Controlling unfolding in type theory

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

HoTT 2023 2023-05-23

Aarhus University, Carnegie Mellon University, Chalmers University

Proof assistants versus core type theory

What differentiates a core theory from an actual proof assistant?

- Advanced features: implicit arguments, unification, pattern-matching
- Intermediate features: termination checking, schemata for inductive types
- Very basic features: definitions

Our goal: improve the UX of a feature by pushing the core theory to include it.

Definitions in proof assistants

Turns out this is hard, so let's start with the basics: definitions

Crucial point:

two : $\mathbb{N}$
two $\triangleq 2$

_ : two $= 2$
_ $\triangleq$ refl

Definitions in proof assistants

Turns out this is hard, so let's start with the basics: definitions

Crucial point:

$\mathsf{two} : \mathbb{N}$

$\mathsf{two} \triangleq 2$

Definitions should unfold.... definitionally

$\_ : \mathsf{two} = 2$

$\_ \triangleq \mathsf{refl}$

Definitions in proof assistants

Turns out this is hard, so let's start with the basics: definitions

Crucial point:

two : $\mathbb{N}$

two $\triangleq$ 2

Definitions should unfold.... definitionally

_ : two $=$ 2

_ $\triangleq$ refl

Hardly a startling insight, but it is rather crucial; only way to prove something

The next steps

Fully translucent definitions certainly work, but not without cost.

Pros of unfolding	Cons of unfolding
We can prove things	Goals become unreadable
	Type-checking performance degrades
	Increases coupling between implementation and use

The next steps

Fully translucent definitions certainly work, but not without cost.

Pros of unfolding	Cons of unfolding
We can prove things	Goals become unreadable
	Type-checking performance degrades
	Increases coupling between implementation and use

In practice, the left-hand column wins.

Controlled unfolding: desiderata

We can't just refuse to unfold definitions, but we can control when it happens...

- Default opaque/abstract definitions
- Users may *explicitly* unfold a definition within a fixed scope
- The system tracks dependencies to ensure type-soundness
- Unfolding should be silent in terms; can't obstruct further computation

Library authors leave things abstract-by-default. If a user must unfold, they can.

Our contributions

Our core idea is to design a mechanism satisfying these desiderata

- We revisit the type-theoretic account of translucent definitions (singleton types)
- Refine this idea by replacing singleton types with extension types
- Propose a surface syntax/elaboration mechanism

Starting with the core language makes it easy to propose various extensions

https://arxiv.org/abs/2210.05420

Singleton types: an account of translucent definitions

How does one express normal/translucent definitions type-theoretically?

- Each definition will be encoded by a variable
- ... but with a fancy type.
- This idea doesn't come from dependent type theory, but from module systems

Encode a definition $x : A \triangleq M$ through a type $S_A(M)$ containing only one element: M.

Translucent definitions versus abstract definitions

Very roughly, we have the following:

- Abstract/opaque definitions:

$$x : A \cong \left(\sum_{a:A} \bot \to (a = M)\right)$$

- Normal/translucent definitions:

$$x : S_A(M) \cong \left(\sum_{a:A} \top \to (a = M)\right)$$

Either we never gain access to the proof $a = M$ or we're always stuck with it.

Translucent definitions versus abstract definitions

(For the sake of this slide: extensional equality)

Very roughly, we have the following:

- Abstract/opaque definitions:

$$x : A \cong \left(\sum_{a:A} \bot \to (a \doteq M)\right)$$

- Normal/translucent definitions:

$$x : S_A(M) \cong \left(\sum_{a:A} \top \to (a \doteq M)\right)$$

Either we never gain access to the proof $a = M$ or we're always stuck with it.

Extension types

- Key idea: let's allow propositions other than $\top$ and $\bot$.
- We need a universe of very strict propositions $\mathbb{F}$.
- Close $\mathbb{F}$ under (at least) $\top$ and $\wedge$.
- New form of context Γ, ϕ and new judgment $\Gamma \vdash \phi$ *true*.

Notation and properties inspired by cofibrations from cubical type theory.

(Spoilers): $\mathbb{F}$ isolates subshapes $\rightsquigarrow$ $\mathbb{F}$ classifies which definitions unfold.

New type formers: extension types

$$\frac{A \text{ type} \qquad \phi \vdash M : A}{\{A \mid \phi \hookrightarrow M\} \text{ type}}$$

$$\frac{N : A \qquad \phi \vdash N = M : A}{\mathsf{in}(N) : \{A \mid \phi \hookrightarrow M\}} \qquad \frac{N : \{A \mid \phi \hookrightarrow M\}}{\mathsf{out}(N) : A}$$

Normal β/η rules

New type formers: extension types

$$\frac{A \text{ type} \qquad \phi \vdash M : A}{\{A \mid \phi \hookrightarrow M\} \text{ type}}$$

$$\frac{N : A \qquad \phi \vdash N = M : A}{\mathsf{in}(N) : \{A \mid \phi \hookrightarrow M\}} \qquad \frac{N : \{A \mid \phi \hookrightarrow M\}}{\mathsf{out}(N) : A}$$

Normal β/η rules

$$\frac{N : \{A \mid \phi \hookrightarrow M\} \qquad \phi \text{ true}}{\mathsf{out}(N) = M : A}$$

New type formers: extension types

Only defined when ϕ is true.

$$\frac{A \text{ type} \qquad \phi \vdash M : A}{\{A \mid \phi \hookrightarrow M\} \text{ type}}$$

$$\frac{N : A \qquad \phi \vdash N = M : A}{\mathsf{in}(N) : \{A \mid \phi \hookrightarrow M\}} \qquad \frac{N : \{A \mid \phi \hookrightarrow M\}}{\mathsf{out}(N) : A}$$

Normal β/η rules

$$\frac{N : \{A \mid \phi \hookrightarrow M\} \qquad \phi \text{ true}}{\mathsf{out}(N) = M : A}$$

Big idea: definitions become extension types

Fix a definition $x : A \triangleq M$.

1. Associate a fresh proposition symbol Υ_x to the definition.
2. Encode the definition as a constant $x : \{A \mid \Upsilon_x \hookrightarrow M\}$.
3. Replace subsequent occurrences of x with $\mathsf{out}(x)$.

Taking $\Upsilon_x = \top$ gives normal definitions.

Big idea: definitions become extension types

Fix a definition $x : A \triangleq M$.

1. Associate a fresh proposition symbol Υ_x to the definition.
2. Encode the definition as a constant $x : \{A \mid \Upsilon_x \hookrightarrow M\}$.
3. Replace subsequent occurrences of x with $\mathsf{out}(x)$.

Taking $\Upsilon_x = \top$ gives normal definitions.

If Υ_x is some fresh symbol, how can we ever unfold this definition?

Unfolding definitions via extension types

Short answer: more extension types.

- We first consider how to unfold definitions for an entire subsequent definition.
- Following our scheme, have

$$x : \{A \mid \Upsilon_x \hookrightarrow M\} \qquad y : \{B \mid \Upsilon_y \hookrightarrow N\}$$

- If we want to make sure x unfolds definitionally in N, force $\Upsilon_y \implies \Upsilon_x$

Unfolding definitions via extension types

Short answer: more extension types.

- We first consider how to unfold definitions for an entire subsequent definition.
- Following our scheme, have

$$x : \{A \mid \Upsilon_x \hookrightarrow M\} \qquad y : \{B \mid \Upsilon_y \hookrightarrow N\}$$

- If we want to make sure x unfolds definitionally in N, force $\Upsilon_y \implies \Upsilon_x$

$$\begin{aligned} &\text{We check } N \text{ after assuming } \Upsilon_y \\ \implies &\text{so } \Upsilon_x \text{ holds when checking } N \\ \implies &\text{so } \mathrm{out}(x) = M \text{ in } N \end{aligned}$$

This is why we want to be sure to check N as a partial element!

Big idea II

Fix a definition $x : A \triangleq M$.

1. Specify which definitions x unfolds e.g. $y_0 \ldots y_n$
2. Associate a fresh proposition symbol Υ_x to the definition.
3. Add the following principle:

$$\frac{\Gamma \vdash \Upsilon_x \ \mathit{true}}{\Gamma \vdash \Upsilon_{y_i} \ \mathit{true}}$$

4. Encode the definition as a constant $x : \{A \mid \Upsilon_x \hookrightarrow M\}$.
5. Replace subsequent occurrences of x with $\mathsf{out}(x)$.

Big idea II

Fix a definition $x : A \triangleq M$.

1. Specify which definitions x unfolds e.g. $y_0 \ldots y_n$
2. Associate a fresh proposition symbol Υ_x to the definition.
3. Add the following principle:
$$\frac{\Gamma \vdash \Upsilon_x \; \mathit{true}}{\Gamma \vdash \Upsilon_{y_i} \; \mathit{true}}$$
4. Encode the definition as a constant $x : \{A \mid \Upsilon_x \hookrightarrow M\}$.
5. Replace subsequent occurrences of x with $\mathsf{out}(x)$.

Warning

A bunch of ways to specify what it means to add these propositions/inequalities.

Don't worry about it.

What is a program?

- Normally, a program is a sequence of definitions
- For us then, a program is a sequence of axioms
- Each axiom either specified a proposition, an inequality, and an extension type.

$$\begin{array}{ll} & \textbf{prop } \Upsilon_{\mathsf{neg}} \\ \mathsf{neg} : \mathbb{Z} \to \mathbb{Z} & \textbf{axiom } \mathsf{neg} : \{\mathbb{Z} \to \mathbb{Z} \mid \Upsilon_{\mathsf{neg}} \hookrightarrow \ldots\} \\ \mathsf{neg} \triangleq \ldots & \\ & \rightsquigarrow \textbf{prop } \Upsilon_{\mathsf{invol}} \\ \mathsf{invol} : (n : \mathbb{Z}) \to \mathsf{neg}(\mathsf{neg}\, n) = n & \textbf{inequality } \Upsilon_{\mathsf{invol}} \leq \Upsilon_{\mathsf{neg}} \\ \mathsf{invol} \triangleq \ldots & \textbf{axiom } \mathsf{invol} : \\ & \quad \{(n : \mathbb{Z}) \to \mathsf{neg}(\mathsf{neg}\, n) = n \mid \Upsilon_{\mathsf{invol}} \hookrightarrow \ldots\} \end{array}$$

Two forms of dependence

Dependence can now be *transparent* or *opaque*.

Suppose A depends on B depends on C.

- If A $\rightarrow$ B is transparent and B $\rightarrow$ C is transparent, so is A $\rightarrow$ C.
- Not the case for any of the other instances of 2-of-3

This is crucial: we can unfold something without having it infect the whole codebase.

Evaluating this mechanism

- Using extension types automatically ensures we unfold “just enough”
- Unless requested, nothing will unfold!
- Automatically type safe & respects conversions
- Equations are *definitional* and don’t produce coherence hell!

Evaluating this mechanism

- Using extension types automatically ensures we unfold “just enough”
- Unless requested, nothing will unfold!
- Automatically type safe & respects conversions
- Equations are *definitional* and don’t produce coherence hell!

Not a panacea

- Currently at the granularity of definitions
- Writing these extension types is weird

Evaluating this mechanism

- Using extension types automatically ensures we unfold "just enough"
- Unless requested, nothing will unfold!
- Automatically type safe & respects conversions
- Equations are *definitional* and don't produce coherence hell!

Not a panacea

- Currently at the granularity of definitions
- Writing these extension types is weird

Solved through elaboration!

A surface syntax for unfolding

- Now that we have a target core language in place, we want nice syntax
- Should abstract a bit, but the translation should be simple and predictable
- In particular, the transformation should be compositional and local

We will define the surface syntax by elaboration.

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

$\quad$ **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

$\mathsf{foo} \triangleq M$

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

  unfolding $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

$\mathsf{foo} \triangleq M$

Normal components of a definition

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

$\quad$ **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$ $\leftarrow$ What is unfolded in M

$\mathsf{foo} \triangleq M$

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

  unfolding $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

$\mathsf{foo} \triangleq M$

M may make use definitions other than bar_i! They just won't unfold

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

$\quad$ **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

$\mathsf{foo} \triangleq M$

M may make use definitions other than bar_i! They just won't unfold

Many other convenience features are possible (local unfolds, abbreviations, etc.)

A small amount of precision.

How can we actually crystallize this?

- Define several *elaboration judgments*
- Term-level components look like fancy bidirectional type-checking
- Should be decidable $\rightsquigarrow$ elaboration can be implemented

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

$$\Sigma \vdash \vec{S} \leadsto \Sigma'$$

$$\Sigma; \Gamma \vdash \tau \Leftarrow \mathit{type} \leadsto \Sigma', A$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \leadsto \Sigma', M$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \leadsto \Sigma', M$$

Σ is a signature: a list of fresh propositions, axioms, and inequalities.

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

$$\Sigma \vdash \vec{S} \rightsquigarrow \Sigma'$$

Main judgment; essentially `flatMap`

$$\Sigma; \Gamma \vdash \tau \Leftarrow \mathit{type} \rightsquigarrow \Sigma', A$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma', M$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma', M$$

Σ is a signature: a list of fresh propositions, axioms, and inequalities.

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

$$\Sigma \vdash \vec{S} \leadsto \Sigma'$$

$$\Sigma; \Gamma \vdash \tau \Leftarrow \mathit{type} \leadsto \Sigma', A$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \leadsto \Sigma', M$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \leadsto \Sigma', M$$

Elaborate a type;
Σ: input signature
Γ: local variables
hoist local-unfolds into Σ'
Invariant: A wf wrt Σ, Γ, Σ'

Σ is a signature: a list of fresh propositions, axioms, and inequalities.

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

$$\Sigma \vdash \vec{S} \rightsquigarrow \Sigma'$$

$$\Sigma; \Gamma \vdash \tau \Leftarrow \mathit{type} \rightsquigarrow \Sigma', A$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma', M$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma', M$$

Elaborate a term
A is given & wf'd
Output is a core term

Σ is a signature: a list of fresh propositions, axioms, and inequalities.

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

$$\Sigma \vdash \vec{S} \rightsquigarrow \Sigma'$$

$$\Sigma; \Gamma \vdash \tau \Leftarrow \mathit{type} \rightsquigarrow \Sigma', A$$

$$\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma', M$$

Elaborate a term

Key difference: A is output.

$$\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma', M$$

Σ is a signature: a list of fresh propositions, axioms, and inequalities.

Deciding conversion

One final foray into some theory.

- As indicated before, elaboration should be decidable.
- So we need to decide conversion in the core theory.
- Our approach: normalization
- Our approach to this approach: Synthetic Tait Computability

The hard bit: the conditional rule for extension types

Unstable neutrals

- Crucial step in normalization proofs: carve out renamings
- Big problem: the neutrality of $\textbf{out}(e)$ isn't stable under renamings

Unstable neutrals

- Crucial step in normalization proofs: carve out renamings
- Big problem: the neutrality of **out**(e) isn't stable under renamings

A renaming could make a proposition true, so **out**(e) should reduce.

Unstable neutrals

- Crucial step in normalization proofs: carve out renamings
- Big problem: the neutrality of **out**(e) isn't stable under renamings

- Authors 2 & 3 already considered STC for cubical type theory (similar problems)
- Reuse a key idea: unstable neutrals
- TLDR: type theory with extension types & partial element types enjoys normalization.

Implementations

Currently, there are two implementations of controlled unfolding:

- **`cooltt`**: already had extension types, implemented as described above.
  `https://github.com/RedPRL/cooltt`
- Agda: doesn't use extension types, implemented by Amélia Liao & Jesper Cockx (*now merged!*)
  `https://github.com/agda/agda/pull/6354`

The role of extension types

We can implement controlled unfolding without fancy types, so why bother with them?

- To structure the proof of decidability of conversion
- To guide us in various design choices (what is unfolded where)
- Give a reference for users to reason about to predict interactions

However, don't have to implement extension types to use controlled unfolding!

Conclusions

- We revisit the type-theoretic account of translucent definitions (singleton types)
- Refine this idea by replacing singleton types with extension types
- Show that extension types can be used to encode semi-translucent definitions
- Propose a surface syntax/elaboration mechanism

https://arxiv.org/abs/2210.05420