Staged Compilation with Two-Level Type Theory

2023-08-11

Staged Compilation

Staged compilation separates code compilation in at two stages.

- Compile time (aka meta-level, indexed 1) stage and runtime (aka object-level, indexed 0) stage.
- Each stage can have their own language.

A staging algorithm converts/stages a metaprogram to a program with only runtime language.

- A metaprogram is a term with runtime type but uses type/terms from the compile-time language through staging annotations.

Examples where staged compilation is useful:

- Metaprogramming: evaluate the code-generation annotations (e.g. macros, inlining) to runtime language, i.e. without annotations.
- Domain specific languages: e.g. LINQ (C#)

Two-Level Type Theory

A staging algorithm needs to be sound:

- Any well-typed metaprogram should be staged into a well-typed runtime code.
- The resulting code of staging does not use any types/terms from the compile-time language.

To justify the the soundness of staging

- Two-level type theory can be applied.
- Treat the two stages as separate type systems.
- Restrict the interaction between stages with explicit annotations.

Unlike previous works like MetaML [TS97], staged compilation with 2LTT supports dependent types.

Π-Types

To adhere to the notations from the original paper on staged compilation [Kov22], we introduce new notations in comparison with Pie.

- Dependent function types (Π-types)

$$f : (a : A) \to (b : B) \to C$$

```
(claim f (Π ([a A] [b B]) C))
```

- Functions (lambdas)

$$f := \lambda\, a\, b.\, \mathrm{body}$$

```
(define f (λ (a b) body))
```

Moving Between Stages

We have two universes of types, one for each stage:

- U_0 for the universe of stage 0 (recall 0 is index for runtime/object stage)
- U_1 for the universe of stage 1 (recall 1 is index for compile-time/meta stage)

For interaction between stages, we define three staging annotations on the compile-time level.

Lifting: Compute Runtime Expressions at Compile-Time

Given $A : \mathsf{U}_0$, we have $\Uparrow A : \mathsf{U}_1$

- The lifted type $\Uparrow A : \mathsf{U}_1$ is the type of metaprograms that compute runtime expression of type $A : \mathsf{U}_0$.

Quoting: Metaprograms from Runtime Terms

Given $t : A$ and $A : \mathsf{U}_0$, we have $\langle t \rangle : {\Uparrow}A$ and ${\Uparrow}A : \mathsf{U}_1$

- The quoted term $\langle t \rangle : {\Uparrow}A$ is a metaprogram that immediately yields the term $t : A$.

Splicing: Executing Metaprograms During Staging

Given $t : {\Uparrow}A$ and ${\Uparrow}A : \mathsf{U}_1$, we have ${\sim}t : A$ and $A : \mathsf{U}_0$

- The spliced metaprogram ${\sim}t$ will be executed during staging, and substituted by result expression.

Moving Between Stages

With the three staging annotations for moving between stages:

- *Lifting*: Given $A : \mathsf{U}_0$, we have $\Uparrow A : \mathsf{U}_1$
- *Quoting*: Given $t : A$ and $A : \mathsf{U}_0$, we have $\langle t \rangle : \Uparrow A$
- *Splicing*: Given $t : \Uparrow A$ and $\Uparrow A : \mathsf{U}_1$, we have $\sim t : A$

We have two equalities:

$$\sim\langle t \rangle = t$$
$$\langle \sim s \rangle = s$$

The Natural Eliminator

We introduce natural numbers for each stage $i \in \{0, 1\}$

$$
\begin{aligned}
\mathsf{Nat}_i &: U_i \\
\mathsf{zero}_i &: \mathsf{Nat}_i \\
\mathsf{suc}_i &: \mathsf{Nat}_i \to \mathsf{Nat}_i \\
\mathsf{NatElim}_i &: (P : \mathsf{Nat}_i \to \mathsf{U}_{i,j}) \\
&\to P\,\mathsf{zero}_i \\
&\to ((n : \mathsf{Nat}_i) \to P\,n \to P\,(\mathsf{suc}_i\,n)) \\
&\to (t : \mathsf{Nat}_i) \\
&\to P\,t
\end{aligned}
$$

For simplicity, let us define iter_i to represent `iter-Nat` from Pie

$$
\begin{aligned}
&\mathsf{iter}_i : (X : U_i) \to \mathsf{Nat}_i \to X \to (X \to X) \to X \\
&\mathsf{iter}_i := \lambda\,X\,t\,z\,s.\,\mathsf{NatElim}_i\,(\lambda\,n.\,X)\,z\,(\lambda\,n\,\mathsf{acc}.\,s\,\mathsf{acc})\,t
\end{aligned}
$$

Addition and Multiplication

$$\mathsf{iter}_i : (X : U_i) \to \mathsf{Nat}_i \to X \to (X \to X) \to X$$
$$\mathsf{iter}_i := \lambda\, X\, t\, z\, s.\, \mathsf{NatElim}_i\,(\lambda\, n.\, X)\, z\,(\lambda\, n\, \mathsf{acc}.\, s\, \mathsf{acc})\, t$$

Similar to the definition of + and * in Pie, we can implement them in 2LTT as well.

- We put add_0 to stage 0:

$$\mathsf{add}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0$$
$$\mathsf{add}_0 := \lambda\, a\, b.\, \mathsf{iter}_0\, \mathsf{Nat}_0\, a\, b\,(\lambda\, n. \mathsf{suc}_0\, n)$$

- For multiplication, it takes a compile-time number $x : \mathsf{Nat}_1$ and produces a metaprogram that computes the product with x at runtime.

$$\mathsf{mul}_1 : \mathsf{Nat}_1 \to \Uparrow\mathsf{Nat}_0 \to \Uparrow\mathsf{Nat}_0$$
$$\mathsf{mul}_1 = \lambda\, x\, t.\, \mathsf{iter}_1\,(\Uparrow\mathsf{Nat}_0)\, x\, \langle \mathsf{zero}_0 \rangle\, \langle \mathsf{add}_0 \sim t \rangle$$

The Staging Process

$$\mathsf{iter}_i : (X : U_i) \to \mathsf{Nat}_i \to X \to (X \to X) \to X$$
$$\mathsf{iter}_i := \lambda\, X\, t\, z\, s.\, \mathsf{NatElim}_i\, (\lambda\, n.\, X)\, z\, (\lambda\, n\, \mathsf{acc}.\, s\, \mathsf{acc})\, t$$
$$\mathsf{add}_0 : \mathsf{Nat}_0 \to \mathsf{Nat}_0 \to \mathsf{Nat}_0$$
$$\mathsf{add}_0 := \lambda\, a\, b.\, \mathsf{iter}_0\, \mathsf{Nat}_0\, a\, b\, (\lambda\, n. \mathsf{suc}_0\, n)$$
$$\mathsf{mul}_1 : \mathsf{Nat}_1 \to {\Uparrow}\mathsf{Nat}_0 \to {\Uparrow}\mathsf{Nat}_0$$
$$\mathsf{mul}_1 = \lambda\, x\, t.\, \mathsf{iter}_1\, ({\Uparrow}\mathsf{Nat}_0)\, x\, \langle \mathsf{zero}_0 \rangle\, \langle \mathsf{add}_0 \sim t \rangle$$

With add_0 being a function on stage 0 and mul_1 on stage 1, a metaprogram, for instance

$$\mathsf{double} : \mathsf{Nat}_0 \to \mathsf{Nat}_0$$
$$\mathsf{double} := \lambda\, x.\, {\sim}(\mathsf{mul}_1\, 2\, \langle x \rangle)$$

will get staged to

$$\mathsf{double} := \lambda\, x.\, \mathsf{add}_0\, x\, (\mathsf{add}_0\, x\, \mathsf{zero}_0)$$

Formal Inference Rules

$$\textsc{lift}\ \frac{\Gamma \vdash_{0,j} A}{\Gamma \vdash_{1,j} \Uparrow A} \qquad \textsc{quote}\ \frac{\Gamma \vdash_{0,j} t : A}{\Gamma \vdash_{1,j} \langle t \rangle : \Uparrow A} \qquad \textsc{splice}\ \frac{\Gamma \vdash_{1,j} t : \Uparrow A}{\Gamma \vdash_{0,j} {\sim} t : A}$$

$$\textsc{quote-splice}\ \frac{\Gamma \vdash_{1,j} t : \Uparrow A}{\Gamma \vdash_{1,j} \langle {\sim} t \rangle = t : \Uparrow A} \qquad \textsc{splice-quote}\ \frac{\Gamma \vdash_{0,j} t : A}{\Gamma \vdash_{0,j} {\sim} \langle t \rangle = t : A}$$

Limitation of Staging

The original purpose of 2LTT is to express meta-theoretical statements about homotopy type theory (HoTT)

- “From a type in HoTT, we can extract a statement that can be phrased in the meta-theory. From a meta-theoretical statement *about* HoTT, it is not always possible to construct a type. Thus, we can convert inner types into outer one, but not always vice versa.” [Ann+19]
- Therefore cannot splice arbitrary stage 1 term.
- Stage 0 don’t always have ways to represent types in stage 1.
- $\sim$zero$_1$ would be invalid.

Isomorphism Between Types

$$\Uparrow((a : A) \to B\,a) \simeq (a : \Uparrow A) \to \Uparrow(B \sim a)$$

$$\Uparrow((a : A) \times B\,a) \simeq ((a : \Uparrow A) \times \Uparrow(B \sim a))$$

Isomorphism Example

$$\text{pres}_{\to} : \Uparrow((x : A) \to B\,x) \to ((x : \Uparrow A) \to \Uparrow(B \sim x))$$
$$\text{pres}_{\to}\, f := \lambda\, x.\ \langle \sim f \sim x \rangle$$
$$\text{pres}_{\to} := \lambda\, f\, x.\ \langle \sim f \sim x \rangle$$

$$\text{pres}_{\to}^{-1} : ((x : \Uparrow A) \to \Uparrow(B \sim x)) \to \Uparrow((x : A) \to B\,x)$$
$$\text{pres}_{\to}^{-1}\, f := \langle \lambda\, x.\ \sim(f\, \langle x \rangle) \rangle$$
$$\text{pres}_{\to}^{-1} := \lambda\, f.\ \langle \lambda\, x.\ \sim(f\, \langle x \rangle) \rangle$$

Isomorphism Example

$$
\begin{aligned}
\mathsf{pres}_{\to}(\mathsf{pres}_{\to}^{-1} f) &= \;\; (\lambda\, f\, x.\ \langle \sim\! f \sim\! x \rangle)\ ((\lambda\, f.\ \langle \lambda\, x.\ \sim\!(f\ \langle x \rangle)\rangle)\ f) \\
&=_\beta (\lambda\, f\, x.\ \langle \sim\! f \sim\! x \rangle)\ \langle \lambda\, x.\ \sim\!(f\ \langle x \rangle)\rangle \\
&=_\beta \lambda\, x.\ \langle \sim\!\langle \lambda\, x.\ \sim\!(f\ \langle x \rangle)\rangle \sim\! x \rangle \\
&= \;\; \lambda\, x.\ \langle (\lambda\, x.\ \sim\!(f\ \langle x \rangle)) \sim\! x \rangle \\
&=_\beta \lambda\, x.\ \langle \sim\!(f\ \langle \sim\! x \rangle)\rangle \\
&= \;\; \lambda\, x.\ \langle \sim\!(f\ x)\rangle \\
&= \;\; \lambda\, x.\ (f\ x) \\
&=_\eta f
\end{aligned}
$$

Result of 2LTT

- Staging: Given a program $t : A$, where $A : \mathsf{U}_0$. *Staging* computes metaprograms and replace all splices in t and A with resulting runtime expression.
- 2LTT guarantees resulting computation does not contain more splices.
- Regardless of the body, if you have a runtime type program, it can be turned into a program using strictly stage 0 terms and constructors.

Bibliography

[TS97] Walid Taha and Tim Sheard. "MetaML and Multi-Stage Programming with Explicit Annotations". In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203–217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: https://dl.acm.org/doi/10.1145/258994.259019.

[Ann+19] Danil Annenkov et al. "Two-Level Type Theory and Applications". In: *ArXiv e-prints* (May 2019). URL: http://arxiv.org/abs/1705.03307.

[Kov22] András Kovács. "Staged Compilation with Two-Level Type Theory". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547641. URL: https://doi.org/10.1145/3547641.

End