rangle)\langle a\rangle &= ((\lambda f\,x.\ \langle \sim f\ \sim x\rangle)\ \langle f\rangle)\ \langle a\rangle \\
&= (\lambda x.\ \langle \sim\langle f\rangle\ \sim x\rangle)\ \langle a\rangle \\
&= \langle \sim\langle f\rangle\ \sim\langle a\rangle\rangle \\
&= \langle f\ a\rangle \\
&= \langle b\rangle
\end{aligned}
$$

6 Conclusion

We introduced staged compilation, more specifically, two-stage compilation as a useful way to write metaprograms that generate code with safety. With the special operations for moving between stages (namely lifting, quoting, and splicing) integrated to the typing rules, the 2LTT model provides not only guarantees the well-formedness of the generated output, but also support for dependent types on both stages. While this paper only covered the typing rules for type-checking a metaprogram, the staging algorithm (also known as the substitution calculus) can alse be formalised into typing rules through categorical logic (also known as abstract nonsense logic).

References

[TS97] Walid Taha and Tim Sheard. "MetaML and Multi-Stage Programming with Explicit Annotations". In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: https://dl.acm.org/doi/10.1145/258994.259019.

[Ann+19] Danil Annenkov et al. "Two-Level Type Theory and Applications". In: *ArXiv e-prints* (May 2019). URL: http://arxiv.org/abs/1705.03307.

[Dev23] .NET Developers. *Language Integrated Query (LINQ).* 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (visited on 07/18/2023).