Staged Compilation With Dependent Types

András Kovács

Eötvös Loránd University

26 May 2022

Application Domain Specific Highly Reliable IT Solutions - Thematic Excellence Project - Closing Conference

Metaprogramming & code generation

Metaprograms are programs which generate program code.

Metaprogramming & code generation

Metaprograms are programs which generate program code.

In any usual language, we can write programs which output code (as strings).

Metaprogramming & code generation

Metaprograms are programs which generate program code.

In any usual language, we can write programs which output code (as strings).

However, this has serious safety and ergonomics drawbacks.

- The well-typing and well-formedness of the output is not guaranteed.
- We have to work directly with syntax trees and/or strings.

Metaprogramming & code generation

Metaprograms are programs which generate program code.

In any usual language, we can write programs which output code (as strings).

However, this has serious safety and ergonomics drawbacks.

- The well-typing and well-formedness of the output is not guaranteed.
- We have to work directly with syntax trees and/or strings.

Staged compilation (two-stage):

- Users work in a language with structured metaprogramming features.
- **Staging** means running metaprograms and extracting code output.
- The compiler further proccesses the staging output.

Metaprogramming & code generation

Metaprograms are programs which generate program code.

In any usual language, we can write programs which output code (as strings).

However, this has serious safety and ergonomics drawbacks.

- The well-typing and well-formedness of the output is not guaranteed.
- We have to work directly with syntax trees and/or strings.

Staged compilation (two-stage):

- Users work in a language with structured metaprogramming features.
- **Staging** means running metaprograms and extracting code output.
- The compiler further proccesses the staging output.

Examples: *templates*, *generics*, *macros*.

Contribution

A highly general & expressive framework for staged compilation.

[1] Annekov, Capriotti, Kraus, Sattler: *Two-Level Type Theory and Applications.*

Contribution

A highly general & expressive framework for staged compilation.

Based on **two-level type theory** (2LTT), which was originally intended as a mathematical language of synthetic homotopy theory.[1]

[1] Annekov, Capriotti, Kraus, Sattler: *Two-Level Type Theory and Applications.*

Contribution

A highly general & expressive framework for staged compilation.

Based on **two-level type theory** (2LTT), which was originally intended as a mathematical language of synthetic homotopy theory.[1]

- The first staged system to support *dependent types*.
- Generalizes a wide range of existing typed metaprogramming systems.
- Has an efficient staging implementation + proof of soundness.

[1] Annekov, Capriotti, Kraus, Sattler: *Two-Level Type Theory and Applications.*

Contribution

A highly general & expressive framework for staged compilation.

Based on **two-level type theory** (2LTT), which was originally intended as a mathematical language of synthetic homotopy theory.[1]

- The first staged system to support *dependent types*.
- Generalizes a wide range of existing typed metaprogramming systems.
- Has an efficient staging implementation + proof of soundness.

Draft paper *"Staged Compilation With Two-Level Type Theory"* by AK, conditionally accepted at ICFP 2022.

---

[1] Annekov, Capriotti, Kraus, Sattler: *Two-Level Type Theory and Applications.*

2LTT language overview

A dependent type theory + extra staging features.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.
2. Type_1 is the type of **compile time** (meta-level) types. Meta-level types & their values only appear during compilation.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.
2. Type_1 is the type of **compile time** (meta-level) types. Meta-level types & their values only appear during compilation.
3. For $A : \mathsf{Type}_0$ we have $\Uparrow A : \mathsf{Type}_1$. This is the **type of metaprograms which generate code with type** A.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.
2. Type_1 is the type of **compile time** (meta-level) types. Meta-level types & their values only appear during compilation.
3. For $A : \mathsf{Type}_0$ we have $\Uparrow A : \mathsf{Type}_1$. This is the **type of metaprograms which generate code with type** A.
4. For $A : \mathsf{Type}_0$ and $t : A$, we have $\langle t \rangle : \Uparrow A$. This is the **metaprogram which returns** t **as an expression ("quote")**.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.
2. Type_1 is the type of **compile time** (meta-level) types. Meta-level types & their values only appear during compilation.
3. For $A : \mathsf{Type}_0$ we have $\Uparrow A : \mathsf{Type}_1$. This is the **type of metaprograms which generate code with type** A.
4. For $A : \mathsf{Type}_0$ and $t : A$, we have $\langle t \rangle : \Uparrow A$. This is the **metaprogram which returns** t **as an expression ("quote")**.
5. For $t : \Uparrow A$, we have $\sim t : A$. This **inserts the result of a metaprogram into an expression ("splice")**.

2LTT language overview

A dependent type theory + extra staging features.

Staging features

1. Type_0 is the type of **runtime** (object-level) types. Object-level types & their values will appear in generated code.
2. Type_1 is the type of **compile time** (meta-level) types. Meta-level types & their values only appear during compilation.
3. For $A : \text{Type}_0$ we have $\Uparrow A : \text{Type}_1$. This is the **type of metaprograms which generate code with type** A.
4. For $A : \text{Type}_0$ and $t : A$, we have $\langle t \rangle : \Uparrow A$. This is the **metaprogram which returns** t **as an expression ("quote")**.
5. For $t : \Uparrow A$, we have $\sim t : A$. This **inserts the result of a metaprogram into an expression ("splice")**.
6. These are the **only ways** to convert between Type_0 and Type_1.

Examples (1)

We use Agda-like syntax.

Runtime identity function

$$\begin{aligned} &\mathsf{id}_0 : (A : \mathsf{Type}_0) \to A \to A \\ &\mathsf{id}_0\, A\, x = x \end{aligned}$$

Examples (1)

We use Agda-like syntax.

Runtime identity function

$$\mathsf{id}_0 : (A : \mathsf{Type}_0) \to A \to A$$
$$\mathsf{id}_0\, A\, x = x$$

Compile-time identity function

$$\mathsf{id}_1 : (A : \mathsf{Type}_1) \to A \to A$$
$$\mathsf{id}_1\, A\, x = x$$

Examples (1)

We use Agda-like syntax.

Runtime identity function

$$\mathrm{id}_0 : (A : \mathrm{Type}_0) \to A \to A$$
$$\mathrm{id}_0\, A\, x = x$$

Compile-time identity function

$$\mathrm{id}_1 : (A : \mathrm{Type}_1) \to A \to A$$
$$\mathrm{id}_1\, A\, x = x$$

Assume $\mathrm{Bool}_0 : \mathrm{Type}_0$ and $\mathrm{true}_0 : \mathrm{Bool}_0$. Now, id_1 can be used on *expressions* as well:

$$\mathrm{id}_1\,(\Uparrow\mathrm{Bool})\,\langle\mathrm{true}\rangle : \Uparrow\mathrm{Bool}$$

This becomes simply $\langle\mathrm{true}\rangle$ after staging.

Examples (2)

Inlined map function

$$\mathsf{map} : (A\,B : {\Uparrow}\mathsf{Type}_0) \to ({\Uparrow}{\sim}A \to {\Uparrow}{\sim}B) \to {\Uparrow}(\mathsf{List}_0\;{\sim}A) \to {\Uparrow}(\mathsf{List}_0\,{\sim}B)$$

$$\begin{aligned}
&\mathsf{map\,A\,B\,f\,as} = \\
&\qquad \langle \mathsf{let\,go\,[]} \qquad\quad = [] \\
&\qquad\qquad \mathsf{go\,(a : as)} = {\sim}(\mathsf{f}\;\langle \mathsf{a}\rangle) : \mathsf{go\,as} \\
&\qquad\;\; \mathsf{in}\;\; \mathsf{go}\,{\sim}\mathsf{as}\rangle
\end{aligned}$$

Examples (2)

Inlined map function

$$\mathsf{map} : (A\,B : \Uparrow\mathsf{Type}_0) \to (\Uparrow{\sim}A \to \Uparrow{\sim}B) \to \Uparrow(\mathsf{List}_0\ {\sim}A) \to \Uparrow(\mathsf{List}_0\,{\sim}B)$$

$$\begin{aligned}
&\mathsf{map}\,\mathsf{A}\,\mathsf{B}\,\mathsf{f}\,\mathsf{as} = \\
&\qquad \langle \mathsf{let}\;\mathsf{go}\,[] \qquad\quad = [] \\
&\qquad\qquad \mathsf{go}\,(\mathsf{a} : \mathsf{as}) = {\sim}(\mathsf{f}\;\langle\mathsf{a}\rangle) : \mathsf{go}\,\mathsf{as} \\
&\qquad\; \mathsf{in}\;\; \mathsf{go}\,{\sim}\mathsf{as}\rangle
\end{aligned}$$

With inferred staging annotations:

$$\mathsf{map} : (A\,B : \Uparrow\mathsf{Type}_0) \to (A \to B) \to \mathsf{List}_0\ A \to \mathsf{List}_0\ B$$

$$\begin{aligned}
&\mathsf{map}\,\mathsf{A}\,\mathsf{B}\,\mathsf{f}\,\mathsf{as} = \\
&\qquad \mathsf{let}\;\mathsf{go}\,[] \qquad\quad = [] \\
&\qquad\quad \mathsf{go}\,(\mathsf{a} : \mathsf{as}) = \mathsf{f}\;\mathsf{a} : \mathsf{go}\,\mathsf{as} \\
&\qquad \mathsf{in}\;\; \mathsf{go}\,\mathsf{as}
\end{aligned}$$

Computing types at compile time

Vectors as nested pairs

$$
\begin{aligned}
&\mathsf{Vector} : \mathsf{Nat}_1 \to {\Uparrow}\mathsf{Type}_0 \to {\Uparrow}\mathsf{Type}_0 \\
&\mathsf{Vector}\ 0 \qquad\quad\ \mathsf{A} = () \\
&\mathsf{Vector}\ (\mathsf{n}+1)\,\mathsf{A} = \langle(\sim\mathsf{A},\ \sim(\mathsf{Vector}\,\mathsf{n}\,\mathsf{A}))\rangle
\end{aligned}
$$

Computing types at compile time

Vectors as nested pairs

$$\begin{aligned}
&\mathsf{Vector} : \mathsf{Nat}_1 \to {\Uparrow}\mathsf{Type}_0 \to {\Uparrow}\mathsf{Type}_0\\
&\mathsf{Vector}\ 0\qquad\qquad \mathsf{A} = ()\\
&\mathsf{Vector}\ (\mathsf{n}+1)\,\mathsf{A} = \langle(\sim\mathsf{A},\ \sim(\mathsf{Vector}\,\mathsf{n}\,\mathsf{A}))\rangle
\end{aligned}$$

$\sim(\mathsf{Vector}\,3\,\langle\mathsf{Bool}_0\rangle)$ is computed to $(\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ ())))$.

Computing types at compile time

Vectors as nested pairs

$$\begin{aligned}
&\mathsf{Vector} : \mathsf{Nat}_1 \to \Uparrow\mathsf{Type}_0 \to \Uparrow\mathsf{Type}_0 \\
&\mathsf{Vector}\ 0 \qquad\quad\ \ \mathsf{A} = () \\
&\mathsf{Vector}\ (\mathsf{n}+1)\,\mathsf{A} = \langle(\sim\mathsf{A},\ \sim(\mathsf{Vector}\,\mathsf{n}\,\mathsf{A}))\rangle
\end{aligned}$$

$\sim(\mathsf{Vector}\,3\,\langle\mathsf{Bool}_0\rangle)$ is computed to $(\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ ())))$.

We can also write a map for vectors of given lengths. We can generate types + well-typed programs depending on generated types.

Computing types at compile time

Vectors as nested pairs

$$
\begin{aligned}
&\mathsf{Vector} : \mathsf{Nat}_1 \to \Uparrow\mathsf{Type}_0 \to \Uparrow\mathsf{Type}_0 \\
&\mathsf{Vector}\ 0 \qquad\quad \mathsf{A} = () \\
&\mathsf{Vector}\ (\mathsf{n}+1)\,\mathsf{A} = \langle(\sim\!\mathsf{A},\ \sim\!(\mathsf{Vector}\,\mathsf{n}\,\mathsf{A}))\rangle
\end{aligned}
$$

$\sim\!(\mathsf{Vector}\,3\,\langle\mathsf{Bool}_0\rangle)$ is computed to $(\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ (\mathsf{Bool}_0,\ ())))$.

We can also write a map for vectors of given lengths. We can generate types + well-typed programs depending on generated types.

This has not been possible in previous systems.