# 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*.

# 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 $A : U_0$, 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 (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 $<t> : \Uparrow A$.

# 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 $<t> : \Uparrow A$.
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 $<t> : \Uparrow A$.
5. *Splicing:* for $t : \Uparrow A$, we have $\sim t : A$.
6. $<\sim t> \equiv t$ and $\sim <t> \equiv t$.

# 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 $<t> : \Uparrow A$.
5. *Splicing:* for $t : \Uparrow A$, we have $\sim t : A$.
6. $<\sim t> \equiv t$ and $\sim <t> \equiv t$.

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

# Inlined definitions

Staging input:

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

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

# Inlined definitions

Staging input:

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

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

Output:

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

# 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:

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

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

Output:

$$\begin{aligned}
&\mathsf{idBool}_0 : \mathsf{Bool}_0 \to \mathsf{Bool}_0 \\
&\mathsf{idBool}_0 = \lambda\, \mathsf{x}.\, \mathsf{x}
\end{aligned}$$

# An alternative identity function

Input:

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

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

# An alternative identity function

Input:

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

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

*Note that*

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

# 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*

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

Output:

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

# map with inlining

Input:

$$\mathsf{inlMap} : \{A\,B : {\Uparrow}U_0\} \to ({\Uparrow}{\sim}A \to {\Uparrow}{\sim}B) \to {\Uparrow}(\mathsf{List}_0\,{\sim}A) \to {\Uparrow}(\mathsf{List}_0\,{\sim}B)$$
$$\mathsf{inlMap} = \lambda\, f\, as.\ {<}\mathsf{foldr}_0\,(\lambda\, a\, bs.\ \mathsf{cons}_0\,{\sim}(f\,{<}a{>})\, bs)\,\mathsf{nil}_0\,{\sim}as{>}$$

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

# map with inlining

Input:

$$\mathsf{inlMap} : \{A\,B : \Uparrow U_0\} \to (\Uparrow {\sim}A \to \Uparrow {\sim}B) \to \Uparrow(\mathsf{List}_0\,{\sim}A) \to \Uparrow(\mathsf{List}_0\,{\sim}B)$$
$$\mathsf{inlMap} = \lambda\, f\, as.\ {<}\mathsf{foldr}_0\,(\lambda\, a\, bs.\ \mathsf{cons}_0\,{\sim}(f\,{<}a{>})\,bs)\,\mathsf{nil}_0\,{\sim}as{>}$$

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

Output:

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

# 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**.

$$\begin{aligned}&\mathsf{inlMap} : \{A\,B : {\Uparrow}U_0\} \to ({\Uparrow}A \to {\Uparrow}B) \to {\Uparrow}(\mathsf{List}_0\,A) \to {\Uparrow}(\mathsf{List}_0\,B)\\&\mathsf{inlMap} = \lambda\,f.\,\mathsf{foldr}_0\,(\lambda\,a\,bs.\,\mathsf{cons}_0\,(f\,a)\,bs)\,\mathsf{nil}_0\end{aligned}$$

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

# Staged types

Input:

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

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

# Staged types

Input:

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

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

Output:

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

# map for Vec

Input:

$$
\begin{aligned}
&\mathsf{map} : \{A\,B : \Uparrow U_0\} \to (n : \mathsf{Nat}_1) \to (\Uparrow{\sim}A \to \Uparrow{\sim}B) \\
&\qquad\qquad\qquad\quad \to \Uparrow(\mathsf{Vec}\,n\,A) \to \Uparrow(\mathsf{Vec}\,n\,B) \\
&\mathsf{map}\;\mathsf{zero}_1 \quad\; f\;as = <\mathsf{tt}_0> \\
&\mathsf{map}\;(\mathsf{suc}_1\,n)\;f\;as = <({\sim}(f\,<\mathsf{fst}_0\,{\sim}as>),\;{\sim}(\mathsf{map}\,n\,f\,<\mathsf{snd}_0\,{\sim}as>))>
\end{aligned}
$$

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

# map for Vec

Input:

$$
\begin{aligned}
&\mathsf{map} : \{A\,B : \Uparrow U_0\} \to (n : \mathsf{Nat}_1) \to (\Uparrow{\sim}A \to \Uparrow{\sim}B) \\
&\qquad\qquad\qquad\qquad \to \Uparrow(\mathsf{Vec}\,n\,A) \to \Uparrow(\mathsf{Vec}\,n\,B) \\
&\mathsf{map}\,\mathsf{zero}_1 \quad\; f\,as = <\mathsf{tt}_0> \\
&\mathsf{map}\,(\mathsf{suc}_1\,n)\,f\,as = <(\sim(f <\mathsf{fst}_0 \sim as>),\ \sim(\mathsf{map}\,n\,f <\mathsf{snd}_0 \sim as>))>
\end{aligned}
$$

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

Output:

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

# 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!