Staged Compilation and Generativity

András Kovács

Eötvös Loránd University, Department of Programming Languages and Compilers

kovacsandras@inf.elte.hu

Overview

The purpose of staged compilation is to write code-generating programs in a safe and ergonomic way. Although it is always possible to write metaprograms by simply manipulating strings or deeply embedded syntax trees, this is often error-prone and tedious. Staging is a way to have more guarantees about the safety and well-typing of metaprograms, and also a way to integrate object-level and meta-level syntaxes more organically.

Two-level type theory (2LTT) [2] was originally developed for the purpose of doing synthetic homotopy theory, by adding a metaprogramming layer on top of homotopy type theory [6]. However, it turns out that 2LTT is also a great framework for metaprogramming and staging in general, and it is applicable to a wide range of theories, both on the object and meta level.

There is a simple semantics to 2LTT which justifies the metaprogramming view: this is the *presheaf model* of 2LTT. Here, meta-level types are presheaves over the underlying category of the object theory. Hence, every meta-level construction must be stable under the object-level morphisms. The advantage of working in 2LTT stems from stability under morphisms: if every construction is automatically stable, it becomes possible to omit explicit handling of base morphisms. More concretely, from the staging perspective, this means that we never have to deal with scoping, renaming, substitution or de Bruijn indices in the object syntax, when working in 2LTT.

Generativity. Generativity means that we can only generate code, but not make decisions based on the internal structure of object-level syntax. Generativity simplifies staging, and it is often enforced in practical implementations [5]. However, non-generative staging provides additional power and flexibility. A simple example for a non-generative feature is conversion checking. This can be viewed as an axiom in 2LTT, which says that meta-level ("strict") equality of object-level values is decidable. We aim to investigate semantics of non-generativity in the following.

Basic Rules and Usage of 2LTT

To illustrate using 2LTT for staging, we specify a simple variant of 2LTT where we have exactly the same dependent type theory for the object-level and meta-level theories.

We have universes U_i^s, where $s \in \{0, 1\}$, denoting a *stage* or level in the 2LTT sense, and $i \in \mathbb{N}$ denotes a usual level index of *sizing* hierarchies. The two dimensions of indexing are orthogonal, and we will elide the i indices in the following. We assume Russell-style universes. Both U_0 and U_1 may be closed under arbitrary type formers, but eliminators

in each *only target the same universe*, i.e. elimination cannot cross universes. We have the following operations:

- For $A : \mathsf{U}_0$, we have $\mathsf{Code}\, A : \mathsf{U}_1$. This is the type of meta-level programs which return object-level code with type A.
- Quoting: for $A : \mathsf{U}_0$ and $t : A$ we have $\langle t \rangle : \mathsf{Code}\, A$. In other words, for every object-level term we have a metaprogram which immediately returns that term.
- Splicing: for $t : \mathsf{Code}\, A$, we have $\sim t : A$. This means running a metaprogram and inserting its result into object-level code.
- We also know that quoting/splicing is a definitional isomorphism, so $\langle \sim t \rangle = t$ and $\sim \langle t \rangle = t$.

A *staging algorithm* takes as input a closed term $t : A$ where $A : \mathsf{U}_0$, and splices the results of all metaprograms, so that we get an output term which is free of splices. This can be implemented using variations of normalization-by-evaluation [1] which track current stages. We do not detail staging algorithms here.

Let's look at some examples. We have the object-level identity function as usual:

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

Staging does not do anything with id_0, since it has no splices. Likewise if we apply id_0 to object-level values, as in $\mathsf{id}_0\, \mathsf{Bool}\, \mathsf{true}$. We also have the meta-level version:

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

Note that this also works on object-level values, because of quoting:

$$\sim(\mathsf{id}_0\, (\mathsf{Code}\, \mathsf{Bool}_0)\, \langle \mathsf{true}_0 \rangle) : \mathsf{Bool}_0$$

Staging the above term computes to $\sim\langle \mathsf{true}_0 \rangle$, which in turn computes to true_0. Thus, id_1 is compile-time evaluated. There's a third version, which is a specialized version of id_1: it's also evaluated at compile time, but it only works on object-level types:

$$\begin{aligned}&\mathsf{id}_{\mathsf{Code}} : (A : \mathsf{Code}\, \mathsf{U}_0) \to \mathsf{Code}\, (\sim A) \to \mathsf{Code}\, (\sim A)\\ &\mathsf{id}_{\mathsf{Code}} := \lambda\, A\, x.\, x\end{aligned}$$

Now, $\sim(\mathsf{id}_{\mathsf{Code}}\, \langle \mathsf{Bool}_0 \rangle\, \langle \mathsf{true}_0 \rangle)$ also stages to true_0. Meta-functions which are restricted to Code are also useful when we want to define functions which are partially evaluated at compile time. For example, if we want to inline a function argument for object-level list mapping:

$$\begin{aligned}&\mathsf{map} : (A\, B : \mathsf{Code}\, \mathsf{U}_0) \to (\mathsf{Code}(\sim A) \to \mathsf{Code}(\sim B))\\ &\qquad \to \mathsf{Code}(\mathsf{List}_0\, (\sim A)) \to \mathsf{Code}(\mathsf{List}_0\, (\sim B)))\\ &\mathsf{map} := \lambda\, A\, B\, f\, as.\, \langle \mathsf{foldr}_0\, (\lambda\, a\, bs.\, \mathsf{cons}_0\, (\sim (f\, \langle a \rangle))\, bs)\, \mathsf{nil}_0\, (\sim as) \rangle\end{aligned}$$

Presheaf Model

Why consider the presheaf model? The reason is that it's the simplest semantics which justifies the metaprogramming view of 2LTT. It is *not* the same thing as the staging algorithm, which is based on normalization-by-evaluation, which is much more complicated to formalize [3]. The presheaf model is fairly simple as far as models go, so it's interesting to see which staging features can be justified with it. We skip presenting the whole presheaf model. For details, we refer the reader to [4, Section 1.2].

We give some examples for interpreting constructions in the model. We present the results up to isomorphism, with some simplifications. We assume now that object-level morphisms are substitutions. We have $\mathsf{Bool}_0 : \mathsf{U}_0$ and $\mathsf{Bool}_1 : \mathsf{U}_1$.

- A closed function $t : \mathsf{Bool}_1 \to \mathsf{Bool}_1$ becomes a metatheoretical function in $\mathbb{B} \to \mathbb{B}$.
- A closed function $t : \mathsf{Bool}_0 \to \mathsf{Bool}_0$ becomes a closed object-theoretic function in $\mathsf{Bool} \to \mathsf{Bool}$.
- A closed function $t : \mathsf{Code}\,\mathsf{Bool}_0 \to \mathsf{Bool}_1$ becomes a function which maps a Bool term in any context to $\mathbb{B}$, such that the function commutes with object-theoretic substitution. For example if we have a variable x, we can substitute it with any term before feeding it to the semantic t, and the result is the same. In fact, this means that t cannot depend on its term argument, hence t is specified simply by a $\mathbb{B}$.
- A closed function $t : \mathsf{Bool}_1 \to \mathsf{Code}\,\mathsf{Bool}_0$ becomes simply a pair of closed Bool terms.

Yoneda lemma. The Yoneda lemma is a general statement which restricts the way meta-level values can depend on object-level ones. First, note that any object-level typing context Γ can be mapped to a presheaf, by taking the sets of parallel substitutions into Γ. This is the *Yoneda embedding* of Γ, denoted by $\mathsf{y}\Gamma$. The Yoneda lemma says that we have the following isomorphism of sets:

$$(\mathsf{y}\,\Gamma \Rightarrow \Delta) \simeq |\Delta|\,\Gamma$$

where $\Rightarrow$ means a natural transformation, and $|\Delta|\,\Gamma$ denotes the set that we get by evaluating the Δ presheaf at the object-level Γ context. From this, what we essentially get is that any 2LTT term $\Gamma \vdash t : A$, such that Γ is essentially interpreted as $\mathsf{y}\,\Gamma'$ for some Γ', is interpreted as an element of $|A|\Gamma'$. We call Γ *representable* is there is such Γ'.

In particular, if $\Gamma \vdash t : \mathsf{Bool}_1$ and Γ representable, then since $|\mathsf{Bool}_1|\,\Gamma' = \mathbb{B}$, t is simply an element of $\mathbb{B}$ in the semantics, and cannot depend on the typing context.

In short, whether the Yoneda lemma applies to a given term, depends on whether the typing context is representable. In turn, the representability of the context depends on what morphisms are in the object theory. We consider two options.

1. Morphisms are substitutions. In this case, y preserves context extension, i.e. $\mathsf{y}\,(\Gamma, x : A) \simeq (\mathsf{y}\,\Gamma, x : \mathsf{y}A)$ in the presheaf model. That's because a substitution which targets $(\Gamma, x : A)$ is equivalent to a pair of substitutions, targeting Γ and A respectively. Therefore, if we have $x_1 : A_1$, $x_2 : A_2$, ..., $x_i : A_i \vdash t : B$, such that all A_i are representable, the entire context is also representable, and the Yoneda lemma applies.

Which types are representable? For starters, every type of the form $\mathsf{Code}\,A$, since Code in the model is essentially interpreted as y (eliding the formal complications arising from possible dependencies of A on the context). Also, if we have $x : A$ in a context, where $A : \mathsf{U}_0$, that context extension is also interpreted as extension with $\mathsf{y}A$. In short: if the context only has Code types or types in U_0, it is representable.

This greatly limits non-generative features in the model. Consider adding the axiom which says that meta-level equality of object-level values is decidable: $x : \mathsf{Code}\,A, y : \mathsf{Code}\,A \vdash \mathsf{conversion}_A : (x = y) + (x = y \to \bot)$. This is a simple non-generative axiom, since any constructive interpretation must look inside Code-s. This axiom is false if object morphisms are substitutions. That's because the context is representable, so we can simplify using the Yoneda lemma. The statement that we get in the model is that "two terms are either equal, or they are inequal and remain inequal after arbitrarily substitutions". Now, if we pick two *variables* x and y such that $x \neq y$, then they are not equal, but they can be also made equal by substituting both variables with the same term.

Can we repair this? One possibility is to have decidable equality only for *closed* terms. However, the syntax of 2LTT provides no way to talk about closed terms. Instead we'd have to use a closed modality [3]. This would be interesting to investigate in future work.

2. Morphisms are weakenings. In this case, object morphisms are are so-called *order-preserving embeddings*, meaning that a morphism can drop zero or more entries from a context, so morphisms are essentially bitmasks which mark a sub-context. The action of weakening embeds terms in larger contexts. Moreover, y does not preserve context extension. Hence, typing contexts are not necessarily representable, even if they only contain object-level bindings. So the Yoneda-reduction of dependencies generally does not apply.

Now, $\mathsf{conversion}$ is fine, because inequality of terms is stable under weakening. On the other hand, by only having weakening in the object theory, the range of supported object theories is greatly restricted. For example, we can't have β-reduction for functions in the equational theory, since that's specified using substitution. Likewise, dependent types are out, since the typing of dependent elimination involves substitution.

Simple type theories still work, if we only have weakening in their equational theory. For the perspective of staging, this is fine, because in *code generation* we care about the intensional definition of programs, and we do not want to equate β-reducts, since a primary use-case of staging is to improve runtime performance, hence distinguish between possibly β-convertible programs.

It appears that if morphisms are weakenings, then the presheaf model is compatible with a wide range of non-generative axioms. For example, we can also postulate *countability* of $\mathsf{Code}\,A$, i.e. that there are injections $\mathsf{index}_A : \mathsf{Code}\,A \to \mathsf{Nat}_1$. In the presheaf model, the indexing function works by enumerating maximally strengthened terms, which are stable under weakening.

Other ways of justifying non-generativity. An alternative solution would be to use something other than the presheaf model to justify non-generative axioms. For example: could we use the staging algorithm itself, i.e. does normalization-by-evaluation support non-generativity? It seems likely, as semantic values need only be stable under weakening. This remains future work.

References

[1] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* PhD thesis, Ludwig-Maximilians-Universität München, 2013. Habilitation thesis.

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

[3] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. Induction principles for type theories, internally to presheaf categories. *arXiv preprint arXiv:2102.11649*, 2021.

[4] Simon Huber. *Cubical Interpretations of Type Theory.* PhD thesis, University of Gothenburg, 2016.

[5] Oleg Kiselyov. The design and implementation of BER metaocaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014.

[6] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.