Staged Compilation With Two-Level Type Theory

ANDRÁS KOVÁCS, Eötvös Loránd University, Hungary

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

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

ACM Reference Format:
András Kovács. 2018. Staged Compilation With Two-Level Type Theory. *J. ACM* 37, 4, Article 111 (August 2018), 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

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

Two-level type theory, or 2LTT in short, was described by Annekov, Capriotti, Kraus and Sattler [1], building on ideas from Vladimir Voevodsky [3]. The motivation was to allow convenient metatheoretical reasoning about a certain mathematical language (homotopy type theory), and to enable concise and modular ways to extend the language with axioms.

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

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

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

1.1 Contributions

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

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

2 A TOUR OF TWO-LEVEL TYPE THEORY

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

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

2.1 Rules of 2LTT

Universes. We have universes $\mathsf{U}_{i,j}$, where $i \in \{0, 1\}$, and $j \in \mathbb{N}$. The i index denotes stages, where 0 is the runtime (object-level) stage, and 1 is the compile time (meta-level) stage. The j index denotes universe sizes in the usual sense of type theory. We assume Russell-style universes, with $\mathsf{U}_{i,j} : \mathsf{U}_{i,j+1}$. However, for the sake of brevity we will usually omit the j indices in this section, and simply write U_0 or U_1.

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

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

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

It is not required that we have the *same* type formers at both stages. As we will see in Section [**?**], simpler object languages have the advantage that they are easier to process during downstream compilation.

Moving between stages. At this point, our system is rather limited, since there is no interaction between the stages. We add such interaction via the follow three 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 to 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 computes to t.
- *Splicing:* for $A : \mathsf{U}_0$ and $t : \Uparrow A$, we have $\sim t : A$. During staging, the metaprogram in the splice is executed, and the resulting expression is inserted into the output.

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

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

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

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

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

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

2.2 Staged Programming in 2LTT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then we define the interpretation functions:

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

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

2.3 Properties of Lifting

preservation properties

stage inference

2.4 Restricting the Object Language

monomorphization

memory layout control

typed closures

2.5 Fusion

2LTT is also a natural setting for a wide range of *fusion optimizations*. We only describe here a simple example, *foldr-build fusion* for lists, which is notably used in GHC Haskell [?]. The idea is the following: we want to eliminate intermediate lists, e.g. fuse repeated mapping to a single traversal. For this, we exploit that there is an isomorphic *meta-level* representation for runtime lists which supports more definitional equations. Starting from $\Uparrow(\mathsf{List}_0\,A)$, we can use Böhm-Berarducci encoding under the lifting to get

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

However, $\Uparrow$ preserves function types, so the above is definitionally isomorphic to

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

Now, we may abbreviate the above type as $\mathsf{BBList}\,A$. We write fusible functions on $\mathsf{BBList}\,A$, and when we want to fuse a definition, we convert from $\Uparrow(\mathsf{List}_0\,A)$ to $\mathsf{BBList}\,A$, apply operations on that representation, then convert back in the end.

Now, in 2LTT it is not provable that Böhm-Berarducci encoding is an isomorphism [?], so it certainly does not hold that back-and forth conversion between the two representations is definitionally the identity function. In GHC Haskell, this equality is added as a *rewrite rule*. This makes it possible to hide the back-and-forth conversions from library users, since inverse conversions will hopefully cancel out and cause no additional overhead. A similar mechanism could be used in 2LTT as well. However, such implicit conversions are not used in most languages which support fusion. For example, iterators in the Rust language [?], which use a form of stream fusion, require explicit conversion to iterators.

We may compare GHC and 2LTT with respect to the implementation of fusion. In 2LTT, fusion happens through meta-level β-reduction, which is formally guaranteed to go through, and which can be also computed very efficiently. In GHC, all of the necessary computation is realized by rewrite rules and general-purpose optimization passes. This is in practice very fragile and non-composable. It requires carefully tuning every individual collection of rewrite rules, and also ordering rule applications in specific ways, while keeping details of the compiler optimization passes in mind. And when fusion fails to go through, it is usually a performance *pessimization*.

2.6 Properties of Lifting

2.7 Informal Syntax

3 VARIATIONS & APPLICATIONS

3.1 Fusion

3.2 Monomorphization

3.3 Levity Polymorphism

3.4 Typed Closures

4 FORMAL SYNTAX & MODELS

4.1 Metatheory

4.2 Models

5 STAGING BY EVALUATION

5.1 Presheaf Model of 2LTT

5.2 Closed staging

5.3 Yoneda, representability, intensional analysis

6 STABILITY

6.1 Open staging

7 SOUNDNESS

7.1 Internal Language & Features

7.2 The Logical Relation

7.3 Externalization & Soundness

8 RELATED WORK

Igarashi, Kiselyov et al, MetaML, MetaOCaml, Carette, TH, Scala ppl (?) PE lit ? 2LTT, Voevodsky,

9 FUTURE WORK & CONCLUSIONS

REFERENCES

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

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

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