Using Two-Level Type Theory for Staged Compilation [*]

András Kovács

Eötvös Loránd University, Budapest, Hungary
kovacsandras@inf.elte.hu

Two-level type theory [2] (2LTT) is a system for performing metatheoretic constructions and reasoning involving certain object-level type theories. Such reasoning is always possible by simply embedding object theories in metatheories, but in that case we have to explicitly handle a deluge of technical details about the object theory, most notably substitutions. If everything that we aim to do is natural with respect to object-level substitution, we can instead use 2LTT, which can be viewed as a notation for working with presheaves over the object-level category of substitutions.

Likewise in metaprogramming, there is a spectrum: we can simply write programs which output raw source code, or use staging instead, which is safer and more convenient, but also restricted in some ways. In the current work we observe that 2LTT is a powerful model for generative staged compilation.

Basic rules of 2LTT. 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. For each $\Gamma \vdash A : \mathsf{U}^0$, we have $\Gamma \vdash \mathsf{Code}\,A : \mathsf{U}^1$. Quoting: for each $\Gamma \vdash t : A$, we have $\Gamma \vdash <t>: \mathsf{Code}\,A$. Unquoting: for each $\Gamma \vdash t : \mathsf{Code}\,A$, we have $\Gamma \vdash \sim t : A$. Moreover, quoting and unquoting form an isomorphism up to definitional equality. U^0 and U^1 can be closed under arbitrary additional type formers.

The idea of staging is the following: given a closed $A : \mathsf{U}^0$ with a closed $t : A$ in 2LTT, there are unique A' and t' in the object theory, which become definitionally equal to A and t respectively after being embedded in 2LTT. In short, every meta-level construction can be computed away, and only object-level constructions remain in the result. Annekov et al. [2] showed this without uniqueness of A' and t' (as a conservativity theorem for 2LTT). We can get uniqueness as well, using the normalization of 2LTT: by induction on the (unique) normal forms of A and t, we can show that they cannot contain meta-level subterms. This shows that normalization is a sound staging algorithm, but in practice we do not want to compute full normal forms; we want to compute *meta-level redexes only.* This can be done with a variation of standard normalization-by-evaluation [1, 5] which also keeps track of stages.

Applications

Control over inlining and compile-time computation. We can define two variations of the polymorphic identity function for object-level types:

$$\begin{array}{ll} id : (A : \mathsf{U}^0) \to A \to A & id' : (A : \mathsf{Code}\,\mathsf{U}^0) \to \mathsf{Code}\ {\sim}A \to \mathsf{Code}\ {\sim}A \\ id = \lambda\, A\, x.\, x & id' = \lambda A\, x. x \end{array}$$

The second version is evaluated at compile time. For example, $\sim(id'\ <\mathsf{Bool}^0><\mathsf{true}^0>)$ can be used in object-level code, which is computed to true^0 by staging. We can also freely

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

use induction on meta-level values to generate object-level code, including types. Hence, 2LTT supports full dependent types (with universes and large elimination) in staging.

Monomorphization. We assume now that the object language is a *simple type theory*. In this case, there is no universe U^0 in the object-level, so there is no $\mathsf{Code}\,\mathsf{U}^0$, but we can still freely include a meta-level type Ty^0 whose terms are identified with object-level types. Now, meta-level functions can be used for quantification over object-level types, as in $id : (A : \mathsf{Ty}^0) \to \mathsf{Code}\,A \to \mathsf{Code}\,A$. However, since the object theory is simply typed and monomorphic, all polymorphism is guaranteed to compute away during staging.

Control over lambda lifting and closure creation. We assume now a dependent type theory on both levels, but with a *first-order function type* on the object level. This is defined by splitting U^0 to a universe V^0 which is closed under inductive types but not functions, and a universe C^0 which has V^0 as a sub-universe, and is closed under functions with domains in V^0 and codomains in C^0. This object theory supports compilation which requires only lambda lifting, but no closures. On its own, the object theory is fairly restricted, but together with staging we have a remarkably expressive system. Then, we can close V^0 under a separate type former of closure-based functions, thereby formally distinguishing lambda-liftable functions from closure-based functions. This enables typed analysis of various optimization and fusion techniques. E.g. we get guaranteed closure-freedom in code output if a certain fusion technique can be formalized with only first-order function types. In particular, this may obviate the need for arity analysis [3] in fold-based fusion.

Memory layout control. We assume again a dependent theory on both levels, but now index U^0 over *memory layouts*. For example, $\mathsf{U}^0\,\mathsf{erased}$ may contain runtime-erased types, and $\mathsf{U}^0\,(\mathsf{word64} \times \mathsf{word64})$ may contain types represented as unboxed pairs of machine words. We assume a meta-level type of layouts. Hence, we can abstract over layouts, but after staging every layout will be concrete and canonical in the output. This can be viewed as a more powerful version of *levity polymorphism* in GHC [4], and a way to retain both dependent types and non-uniform memory layouts in the object theory.

Potential extensions

More stages, stage polymorphism. The standard presheaf semantics of 2LTT can be extended to more levels in a straightforward way. It seems feasible to also allow quantifying over all smaller levels, at a given level.

Stage inference. Code preserves all negative type formers up to definitional isomorphism [2], e.g. $\mathsf{Code}\,(A \to B) \simeq (\mathsf{Code}\,A \to \mathsf{Code}\,B)$. This can be used to support inference for staging annotations, by automatically inserting transports along preservation isomorphisms during elaboration. The previous $\sim\!(id' <\mathsf{Bool}^0><\mathsf{true}^0>)$ example could be simply written as $id'\,\mathsf{Bool}^0\,\mathsf{true}^0$ in the surface language, and elaboration would transport id' appropriately. We demonstrated the practical feasibility of such stage inference in a prototype implementation.

Induction on Code. Basic 2LTT supports *any model* of the object theory in the presheaf semantics, it does not assume that we have presheaves over the initial model (syntax). Hence, $\mathsf{Code}\,A$ is a black box without elimination principles. However, if we are interested in staged compilation, we can assume the object-level to be syntactic and consistently add operations on $\mathsf{Code}\,A$ which rely on that assumption, e.g. conversion checking, pattern matching, or induction on normal forms of object-level expressions.

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] Joachim Breitner. Call arity. *Comput. Lang. Syst. Struct.*, 52:65–91, 2018.

[4] Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 525–539. ACM, 2017.

[5] PawełWieczorek and Dariusz Biernacki. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 266–279, New York, NY, USA, 2018. ACM.