Controlling unfolding in type theory using extension types

Daniel Gratzer[1], Jonathan Sterling[1], Carlo Angiuli[2], Thierry Coquand[3], and Lars Birkedal[1]

[1] Aarhus University
gratzer@cs.au.dk jsterling@cs.au.dk birkedal@cs.au.dk
[2] Carnegie Mellon University
cangiuli@cs.cmu.edu
[3] Chalmers University
Thierry.Coquand@cse.gu.se

In dependent type theory, terms are type checked modulo definitional equality, a congruence generated by α-, $\beta-$, and η-laws, as well as unfolding of definitions. The last aspect is an important convenience in practice; it allows a user to factor their development into lemmas and definitions without obstructing later proofs. Despite this advantage, it is not universally beneficial to unfold definitions when type-checking. Unfolding every definition, for instance, would result in cluttered proof states and unreadable error messages. More abstractly, allowing every definition to unfold allows all subsequent proofs to depend on the precise implementation of each definition and theorem. Definitional equality is notoriously brittle, so even small changes to a widely-used definition will result in wide-spread breakage if that definition is always unfoldable.

To combat this, most proof assistants have some ability to mark definitions as abstract and prevent them unfolding outside a well-defined scope. For instance, Agda's `abstract` blocks [10] allow a user to define an operation and prove lemmas relying on its definition but treat the operation (and lemmas) as abstract after exiting the block. Unfortunately, it is frequently difficult to predict exactly which lemmas need to be proven prior to sealing a definition. Faced with the choice between having to guess an all-encompassing set of lemmas or leaving a definition available for users, library authors are typically forced to leave their core definitions completely translucent. For instance, as of v1.7.2 the Agda standard library uses `abstract` only once.

We propose a more refined mechanism for controlling which definitions may be unfolded and when, allowing the user fine-grained control over the opacity of a definition. Concretely, in our system every definition is abstract by default but a user may mark a definition translucent within the scope of a particular definition. We have demonstrated the theoretical soundness of our mechanism by providing an elaboration algorithm to type theory supplemented with *extension types* [6]. We prove normalization for this calculus using synthetic Tait computability [9, 7] and thereby derive the necessary metatheory to implement our elaboration procedure. We have also implemented our elaboration procedure within `cooltt` [5] to show that controlled unfolding is highly usable in practice. Recently, Liao and Cockx have given an independent implementation of controlled unfolding in Agda [4]. Further details are available in our preprint on this topic [3].

Controlled unfolding by example For a running example, consider the definition of the $+$ operator and the constructors of a type of length-indexed lists:

$$\mathsf{ze} + n = n \qquad \mathsf{vnil} : \mathsf{vec}\ \mathsf{ze}\ A$$
$$\mathsf{su}\ m + n = \mathsf{su}\ (m + n) \qquad \mathsf{vcons} : A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$$

By default, $+$ is opaque. Left this way, however, it becomes impossible to define an append operation $\oplus$ for vectors. Accordingly, we must mark $+$ as unfoldable in the definition of $\oplus$:

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$\mathsf{vnil} \oplus y = y \qquad \mathsf{vcons}(a, x) \oplus y = \mathsf{vcons}(a, x \oplus y)$

With this annotation, $+$ unfolds definitionally within the body of $\oplus$ but *not* its type. If the definition of $+$ could be unfolded in the type of $\oplus$, any definition using the latter must also be able to unfold the former. As is, subsequent definitions may use $\oplus$ without unfolding $+$.

While the basic picture is clear, dependency offers two complications. First, what should be done in the case where we do wish to unfold $+$ in a type? Second, what happens if a third definition unfolds $\oplus$? The first question is handled through a design pattern: we isolate a type that requires unfolding a definition into its own definition (of a constant of type $\mathcal{U}$). The tedium can be alleviated through a simple elaboration mechanism. The second question is more complex. If a definition unfolds $\oplus$, it must also necessarily unfold $+$. Without this transitive unfolding, one can quickly produce ill-typed terms.[1]

Elaborating controlled unfolding Rather than providing an equational theory for controlled unfolding directly, we justify its soundness through an elaboration procedure. This algorithm reduces the surface language with controlled unfolding to $\mathbf{TT}_{\mathbb{P}}$, a core language featuring only abstract definitions, constants and, crucially, extension types. Extension types $\{A \mid \phi \hookrightarrow M\}$ were first axiomatized by Riehl and Shulman [6] and have played an important role in cubical type theory [2, 1]. To a first approximation, an element $N : \{A \mid \phi \hookrightarrow M\}$ is an element of A such that $\phi \vdash M = N : A$ where ϕ is drawn from a collection of (very strict) propositions.[2] More precisely, extension types are governed by the following rules:

$$\frac{A\ \mathsf{type} \qquad \phi\ \mathsf{prop} \qquad \phi \vdash M : A}{\{A \mid \phi \hookrightarrow M\}} \qquad \frac{N : A \qquad \phi \vdash N = M : A}{N : \{A \mid \phi \hookrightarrow M\}} \qquad \frac{N : \{A \mid \phi \hookrightarrow M\}}{N : A \qquad \phi \vdash N = M : A}$$

Intuitively, the elaboration procedure replaces each definition $D : A \triangleq M$ with a constant $\alpha_D : \{A \mid \phi_D \hookrightarrow M\}$ where ϕ_D is a freshly generated proposition associated with the definition D. If D unfolds another definition $D' : B \triangleq N$, we add an *entailment* $\phi_D \vdash \phi_{D'}$. Examining the rules for extension types, this addition ensures that when checking $\phi_D \vdash M : A$ as part of $\{A \mid \phi_D \hookrightarrow M\}$, each occurrence of $\alpha_{D'}$ is definitionally equal to N.

Under this type-theoretic account of controlled unfolding many of the subtleties mentioned previously simply disappear. For instance, that unfolding declarations does not impact the type of definition is an consequence of the fact that the A in $\{A \mid \phi \hookrightarrow M\}$ must be a well-formed type without assuming ϕ. Similarly, unfolding is transitive *automatically* because entailment is transitive: if $\phi_D \vdash \phi_{D'}$ and $\phi_{D'} \vdash \phi_{D''}$, then $\phi_D \vdash \phi_{D''}$.

In order to extend the sketch above to a bidirectional elaboration algorithm suitable for implementation, some formal properties of type theory with extension types are required. In particular, we must argue that (1) definitional equality of types is decidable and (2) type-constructors such as dependent products are injective. Both are consequences of the following:

Theorem 1. $\mathbf{TT}_{\mathbb{P}}$ *admits a normalization algorithm.*

This result is proven using synthetic Tait computability along with Sterling and Angiuli's *stabilized neutrals* [8]. Moreover, the elaboration algorithm along with several additional convenience features on top of controlled unfolding has been implemented in `cooltt` and is available for experimentation.

[1] From an implementation point of view this is a failure of subject reduction.
[2] In homotopical theories, ϕ is a cofibration. For us, they are drawn from a fixed external partial order.

References

[1] Carlo Angiuli et al. "Syntax and models of Cartesian cubical type theory". In: *Mathematical Structures in Computer Science* 31.4 (2021), pp. 424–468. DOI: 10.1017/S0960129521000347.

[2] Cyril Cohen et al. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: *IfCoLog Journal of Logics and their Applications* 4.10 (2017), pp. 3127–3169. arXiv: 1611.02108 [cs.LO].

[3] Daniel Gratzer et al. *Controlling unfolding in type theory*. 2022. DOI: 10.48550/ARXIV.2210.05420. URL: https://arxiv.org/abs/2210.05420.

[4] Amélia Liao and Jesper Cockx. *Unfolding control for abstract blocks*. 2022. URL: https://github.com/agda/agda/pull/6354.

[5] The RedPRL Development Team. *cooltt*. 2020. URL: http://www.github.com/RedPRL/cooltt.

[6] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1.1 (2017), pp. 147–224. URL: https://arxiv.org/abs/1705.07442.

[7] Jonathan Sterling. "First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory". Version 1.1, revised May 2022. PhD thesis. Carnegie Mellon University, 2021. DOI: 10.5281/zenodo.6990769.

[8] Jonathan Sterling and Carlo Angiuli. "Normalization for Cubical Type Theory". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '21. New York, NY, USA: ACM, 2021.

[9] Jonathan Sterling and Robert Harper. "Logical Relations as Types: Proof-Relevant Parametricity for Program Modules". In: *Journal of the ACM* 68.6 (Oct. 2021). ISSN: 0004-5411. DOI: 10.1145/3474834. arXiv: 2010.08599 [cs.PL].

[10] The Agda Team. *Agda User Manual, Release 2.6.2*. 2021. URL: https://agda.readthedocs.io/_/downloads/en/v2.6.2/pdf/.