

Staged Compilation with Two-Level Type Theory

András Kovács

Eötvös Loránd University

12 September 2022, ICFP, Ljubljana

Staged Compilation

Staged compilation is about writing code-generating code with good ergonomics and safety guarantees.

Staged Compilation

Staged compilation is about writing code-generating code with good ergonomics and safety guarantees.

Examples:

- (Typed) Template Haskell.
- C++ templates.
- Rust traits, macros & generics.

Staged Compilation

Staged compilation is about writing code-generating code with good ergonomics and safety guarantees.

Examples:

- (Typed) Template Haskell.
- C++ templates.
- Rust traits, macros & generics.

Motivations:

- Low-cost abstraction.
- DSLs.
- Inlining & fusion with strong guarantees.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.
- ② Guaranteed well-typing of code output, guaranteed well-staging.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.
- ② Guaranteed well-typing of code output, guaranteed well-staging.
- ③ Supports a wide range of runtime and meta-languages.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.
- ② Guaranteed well-typing of code output, guaranteed well-staging.
- ③ Supports a wide range of runtime and meta-languages.
 - Including dependent types.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.
- ② Guaranteed well-typing of code output, guaranteed well-staging.
- ③ Supports a wide range of runtime and meta-languages.
 - Including dependent types.
- ④ Supports efficient *staging-by-evaluation*.

Two-Level Type Theory (2LTT)

Comes from **homotopy type theory**:

- *Voevodsky: A simple type system with two identity types.*
- *Annekov, Capriotti, Kraus, Sattler: Two-Level Type Theory and Applications.*
- Motivation: meta-programming and modular axioms for HoTT.

2LTT is directly applicable to two-stage compilation.

Features:

- ① Integrates a compile-time (“meta”) language and a runtime language.
- ② Guaranteed well-typing of code output, guaranteed well-staging.
- ③ Supports a wide range of runtime and meta-languages.
 - Including dependent types.
- ④ Supports efficient *staging-by-evaluation*.

This talk

This talk mostly contains **small programming examples**.

This talk

This talk mostly contains **small programming examples**.

For a **tutorial** and **larger programming examples**, see the artifact.

This talk

This talk mostly contains **small programming examples**.

For a **tutorial** and **larger programming examples**, see the artifact.

For **formal details**, see the paper.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- ② No elimination is allowed from one universe to the other.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- ② No elimination is allowed from one universe to the other.
- ③ *Lifting*: for $A : U_0$, we have $\uparrow A : U_1$.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- ② No elimination is allowed from one universe to the other.
- ③ *Lifting*: for $A : U_0$, we have $\uparrow A : U_1$.
- ④ *Quoting*: for $A : U_0$ and $t : A$, we have $\langle t \rangle : \uparrow A$.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- ② No elimination is allowed from one universe to the other.
- ③ *Lifting*: for $A : U_0$, we have $\uparrow A : U_1$.
- ④ *Quoting*: for $A : U_0$ and $t : A$, we have $\langle t \rangle : \uparrow A$.
- ⑤ *Splicing*: for $t : \uparrow A$, we have $\sim t : A$.

Rules of 2LTT

- ① Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- ② No elimination is allowed from one universe to the other.
- ③ *Lifting*: for $A : U_0$, we have $\uparrow A : U_1$.
- ④ *Quoting*: for $A : U_0$ and $t : A$, we have $\langle t \rangle : \uparrow A$.
- ⑤ *Splicing*: for $t : \uparrow A$, we have $\sim t : A$.
- ⑥ $\langle \sim t \rangle \equiv t$ and $\sim \langle t \rangle \equiv t$.

Rules of 2LTT

- 1 Two universes U_0 , U_1 , closed under arbitrary type formers.
 - U_0 is the universe of runtime types.
 - U_1 is the universe of meta-level types.
- 2 No elimination is allowed from one universe to the other.
- 3 *Lifting*: for $A : U_0$, we have $\uparrow A : U_1$.
- 4 *Quoting*: for $A : U_0$ and $t : A$, we have $\langle t \rangle : \uparrow A$.
- 5 *Splicing*: for $t : \uparrow A$, we have $\sim t : A$.
- 6 $\langle \sim t \rangle \equiv t$ and $\sim \langle t \rangle \equiv t$.

Staging runs all metaprograms in splices and inserts their result in the code output.

Inlined definitions

Staging input:

$$\text{two} : \uparrow\uparrow \text{Nat}_0$$
$$\text{two} = \langle \text{suc}_0 (\text{suc}_0 \text{zero}_0) \rangle$$
$$f : \text{Nat}_0 \rightarrow \text{Nat}_0$$
$$f = \lambda x. x + \sim \text{two}$$

Inlined definitions

Staging input:

$$\text{two} : \uparrow\uparrow \text{Nat}_0$$
$$\text{two} = \langle \text{suc}_0 (\text{suc}_0 \text{zero}_0) \rangle$$
$$f : \text{Nat}_0 \rightarrow \text{Nat}_0$$
$$f = \lambda x. x + \sim \text{two}$$

Output:

$$f : \text{Nat}_0 \rightarrow \text{Nat}_0$$
$$f = \lambda x. x + \text{suc}_0 (\text{suc}_0 \text{zero}_0)$$

Compile-time identity function

Input:

$$\text{id} : (A : U_1) \rightarrow A \rightarrow A$$
$$\text{id} = \lambda A x. x$$
$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$
$$\text{idBool}_0 = \lambda x. \sim(\text{id} (\uparrow \text{Bool}_0) \langle x \rangle)$$

Compile-time identity function

Input:

$$\text{id} : (A : U_1) \rightarrow A \rightarrow A$$
$$\text{id} = \lambda A x. x$$
$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$
$$\text{idBool}_0 = \lambda x. \sim(\text{id} (\uparrow \text{Bool}_0) \langle x \rangle)$$

Output:

$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$
$$\text{idBool}_0 = \lambda x. x$$

An alternative identity function

Input:

$$\text{id}_{\uparrow} : (A : \uparrow U_0) \rightarrow \uparrow \sim A \rightarrow \uparrow \sim A$$

$$\text{id}_{\uparrow} = \lambda A x. x$$

$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$

$$\text{idBool}_0 = \lambda x. \sim(\text{id}_{\uparrow} \langle \text{Bool}_0 \rangle \langle x \rangle)$$

An alternative identity function

Input:

$$\text{id}_{\uparrow} : (A : \uparrow U_0) \rightarrow \uparrow \sim A \rightarrow \uparrow \sim A$$

$$\text{id}_{\uparrow} = \lambda A x. x$$

$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$

$$\text{idBool}_0 = \lambda x. \sim(\text{id}_{\uparrow} \langle \text{Bool}_0 \rangle \langle x \rangle)$$

Note that

$$A : \uparrow U_0$$

$$\sim A : U_0$$

$$\uparrow \sim A : U_1$$

An alternative identity function

Input:

$$\text{id}_{\uparrow} : (A : \uparrow U_0) \rightarrow \uparrow \sim A \rightarrow \uparrow \sim A$$

$$\text{id}_{\uparrow} = \lambda A x. x$$

$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$

$$\text{idBool}_0 = \lambda x. \sim(\text{id}_{\uparrow} \langle \text{Bool}_0 \rangle \langle x \rangle)$$

Note that

$$A : \uparrow U_0$$

$$\sim A : U_0$$

$$\uparrow \sim A : U_1$$

Output:

$$\text{idBool}_0 : \text{Bool}_0 \rightarrow \text{Bool}_0$$

$$\text{idBool}_0 = \lambda x. x$$

map with inlining

Input:

$$\text{inlMap} : \{A\ B : \uparrow U_0\} \rightarrow (\uparrow \sim A \rightarrow \uparrow \sim B) \rightarrow \uparrow(\text{List}_0 \sim A) \rightarrow \uparrow(\text{List}_0 \sim B)$$
$$\text{inlMap} = \lambda f\ as. \langle \text{foldr}_0 (\lambda a\ bs. \text{cons}_0 \sim(f\ \langle a \rangle) bs) \text{nil}_0 \sim as \rangle$$
$$f : \text{List}_0 \text{Nat}_0 \rightarrow \text{List}_0 \text{Nat}_0$$
$$f = \lambda xs. \sim(\text{inlMap} (\lambda n. \langle \sim n + 2 \rangle) \langle xs \rangle)$$

map with inlining

Input:

$$\text{inlMap} : \{A\ B : \uparrow U_0\} \rightarrow (\uparrow \sim A \rightarrow \uparrow \sim B) \rightarrow \uparrow(\text{List}_0 \sim A) \rightarrow \uparrow(\text{List}_0 \sim B)$$
$$\text{inlMap} = \lambda f\ as. \langle \text{foldr}_0 (\lambda a\ bs. \text{cons}_0 \sim(f\ \langle a \rangle) bs) \text{nil}_0 \sim as \rangle$$
$$f : \text{List}_0 \text{Nat}_0 \rightarrow \text{List}_0 \text{Nat}_0$$
$$f = \lambda xs. \sim(\text{inlMap} (\lambda n. \langle \sim n + 2 \rangle) \langle xs \rangle)$$

Output:

$$f : \text{List}_0 \text{Nat}_0 \rightarrow \text{List}_0 \text{Nat}_0$$
$$f = \lambda xs. \text{foldr}_0 (\lambda a\ bs. \text{cons}_0 (a + 2) bs) \text{nil}_0 xs$$

Inference for staging operations

Most quotes and splices are inferable with **bidirectional elaboration** & **coercive subtyping**.

Inference for staging operations

Most quotes and splices are inferable with **bidirectional elaboration** & **coercive subtyping**.

$$\text{inlMap} : \{A\ B : \uparrow U_0\} \rightarrow (\uparrow A \rightarrow \uparrow B) \rightarrow \uparrow(\text{List}_0 A) \rightarrow \uparrow(\text{List}_0 B)$$
$$\text{inlMap} = \lambda f. \text{foldr}_0 (\lambda a\ bs. \text{cons}_0 (f\ a)\ bs) \text{nil}_0$$
$$f : \text{List}_0 \text{Nat}_0 \rightarrow \text{List}_0 \text{Nat}_0$$
$$f = \text{inlMap} (\lambda n. n + 2)$$

Staged types

Input:

$$\text{Vec} : \text{Nat}_1 \rightarrow \uparrow\uparrow U_0 \rightarrow \uparrow\uparrow U_0$$

$$\text{Vec zero}_1 \quad A = \langle T_0 \rangle$$

$$\text{Vec (suc}_1 n) A = \langle \sim A \times \sim(\text{Vec } n A) \rangle$$

$$\text{Tuple3} : U_0 \rightarrow U_0$$

$$\text{Tuple3 } A = \sim(\text{Vec } 3 \langle A \rangle)$$

Staged types

Input:

$$\text{Vec} : \text{Nat}_1 \rightarrow \uparrow\uparrow U_0 \rightarrow \uparrow\uparrow U_0$$

$$\text{Vec zero}_1 \quad A = \langle T_0 \rangle$$

$$\text{Vec (suc}_1 n) A = \langle \sim A \times \sim(\text{Vec } n A) \rangle$$

$$\text{Tuple3} : U_0 \rightarrow U_0$$

$$\text{Tuple3 } A = \sim(\text{Vec } 3 \langle A \rangle)$$

Output:

$$\text{Tuple3} : U_0 \rightarrow U_0$$

$$\text{Tuple3 } A = A \times (A \times (A \times T_0))$$

map for Vec

Input:

$$\begin{aligned}\text{map} : \{A B : \uparrow U_0\} &\rightarrow (n : \text{Nat}_1) \rightarrow (\uparrow \sim A \rightarrow \uparrow \sim B) \\ &\rightarrow \uparrow(\text{Vec } n A) \rightarrow \uparrow(\text{Vec } n B)\end{aligned}$$

$$\text{map zero}_1 \quad f \text{ as} = \langle \text{tt}_0 \rangle$$

$$\text{map}(\text{suc}_1 n) f \text{ as} = \langle (\sim(f \langle \text{fst}_0 \sim \text{as} \rangle), \sim(\text{map } n f \langle \text{snd}_0 \sim \text{as} \rangle)) \rangle$$

$$f : \sim(\text{Vec } 2 \langle \text{Nat}_0 \rangle) \rightarrow \sim(\text{Vec } 2 \langle \text{Nat}_0 \rangle)$$

$$f \text{ xs} = \sim(\text{map } 2 (\lambda x. \langle \sim x + 2 \rangle) \langle \text{xs} \rangle)$$

map for Vec

Input:

$$\begin{aligned}\text{map} : \{A\ B : \uparrow U_0\} &\rightarrow (n : \text{Nat}_1) \rightarrow (\uparrow \sim A \rightarrow \uparrow \sim B) \\ &\rightarrow \uparrow(\text{Vec } n\ A) \rightarrow \uparrow(\text{Vec } n\ B)\end{aligned}$$

$$\text{map zero}_1\ f\ as = \langle \text{tt}_0 \rangle$$

$$\text{map}(\text{suc}_1\ n)\ f\ as = \langle (\sim(f\ \langle \text{fst}_0\ \sim as \rangle), \sim(\text{map } n\ f\ \langle \text{snd}_0\ \sim as \rangle)) \rangle$$

$$f : \sim(\text{Vec } 2\ \langle \text{Nat}_0 \rangle) \rightarrow \sim(\text{Vec } 2\ \langle \text{Nat}_0 \rangle)$$

$$f\ xs = \sim(\text{map } 2\ (\lambda x. \langle \sim x + 2 \rangle)\ \langle xs \rangle)$$

Output:

$$f : \text{Nat}_0 \times (\text{Nat}_0 \times \top_0) \rightarrow \text{Nat}_0 \times (\text{Nat}_0 \times \top_0)$$

$$f\ xs = (\text{fst}_0\ xs + 2, (\text{fst}_0\ (\text{snd}_0\ xs) + 2, \text{tt}_0))$$

More things

In the artifact:

- Staged foldr/build fusion.
- Well-typed staged STLC interpreter.
- Monadic let-insertion.

More things

In the artifact:

- Staged foldr/build fusion.
- Well-typed staged STLC interpreter.
- Monadic let-insertion.

In the paper:

- **Staging is:** evaluation of 2LTT syntax in presheaves over the object-theory syntax.
- **Correctness of staging is:** strong conservativity of 2LTT over the object theory.
- Correctness is shown by proof-relevant logical relations, internally to the mentioned presheaf category.

Thank you!