Eta conversion for the unit type (is still not that simple)

András Kovács
University of Gothenburg & Chalmers University of Technology
Sweden

η-conversion for the unit type is a simple feature with well-known formal metatheory, but in practical implementations it's more often shunned than supported. The problem is that it requires type-aware conversion, and implementors prefer the comfort of the purely syntax-directed world. To date, Agda is the only big system that attempts to support it, but its support is not particularly efficient or complete [2]. I describe improved designs in the following, which I hope to be much faster than the existing Agda version, while guaranteeing unique solutions of unification problems.

Conversion checking without metavariables

This turns out to be pretty easy. The main principle: *types should be only computed if unit η can possibly make a difference.* This is different from Agda.

- In Agda, conversion checking receives the type of the sides as an extra argument, and scrutinizes the head of the type during each head comparison.
- We use normalization-by-evaluation where neutrals are annotated with their lazily computed semantic type, and also the lazily computed *definitional relevance* of that type. Conversion checking is *still syntax-directed* and ignorant of types for the most part. The only difference is that whenever conversion checking for two neutrals fails, we can "catch" the failure by testing if the type is definitionally irrelevant, i.e. if all inhabitants of the type are convertible.

This design optimizes performance for the definitionally relevant case. We traverse irrelevant terms deeply, and compute relevance only if they are not strictly the same. Agda can immediately short-circuit the comparison of irrelevant terms, but at the cost of weak-head forcing the type of every subterm.

Handling metavariables

This introduces several complications. During conversion checking, metavariables can be compared against terms and become solved. As a general rule, we can only solve metavariables if we're working in a *relevant computational context.*

For example, assuming bound variables f and g and metavariable α, when comparing $f\ (g\ \alpha)$ to $f\ (g\ t)$, if the return type of g is irrelevant, we should not solve α to t, regardless of α's relevance. Doing so would unnecessarily restrict α.

Hence, we pass the lazily computed relevance of the computational context as an extra argument to conversion checking, and we look at it whenever we want to solve a metavariable, or when we hit a rigid mismatch. The context is relevant if every enclosing neutral value has a relevant type, and irrelevant otherwise. If we look at the context relevance and find it to be irrelevant, we throw an exception that gets catched at the innermost enclosing irrelevant neutral. Note that relevance checking can be blocked by metavariables, so relevances can be "unknown".

Here, we typically compute many more types than in the metavariable-free case; every metavariable solution forces the types of enclosing neutrals. The situation is not *too* bad though. First, typically, much of conversion checking doesn't produce meta solutions.

Second, we can exploit *elaboration* for an optimization. For example, when bidirectional elaboration switches from checking to inference, it compares inferred and expected types. At that point, we can additionally compute and record the relevance of the type, and embed it into core syntax in the elaboration output. Then, when evaluation hits a relevance annotation, it can immediately tag neutral values with relevance, without forcing their types.

Pattern unification

There are two more operations where relevance makes a difference: a) pattern inversion in pattern unification b) partial substitution of right hand sides of unification problems [1]; this generalizes "occurs checking" and scope checking.

In both cases, irrelevance works as a get-out-of-jail card: when we run into an error, it turns into a success if we're in an irrelevant context.

- Linearity errors can be disregarded if non-linear variables are irrelevant, e.g. $\alpha\ x\ x =?\ f\ x$ is uniquely solvable as $\alpha = \lambda\ \_\ x.\ f\ x$ if x is irrelevant.
- An otherwise non-invertible neutral term can be inverted if its type is irrelevant. For example, if $f : \mathsf{Bool} \to \top$, then $\alpha\ (f\ \mathsf{True})\ x =?\ x$ is solvable with $\alpha = \lambda\ \_\ x.\ x$.
- In partial substitution for the RHS, an illegal occurrence error can be caught at an enclosing contractible neutral, in which case we replace the neutral with the unique closed inhabitant of the type.

Is this worth the effort?

If we only consider the η-rule for the unit type, all of this extra infrastructure might not look like a great deal. However, almost the same setup can be reused for other things. First, we can use it to support a strict Prop with an embedding from Prop to Type (which Agda supports but Coq doesn't). Second, extension types (cubical or non-cubical) [4, 5] and cubical sub-types [3] also induce definitional irrelevance that needs to be handled.

References

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6690)*, C.-H. Luke Ong (Ed.). Springer, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

[2] Agda developers. 2022. Agda issue 5837. https://github.com/agda/agda/issues/5837

[3] Agda developers. 2024. Documentation for Cubical Agda. https://agda.readthedocs.io/en/v2.7.0.1/language/cubical.html

[4] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2022. Controlling unfolding in type theory. *CoRR* abs/2210.05420 (2022). https://doi.org/10.48550/ARXIV.2210.05420 arXiv:2210.05420

[5] Emily Riehl and Michael Shulman. 2017. A type theory for synthetic ∞-categories. *arXiv preprint arXiv:1705.07442* (2017).