Using Two-Level Type Theory for Staged Compilation

András Kovács[a]

June 15, TYPES 2021

[a]The author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Staged Compilation

Programs which generate programs.

Staged Compilation

Programs which generate programs.

With extra features:

- Guaranteed well-typed output.
- Lightweight syntax.
- Seamless integration of object-level and meta-level.

Staged Compilation

Programs which generate programs.

With extra features:

- Guaranteed well-typed output.
- Lightweight syntax.
- Seamless integration of object-level and meta-level.

Price to pay: some metaprograms are not expressible.

Two-Level Type Theory

Voevodsky's Homotopy Type System [Voe13], 2LTT [ACKS19].

Original motivation: metaprogramming, with HoTT as object theory

Two-Level Type Theory

Voevodsky's Homotopy Type System [Voe13], 2LTT [ACKS19].

Original motivation: metaprogramming, with HoTT as object theory

Turns out to implement two-stage programming:

- Works for wide range of theories

Two-Level Type Theory

Voevodsky's Homotopy Type System [Voe13], 2LTT [ACKS19].

Original motivation: metaprogramming, with HoTT as object theory

Turns out to implement two-stage programming:

- Works for wide range of theories
- Simple rules

Two-Level Type Theory

Voevodsky's Homotopy Type System [Voe13], 2LTT [ACKS19].

Original motivation: metaprogramming, with HoTT as object theory

Turns out to implement two-stage programming:

- Works for wide range of theories
- Simple rules
- Fast staging with NbE

Two-Level Type Theory

Voevodsky's Homotopy Type System [Voe13], 2LTT [ACKS19].

Original motivation: metaprogramming, with HoTT as object theory

Turns out to implement two-stage programming:

- Works for wide range of theories
- Simple rules
- Fast staging with NbE
- Nice model theory and standard semantics

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!
- For $A : \mathsf{U}_0$, we have $\mathsf{Code}\, A : \mathsf{U}_1$

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!
- For $A : \mathsf{U}_0$, we have $\mathsf{Code}\,A : \mathsf{U}_1$
- Quoting: for $A : \mathsf{U}_0$ and $t : A$ we have $\langle t \rangle : \mathsf{Code}\,A$

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!
- For $A : \mathsf{U}_0$, we have $\mathsf{Code}\,A : \mathsf{U}_1$
- Quoting: for $A : \mathsf{U}_0$ and $t : A$ we have $\langle t \rangle : \mathsf{Code}\,A$
- Splicing: for $t : \mathsf{Code}\,A$, we have $\sim\! t : A$

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!
- For $A : \mathsf{U}_0$, we have $\mathsf{Code}\,A : \mathsf{U}_1$
- Quoting: for $A : \mathsf{U}_0$ and $t : A$ we have $\langle t \rangle : \mathsf{Code}\,A$
- Splicing: for $t : \mathsf{Code}\,A$, we have $\sim t : A$
- $\langle \sim t \rangle = t$
- $\sim \langle t \rangle = t$

Rules

- U_0 (object-level) and U_1 (meta-level) universes, both closed under arbitrary type formers.
- Constructors and eliminators stay within universes!
- For $A : U_0$, we have $\mathrm{Code}\,A : U_1$
- Quoting: for $A : U_0$ and $t : A$ we have $\langle t \rangle : \mathrm{Code}\,A$
- Splicing: for $t : \mathrm{Code}\,A$, we have $\sim t : A$
- $\langle \sim t \rangle = t$
- $\sim \langle t \rangle = t$

Staging: computing away every meta-level subterm in an object-level term.

Identity functions

$$
\begin{aligned}
\mathsf{id}_0 &: (A : \mathsf{U}_0) \to A \to A \\
\mathsf{id}_0\, \mathsf{Bool}_0\, \mathsf{true}_0 &: \mathsf{Bool}_0
\end{aligned}
$$

Identity functions

$$
\begin{aligned}
\mathsf{id}_1 &: (A : \mathsf{U}_1) \to A \to A \\
\mathsf{id}_1\ \mathsf{Bool}_1\ \mathsf{true}_1 &: \mathsf{Bool}_1 \\
\mathsf{id}_1\ (\mathsf{Code}\ \mathsf{Bool}_0)\ \langle \mathsf{true}_0 \rangle &: \mathsf{Code}\ \mathsf{Bool}_0
\end{aligned}
$$

Quantifying over $\mathsf{Code}\,\mathsf{U}_0$

Inlined object-level map:

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

$$\mathsf{map}\,\langle \mathsf{Nat}_0 \rangle\,\langle \mathsf{Nat}_0 \rangle\,(\lambda\, x.\langle (\sim x) + 10 \rangle) : \mathsf{Code}(\mathsf{List}_0\,\mathsf{Nat}_0) \to \mathsf{Code}(\mathsf{List}_0\,\mathsf{Nat}_0))$$

Staging Types, Inference

$$\begin{aligned}
&\mathsf{Vec} : \mathsf{Nat}_1 \to \mathsf{Code}\,\mathsf{U}_0 \to \mathsf{Code}\,\mathsf{U}_0 \\
&\mathsf{Vec}\,\mathsf{zero}_1 \quad\; A = \langle \top_0 \rangle \\
&\mathsf{Vec}\,(\mathsf{suc}_1\, n)\, A = \langle (\sim A) \times_0 \sim(\mathsf{Vec}\, n\, A) \rangle
\end{aligned}$$

Staging Types, Inference

$$\mathsf{Vec} : \mathsf{Nat}_1 \to \mathsf{Code}\,\mathsf{U}_0 \to \mathsf{Code}\,\mathsf{U}_0$$
$$\mathsf{Vec}\,\mathsf{zero}_1 \quad A = \langle \top_0 \rangle$$
$$\mathsf{Vec}\,(\mathsf{suc}_1\; n)\, A = \langle (\sim A) \times_0 \sim(\mathsf{Vec}\, n\, A) \rangle$$

With annotation inference:

$$\mathsf{Vec} : \mathsf{Nat}_1 \to \mathsf{U}_0 \to \mathsf{U}_0$$
$$\mathsf{Vec}\,\mathsf{zero}_1 \quad A = \top_0$$
$$\mathsf{Vec}\,(\mathsf{suc}_1\; n)\, A = A \times_0 \mathsf{Vec}\, n\, A$$

Demo: well-typed staged STLC interpreter, all annotations inferred.

Weak Object Language + Strong Metalanguage

Simpler object theory $\to$ better performance

2LTT recovers features for free: universe, Π, Σ

Simple TT at Object-Level

A system for monomorphization.

Simple TT at Object-Level

A system for monomorphization.

Universe of object types: $\mathsf{Ty} : \mathsf{U}_1$, $\mathsf{Code} : \mathsf{Ty} \to \mathsf{U}_1$.

$$\mathsf{id} : (A : \mathsf{Ty}) \to \mathsf{Code}(A \to A)$$

Simple TT at Object-Level

A system for monomorphization.

Universe of object types: $\mathsf{Ty} : \mathsf{U}_1$, $\mathsf{Code} : \mathsf{Ty} \to \mathsf{U}_1$.

$$\mathsf{id} : (A : \mathsf{Ty}) \to \mathsf{Code}(A \to A)$$

Higher-rank polymorphism via inlining:

$$\mathsf{poly} : ((A : \mathsf{Ty}) \to \mathsf{Code}\,A \to \mathsf{Code}\,A) \to (\mathsf{Code}\,\mathsf{Bool},\ \mathsf{Code}\,\mathsf{Int})$$

What we can't do: store polymorphic functions in object-level data.

First-Order Functions at Object-Level

A system for closure-free compilation.

First-Order Functions at Object-Level

A system for closure-free compilation.

2LTT gives us higher-order functions for free.

First-Order Functions at Object-Level

A system for closure-free compilation.

2LTT gives us higher-order functions for free.

What we can't do: store functions in object-level data.

First-Order Functions at Object-Level

A system for closure-free compilation.

2LTT gives us higher-order functions for free.

What we can't do: store functions in object-level data.

Surprisingly expressive.

Memory Layout-Indexed Types at Object-Level

A system for layout polymorphism (levity polymorphism).

$$\mathsf{id} : (L : \mathsf{Layout}) \to (A : \mathsf{U}_0\, L) \to \mathsf{Code}(A \to A)$$

Staging computes layouts to closed canonical values.

Standard Semantics

Presheaves over the syntactic category of object theory.

Standard Semantics

Presheaves over the syntactic category of object theory.

Code is “dependent” Yoneda-embedding.

Standard Semantics

Presheaves over the syntactic category of object theory.

Code is “dependent” Yoneda-embedding.

Choice of morphisms in the base category:

Standard Semantics

Presheaves over the syntactic category of object theory.

Code is “dependent” Yoneda-embedding.

Choice of morphisms in the base category:

- Substitutions: only generative staging

Standard Semantics

Presheaves over the syntactic category of object theory.

Code is “dependent” Yoneda-embedding.

Choice of morphisms in the base category:

- Substitutions: only generative staging
- Weakenings: allows Code analysis, but fewer object theories

Demos

https://github.com/AndrasKovacs/implicit-fun-elaboration/tree/staging

WIP: https://github.com/AndrasKovacs/staged

References i

Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler.
Two-level type theory and applications.
ArXiv e-prints, may 2019.

Vladimir Voevodsky.
A simple type system with two identity types.
Unpublished note, 2013.