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 ("object") 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 ("object") 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 ("object") 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 ("object") 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 ("object") 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 (object-level) types.
   - U_1 is the universe of compile-time (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 (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.

Rules of 2LTT

1. Two universes U_0, U_1, closed under arbitrary type formers.
   - U_0 is the universe of runtime (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.
3. *Lifting:* for $\mathsf{A} : \mathsf{U}_0$, we have $\Uparrow \mathsf{A} : \mathsf{U}_1$.

Rules of 2LTT

1. Two universes U_0, U_1, closed under arbitrary type formers.
   - U_0 is the universe of runtime (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.
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 (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.
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 (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.
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 (object-level) types.
   - U_1 is the universe of compile-time (meta-level) types.
2. All type/term formers and eliminators stay within the same universe.
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:

$$\text{two} : \Uparrow \text{Nat}_0$$
$$\text{two} = <\text{suc}_0\,(\text{suc}_0\,\text{zero}_0)>$$

$$\text{f} : \text{Nat}_0 \to \text{Nat}_0$$
$$\text{f} = \lambda\,\text{x}.\,\text{x} + \sim\text{two}$$

Inlined definitions

Staging input:

$$\begin{aligned}&\mathsf{two} : \Uparrow \mathsf{Nat}_0\\&\mathsf{two} = <\mathsf{suc}_0\,(\mathsf{suc}_0\,\mathsf{zero}_0)>\end{aligned}$$

$$\begin{aligned}&\mathsf{f} : \mathsf{Nat}_0 \to \mathsf{Nat}_0\\&\mathsf{f} = \lambda\,\mathsf{x}.\,\mathsf{x} + {\sim}\mathsf{two}\end{aligned}$$

Output:

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

Compile-time identity function

Input:

$$\mathsf{id} : (\mathsf{A} : \mathsf{U}_1) \to \mathsf{A} \to \mathsf{A}$$
$$\mathsf{id} = \lambda\, \mathsf{A}\, \mathsf{x}.\, \mathsf{x}$$

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, {\sim}(\mathsf{id}\ (\Uparrow \mathsf{Bool}_0)\ {<}\mathsf{x}{>})$$

Compile-time identity function

Input:

$$\mathsf{id} : (\mathsf{A} : \mathsf{U}_1) \to \mathsf{A} \to \mathsf{A}$$
$$\mathsf{id} = \lambda\, \mathsf{A}\, \mathsf{x}.\, \mathsf{x}$$

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, {\sim}(\mathsf{id}\ (\Uparrow \mathsf{Bool}_0)\ {<}\mathsf{x}{>})$$

Output:

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, \mathsf{x}$$

An alternative identity function

Input:

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

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, \sim(\mathsf{id}_{\Uparrow} <\mathsf{Bool}_0> <\mathsf{x}>)$$

An alternative identity function

Input:

$$\mathsf{id}_{\Uparrow} : (A : \Uparrow U_0) \to \Uparrow \sim A \to \Uparrow \sim A$$
$$\mathsf{id}_{\Uparrow} = \lambda\, A\, x.\, x$$

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, x.\, \sim(\mathsf{id}_{\Uparrow} <\mathsf{Bool}_0> <x>)$$

Note that

$$A : \Uparrow U_0$$
$$\sim A : U_0$$
$$\Uparrow \sim A : U_1$$
$$<x> : \Uparrow \mathsf{Bool}_0$$
$$<x> : \Uparrow \sim <\mathsf{Bool}_0>$$

An alternative identity function

Input:

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

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, \sim(\mathsf{id}_{\Uparrow} <\mathsf{Bool}_0> <\mathsf{x}>)$$

Note that

$$\begin{aligned} \mathsf{A} &: \Uparrow \mathsf{U}_0 \\ \sim \mathsf{A} &: \mathsf{U}_0 \\ \Uparrow \sim \mathsf{A} &: \mathsf{U}_1 \\ <\mathsf{x}> &: \Uparrow \mathsf{Bool}_0 \\ <\mathsf{x}> &: \Uparrow \sim <\mathsf{Bool}_0> \end{aligned}$$

Output:

$$\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0$$
$$\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, \mathsf{x}$$

map with inlining

Input:

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

$$\mathsf{f} : \mathsf{List}_0\ \mathsf{Nat}_0 \to \mathsf{List}_0\ \mathsf{Nat}_0$$
$$\mathsf{f} = \lambda\, \mathsf{xs}.\ \sim(\mathsf{inlMap}\ (\lambda\, \mathsf{n}.\ <\sim \mathsf{n} + 2>)\ <\mathsf{xs}>)$$

map with inlining

Input:

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

$$\begin{aligned}
&\mathsf{f} : \mathsf{List}_0\ \mathsf{Nat}_0 \to \mathsf{List}_0\ \mathsf{Nat}_0 \\
&\mathsf{f} = \lambda\, \mathsf{xs}.\ \sim(\mathsf{inlMap}\ (\lambda\, \mathsf{n}.\ <\sim \mathsf{n} + 2>)\ <\mathsf{xs}>)
\end{aligned}$$

Output:

$$\begin{aligned}
&\mathsf{f} : \mathsf{List}_0\ \mathsf{Nat}_0 \to \mathsf{List}_0\ \mathsf{Nat}_0 \\
&\mathsf{f} = \lambda\, \mathsf{xs}.\ \mathsf{foldr}_0\ (\lambda\, \mathsf{a}\ \mathsf{bs}.\ \mathsf{cons}_0\ (\mathsf{a} + 2)\ \mathsf{bs})\ \mathsf{nil}_0\ \mathsf{xs}
\end{aligned}$$

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**.

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

$$\begin{aligned}
&\mathsf{f} : \mathsf{List}_0\ \mathsf{Nat}_0 \to \mathsf{List}_0\ \mathsf{Nat}_0 \\
&\mathsf{f} = \mathsf{inlMap}\ (\lambda\, \mathsf{n}.\, \mathsf{n} + 2)
\end{aligned}$$

Staged types

Input:

$$\mathsf{Vec} : \mathsf{Nat}_1 \to \Uparrow \mathsf{U}_0 \to \Uparrow \mathsf{U}_0$$
$$\mathsf{Vec}\ \mathsf{zero}_1 \quad \mathsf{A} = <\top_0>$$
$$\mathsf{Vec}\ (\mathsf{suc}_1\ \mathsf{n})\ \mathsf{A} = <\sim \mathsf{A} \times \sim(\mathsf{Vec}\ \mathsf{n}\ \mathsf{A})>$$

$$\mathsf{Tuple3} : \mathsf{U}_0 \to \mathsf{U}_0$$
$$\mathsf{Tuple3}\ \mathsf{A} = \sim(\mathsf{Vec}\ 3\ <\mathsf{A}>)$$

Staged types

Input:

$$\mathsf{Vec} : \mathsf{Nat}_1 \to \Uparrow \mathsf{U}_0 \to \Uparrow \mathsf{U}_0$$
$$\mathsf{Vec}\,\mathsf{zero}_1 \quad \mathsf{A} = <\top_0>$$
$$\mathsf{Vec}\,(\mathsf{suc}_1\,\mathsf{n})\,\mathsf{A} = <\sim \mathsf{A} \times \sim(\mathsf{Vec}\,\mathsf{n}\,\mathsf{A})>$$

$$\mathsf{Tuple3} : \mathsf{U}_0 \to \mathsf{U}_0$$
$$\mathsf{Tuple3}\,\mathsf{A} = \sim(\mathsf{Vec}\,3\,<\mathsf{A}>)$$

Output:

$$\mathsf{Tuple3} : \mathsf{U}_0 \to \mathsf{U}_0$$
$$\mathsf{Tuple3}\,\mathsf{A} = \mathsf{A} \times (\mathsf{A} \times (\mathsf{A} \times \top_0))$$

map for Vec

Input:

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

$$\begin{aligned}
&\mathsf{f} : \sim(\mathsf{Vec}\ 2 <\mathsf{Nat}_0>) \to \sim(\mathsf{Vec}\ 2 <\mathsf{Nat}_0>) \\
&\mathsf{f}\ \mathsf{xs} = \sim(\mathsf{map}\ 2\ (\lambda\, \mathsf{x}. <\sim \mathsf{x} + 2>)\ <\mathsf{xs}>)
\end{aligned}$$

map for Vec

Input:

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

$$\begin{aligned}
&\mathsf{f} : \sim(\mathsf{Vec}\ 2 <\mathsf{Nat}_0>) \to \sim(\mathsf{Vec}\ 2 <\mathsf{Nat}_0>) \\
&\mathsf{f}\ \mathsf{xs} = \sim(\mathsf{map}\ 2\ (\lambda\, \mathsf{x}. <\sim \mathsf{x} + 2>)\ <\mathsf{xs}>)
\end{aligned}$$

Output:

$$\begin{aligned}
&\mathsf{f} : \mathsf{Nat}_0 \times (\mathsf{Nat}_0 \times \top_0) \to \mathsf{Nat}_0 \times (\mathsf{Nat}_0 \times \top_0) \\
&\mathsf{f}\ \mathsf{xs} = (\mathsf{fst}_0\ \mathsf{xs} + 2,\ (\mathsf{fst}_0\ (\mathsf{snd}_0\ \mathsf{xs}) + 2,\ \mathsf{tt}_0))
\end{aligned}$$

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!