Controlling unfolding in type theory

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

WG6 Meeting 2023-04-23

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:

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

$\text{two} \triangleq 2$

Definitions should unfold.... definitionally

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

$\_ \triangleq \text{refl}$

Definitions in proof assistants

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

Crucial point:

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

$\text{two} \triangleq 2$

Definitions should unfold.... definitionally

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

$\_ \triangleq \text{refl}$

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
- Show that extension types can be used to encode semi-translucent definitions
- Propose a surface syntax/elaboration mechanism

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

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
- Show that extension types can be used to encode semi-translucent definitions
- Propose a surface syntax/elaboration mechanism

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

Interesting type theory to be found even in this most basic feature.

Singleton types: an account of translucent definitions

How does one express 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 containing only one element: M.

Singleton types: an account of translucent definitions II

For a given type $M : A$, we define the singleton type $S_A(M)$ by the following rules:

$$\frac{N : A \qquad M = N : A}{N : S_A(M)} \qquad \frac{N : S_A(M)}{N : A} \qquad \frac{N : S_A(M)}{N = M : A}$$

Singleton types: an account of translucent definitions II

For a given type $M : A$, we define the singleton type $S_A(M)$ by the following rules:

$$\frac{N : A \qquad M = N : A}{N : S_A(M)} \qquad \frac{N : S_A(M)}{N : A} \qquad \frac{N : S_A(M)}{N = M : A}$$

Singleton types: an account of translucent definitions II

For a given type $M : A$, we define the singleton type $S_A(M)$ by the following rules:

$$\frac{N : A \qquad M = N : A}{N : S_A(M)} \qquad \frac{N : S_A(M)}{N : A} \qquad \frac{N : S_A(M)}{N = M : A}$$

Hypothesizing over a variable $x : S_A(M) \leftrightsquigarrow$ working relative to $x : A \triangleq M$

Translucent definitions versus abstract definitions

Very roughly, we have the following:

- Opaque definitions:

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

- Translucent definitions:

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

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:

- Opaque definitions:

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

- Translucent definitions:

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

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 $\wedge$.

Notation and properties inspired by cofibrations from cubical type theory.

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

Working with $\mathbb{F}$

New form of context: Γ, ϕ.

New form of judgment $\Gamma \vdash \phi$ *true*:

$$\frac{\phi \in \Gamma}{\Gamma \vdash \phi \textit{ true}} \qquad \frac{}{\Gamma \vdash \top \textit{ true}}$$

$$\frac{\Gamma \vdash \phi \textit{ true} \qquad \Gamma \vdash \psi \textit{ true}}{\Gamma \vdash \phi \wedge \psi \textit{ true}} \qquad \frac{\Gamma \vdash \phi \wedge \psi \textit{ true}}{\Gamma \vdash \phi \textit{ true} \qquad \Gamma \vdash \psi \textit{ true}}$$

"Very strict": user never has to write proofs for elements of $\mathbb{F}$.

Two new type formers: partial element types

$$\frac{\phi : A \text{ type}}{\phi \to A \text{ type}}$$

$$\frac{\phi \vdash M : A}{\langle \phi \rangle M : \phi \to A} \qquad \frac{M : \phi \to A \qquad \phi \text{ true}}{M! : A}$$

Normal β/η rules

Two new type formers: partial element types

No proof of ϕ explicitly given.

$$\frac{\phi : A \text{ type}}{\phi \to A \text{ type}}$$

$$\frac{\phi \vdash M : A}{\langle \phi \rangle M : \phi \to A} \qquad \frac{M : \phi \to A \qquad \phi \text{ true}}{M! : A}$$

Normal β/η rules

Two 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

Two 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

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

Two 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

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

Extension types generalize singleton types

We can make good on an earlier promise:

$$S_A(M) = \{A \mid \top \hookrightarrow M\}$$

$\top$ is always true, so

$$\frac{N : S_A(M)}{\mathsf{out}(N) = M : A}$$

We haven't added $\bot$, but if we did we could prove $\{A \mid \bot \hookrightarrow M\} \cong A$

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.
- In our above language, dictionary, 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.
- In our above language, dictionary, 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. $y_0 \dots y_n$
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.

$$\begin{aligned}&\mathrm{neg} : \mathbb{Z} \to \mathbb{Z}\\&\mathrm{neg} \triangleq \ldots\end{aligned}$$

$$\begin{aligned}&\textbf{prop}\ \Upsilon_{\mathrm{neg}}\\&\textbf{axiom}\ \mathrm{neg} : \{\mathbb{Z} \to \mathbb{Z} \mid \Upsilon_{\mathrm{neg}} \hookrightarrow \ldots\}\end{aligned}$$

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

Some key points

This is the beginning of some informal elaboration strategy

- Automatically "type-safe"
- Automatically invariant under conversion (replacing equals by equals)
- Equations are *definitional* and don't produce coherence hell!

A small tangent

One nice example of how this methodology helps:

Q. Does unfolding A in B allow this unfolding in the type of B?

A. No! Extension types require the type to be fully defined!

Crucial point, otherwise uses might be ill-formed!

Two forms of dependence

This translation surfaces two ways to use a prior definition:

- Opaque usage
- Transparent usage

Two forms of dependence

This translation surfaces two ways to use a prior definition:

- Opaque usage
- Transparent usage

Just caring about the type; default usage

Two forms of dependence

This translation surfaces two ways to use a prior definition:

- Opaque usage
- Transparent usage

Caring about every single aspect of the definition; occasionally necessary

Two forms of dependence

This translation surfaces two ways to use a prior definition:

- Opaque usage
- Transparent usage

Crystallized by whether we require $\Upsilon_x \leq \Upsilon_y$.

Two forms of dependence II

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

Two forms of dependence II

Suppose A depends on B depends on C.

- If A $\to$ B is transparent and B $\to$ C is transparent, so is A $\to$ 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.

Two forms of dependence II

Necessary for "subject reduction".

Suppose A depends on B depends on C.

- If A $\to$ B is transparent and B $\to$ C is transparent, so is A $\to$ 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!
- Still automatically type safe & respects conversions

Evaluating this mechanism

- Using extension types automatically ensures we unfold “just enough”
- Unless requested, nothing will unfold!
- Still automatically type safe & respects conversions

Not a panacea

- Currently at the granularity of definitions
- Writing these extension types is weird
- Within a scope, something unfolds always or never unfolds (no single-stepping.)

Evaluating this mechanism

- Using extension types automatically ensures we unfold "just enough"
- Unless requested, nothing will unfold!
- Still automatically type safe & respects conversions

Not a panacea

- Currently at the granularity of definitions
- Writing these extension types is weird
- Within a scope, something unfolds always or never unfolds (no single-stepping.)

Solved through elaboration!

Future work. *shrug*

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

Surface syntax for unfolding II

We will define the surface syntax by elaboration.

- No typing judgments per se, just elaboration judgments
- Tautologically, well-formed surface programs produce well-formed core terms

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

$\quad$ [**abbreviation**] **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:

foo : A

[**abbreviation**] **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

foo $\triangleq$ M

Normal components of a definition

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

foo : A

[**abbreviation**] **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$ ← What is unfolded in M

foo $\triangleq M$

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

foo : A

[**abbreviation**] **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$

foo $\triangleq M$

Does unfolding $\mathsf{bar}_0 \ldots$ unfold foo

Anatomy of a surface-level definition

A surface-level definition consists of the following parts:

$\mathsf{foo} : A$

$\quad$ [**abbreviation**] **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

How do we elaborate abbreviations?

Most of this is familiar, except **abbreviation**.

Almost identical, except:

$$\Upsilon_{\mathsf{foo}} = \bigwedge_i \Upsilon_{\mathsf{bar}_i}$$

Now if all bar_i unfold, foo will unfold automatically.

How do we elaborate abbreviations?

Most of this is familiar, except **abbreviation**.

Almost identical, except:

$$\Upsilon_{\mathsf{foo}} = \bigwedge_i \Upsilon_{\mathsf{bar}_i}$$

Now if all bar_i unfold, foo will unfold automatically.

Returning to 3-for-2, this gives us one of the two outstanding implications.

Conveniences

- Many, many convenience features are possible.
- We'll settle for one: local unfolds

TLDR: a construct to create a local scope where a definition unfolds.

A warmup: a design pattern with unfolding

What if we *do* want something to unfold in a type?

- Obvious issue: could this be used without this unfolding?
- Potentially yes...
- ... provided no details of the type were exposed

Just create an auxiliary definition for the type which unfolds things.

A warmup: a design pattern with unfolding II

$\text{two} \triangleq 2$

$\texttt{tp} : \mathcal{U}$ **abbreviation unfolding** two

$\texttt{tp} \triangleq (p : \text{two} = 2) \to p = \text{refl}$

$\text{contr} : \texttt{tp}$

$\text{contr} \triangleq \ldots$

- If two isn't unfoldable, well-formed but useless.
- If two is unfoldable, vanishes definitionally.

Local unfold syntax

The basic idea: a new expression form

$$\textbf{unfold}\ \mathsf{foo}\ \textbf{in}\ M$$

Local unfold syntax

The basic idea: a new expression form

$$\textbf{unfold}\ \mathsf{foo}\ \textbf{in}\ M$$

$\mathsf{two} \triangleq 2$

$\mathsf{contr} : \textbf{unfold}\ \mathsf{two}\ \textbf{in}\ (p : \mathsf{two} = 2) \to p = \mathsf{refl}$
$\mathsf{contr} \triangleq \ldots$

Realizing local unfolding

A few complications

- What should this expression be equal to?
- What about the type of M?
- Type may not even be well-formed without some unfolding...

Local unfold through elaboration

Roughly, we elaborate local unfolds by hoisting:

- Elaborating one definition can yield multiple constants
- Each local unfold will yield a constant

Local unfold through elaboration

Roughly, we elaborate local unfolds by hoisting:

- Elaborating one definition can yield multiple constants
- Each local unfold will yield a constant

To elaborate

$\mathsf{foo} : B \ \textbf{unfolding} \ \mathsf{bar}_0 \ldots \mathsf{bar}_n$
$\mathsf{foo} \triangleq N(\textbf{unfold} \ \mathsf{bar} \ \textbf{in} \ M)$

Will produce/use the following:

$$\textbf{axiom} \ \mathsf{hoisted} : \Upsilon_{\mathsf{bar}_0} \to \cdots \to \Upsilon_{\mathsf{bar}_n} \to \{A \mid \Upsilon_{\mathsf{bar}} \hookrightarrow M\}$$

Replace **unfold** bar **in** M with $\mathsf{out}(\mathsf{hoisted})! \cdots !$

Local unfold through elaboration

Roughly, we elaborate local unfolds by hoisting:

- Elaborating one definition can yield multiple constants
- Each local unfold will yield a constant

To elaborate

foo : B **unfolding** $\mathsf{bar}_0 \ldots \mathsf{bar}_n$
foo $\triangleq N(\textbf{unfold}\ \mathsf{bar}\ \textbf{in}\ M)$

Local unfolds need partial element types!

Will produce/use the following:

$$\textbf{axiom}\ \mathsf{hoisted} : \Upsilon_{\mathsf{bar}_0} \to \cdots \to \Upsilon_{\mathsf{bar}_n} \to \{A \mid \Upsilon_{\mathsf{bar}} \hookrightarrow M\}$$

Replace **unfold** bar **in** M with $\mathsf{out}(\mathsf{hoisted})! \cdots !$

Equations with local unfolding

- If they're blocked, a local unfold is generative
- If definition does unfold, local unfold is definitionally equal to the body
- Similar to pattern-matching in Agda

Equations with local unfolding

- If they're blocked, a local unfold is generative
- If definition does unfold, local unfold is definitionally equal to the body
- Similar to pattern-matching in Agda

Still easy to reason about: just encoding a common design pattern.

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

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

Decidable iff conversion in the core language is decidable, so normalization

Signatures in the core language

Output of elaboration must be a *signature* in the core language

- Bind fresh proposition
- Force equalities or inequalities of propositions
- Bind axioms of a given type

$$
\begin{array}{llll}
\textit{(signatures)} & \Sigma & ::= & \epsilon \mid \Sigma, D \\
\textit{(declarations)} & D & ::= & \textbf{axiom}\, x : A \mid \textbf{prop}\, p \leq q \mid \textbf{prop}\, p = q
\end{array}
$$

Signatures in the core language

Output of elaboration must be a *signature* in the core language

- Bind fresh proposition
- Force equalities or inequalities of propositions
- Bind axioms of a given type

(signatures)	Σ	$::=$	$\epsilon \mid \Sigma, D$
(declarations)	D	$::=$	$\mathbf{axiom}\, x : A \mid \mathbf{prop}\, p \leq q \mid \mathbf{prop}\, p = q$

Not explained: signatures induce a context ("A is well-formed wrt Σ").

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

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

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

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

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

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

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

Main judgment; essentially `flatMap`

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

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

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

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

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

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 term

Key difference: A is output.

The judgments for elaboration

Elaboration is controlled by 4 key judgments:

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

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

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

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

Bidirectionalism minimizes user-provided annotations.

An example: one rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e}_0 \Rightarrow (x : A) \to B(x) \rightsquigarrow \Sigma_1; M \qquad \Sigma; \Gamma \vdash \mathsf{e}_1 \Leftarrow A \rightsquigarrow \Sigma_2; N}{\Sigma; \Gamma \vdash \mathsf{e}_0(\mathsf{e}_1) \Rightarrow B[N/x] \rightsquigarrow \Sigma_1, \Sigma_2; M(N)}$$

Read this top-down.

- Elaborate e_0, get the type $(x : A) \to B(x)$ along with the M
- Elaborate e_1 using the type we just computed from e_0
- Combine the computed signatures & use the appropriate core term.

An example: one rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e}_0 \Rightarrow (x : A) \to B(x) \rightsquigarrow \Sigma_1; M \qquad \Sigma; \Gamma \vdash \mathsf{e}_1 \Leftarrow A \rightsquigarrow \Sigma_2; N}{\Sigma; \Gamma \vdash \mathsf{e}_0(\mathsf{e}_1) \Rightarrow B[N/x] \rightsquigarrow \Sigma_1, \Sigma_2; M(N)}$$

Read this top-down.

- Elaborate e_0, get the type $(x : A) \to B(x)$ along with the M
- Elaborate e_1 using the type we just computed from e_0
- Combine the computed signatures & use the appropriate core term.

An example: one rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e}_0 \Rightarrow (x : A) \to B(x) \rightsquigarrow \Sigma_1; M \qquad \Sigma; \Gamma \vdash \mathsf{e}_1 \Leftarrow A \rightsquigarrow \Sigma_2; N}{\Sigma; \Gamma \vdash \mathsf{e}_0(\mathsf{e}_1) \Rightarrow B[N/x] \rightsquigarrow \Sigma_1, \Sigma_2; M(N)}$$

Read this top-down.

- Elaborate e_0, get the type $(x : A) \to B(x)$ along with the M
- Elaborate e_1 using the type we just computed from e_0
- Combine the computed signatures & use the appropriate core term.

An example: one rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e}_0 \Rightarrow (x : A) \to B(x) \rightsquigarrow \Sigma_1; M \qquad \Sigma; \Gamma \vdash \mathsf{e}_1 \Leftarrow A \rightsquigarrow \Sigma_2; N}{\Sigma; \Gamma \vdash \mathsf{e}_0(\mathsf{e}_1) \Rightarrow B[N/x] \rightsquigarrow \Sigma_1, \Sigma_2; M(N)}$$

Read this top-down.

- Elaborate e_0, get the type $(x : A) \to B(x)$ along with the M
- Elaborate e_1 using the type we just computed from e_0
- Combine the computed signatures & use the appropriate core term.

An example: one medium scary rule

(Mostly to convince you that someone considered this)

$$\frac{\Sigma;\Gamma,\Upsilon_\vartheta \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_1; M \qquad \textbf{let } \chi := \textit{gensym}() \qquad \textbf{let } \Sigma_2 := \Sigma_1, \textbf{axiom}\,\chi : \prod_\Gamma \{A \mid \Upsilon_\vartheta \hookrightarrow M\}}{\Sigma;\Gamma \vdash \textbf{unfold } \vartheta \textbf{ in } \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_2; \mathsf{out}_{\Upsilon_\vartheta}\, \chi[\Gamma]}$$

Read this top-down.

- Recursively elaborate e, get some core term $M : A$
- Close up M; extend Σ_1 with hoisted-out constant.
- Output signature is extended Σ_1 & output term uses new constant.

An example: one medium scary rule

(Mostly to convince you that someone considered this)

$$\frac{\begin{array}{c}\Sigma; \Gamma, \Upsilon_\vartheta \vdash \mathsf{e} \Leftarrow A \leadsto \Sigma_1; M \\ \textbf{let } \chi := \mathit{gensym}() \\ \textbf{let } \Sigma_2 := \Sigma_1, \textbf{axiom}\, \chi : \prod_\Gamma \{A \mid \Upsilon_\vartheta \hookrightarrow M\}\end{array}}{\Sigma; \Gamma \vdash \textbf{unfold } \vartheta \textbf{ in } \mathsf{e} \Leftarrow A \leadsto \Sigma_2; \mathsf{out}_{\Upsilon_\vartheta}\, \chi[\Gamma]}$$

Read this top-down.

- Recursively elaborate e, get some core term $M : A$
- Close up M; extend Σ_1 with hoisted-out constant.
- Output signature is extended Σ_1 & output term uses new constant.

An example: one medium scary rule

(Mostly to convince you that someone considered this)

$$\frac{\Sigma; \Gamma, \Upsilon_\vartheta \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_1; M \qquad \textbf{let } \chi := \mathit{gensym}() \qquad \textbf{let } \Sigma_2 := \Sigma_1, \textbf{axiom } \chi : \prod_\Gamma \{A \mid \Upsilon_\vartheta \hookrightarrow M\}}{\Sigma; \Gamma \vdash \textbf{unfold } \vartheta \textbf{ in } \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_2; \mathsf{out}_{\Upsilon_\vartheta}\, \chi[\Gamma]}$$

Read this top-down.

- Recursively elaborate e, get some core term $M : A$
- Close up M; extend Σ_1 with hoisted-out constant.
- Output signature is extended Σ_1 & output term uses new constant.

An example: one medium scary rule

(Mostly to convince you that someone considered this)

$$\frac{\Sigma; \Gamma, \Upsilon_{\vartheta} \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_1; M \qquad \textbf{let } \chi := \mathit{gensym}() \qquad \textbf{let } \Sigma_2 := \Sigma_1, \textbf{axiom } \chi : \prod_{\Gamma} \{A \mid \Upsilon_{\vartheta} \hookrightarrow M\}}{\Sigma; \Gamma \vdash \textbf{unfold } \vartheta \textbf{ in } \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_2; \mathsf{out}_{\Upsilon_{\vartheta}}\, \chi[\Gamma]}$$

Read this top-down.

- Recursively elaborate e, get some core term $M : A$
- Close up M; extend Σ_1 with hoisted-out constant.
- Output signature is extended Σ_1 & output term uses new constant.

An example: one very scary rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma_1; M \qquad \Gamma \vdash A = B \,\mathsf{type}}{\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma_1; M}$$

- Recursively elaborate e, get some core term M and type A
- Ensure the term we're checking against matches the synthesized type

An example: one very scary rule

$$\frac{\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma_1; M \qquad \Gamma \vdash A = B \ \mathsf{type}}{\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma_1; M}$$

- Recursively elaborate e, get some core term M and type A
- Ensure the term we're checking against matches the synthesized type

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 **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

Normalization results

TLDR: type theory with extension types & partial element types enjoys normalization.

Further details are banished to bonus slides.

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
  https://github.com/agda/agda/pull/6354

(Interested in adding controlled unfolding to your proof assistant? I'm around.)

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!

What have I ignored?

A few interesting questions remain...

- What's the best way for this to interact with unification?
- Can we describe unfolding recursive definitions only a fixed number of times?
- What about data types? Can we interpolate between Σ's and records?
- What other features of proof assistants benefit from this attention?

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

STC

- Work internally to a presheaf topos to define the normalization model
- Each type former is modeled in turn, as a sequence of programming exercises.
- Each type is equipped with reify/reflect operations.
- Used for cubical type theory, multimodal type theory, and ∞-type theories.

Cubical type theory is the most relevant: it also has extension types.

Stabilized neutrals

The proof of normalization is almost standard, except for aforementioned issue.

- Standard normalization uses normals and neutrals
- We can't have neutrals, but we can have neutrals keyed by a proposition
- Big idea: proposition represents when the neutral *isn't* meaningful

Key case: the neutral for out_ϕ is associated to ϕ.

Stabilized neutrals II

Reflect function becomes more complicated:

$$
\begin{aligned}
&\mathsf{reflect} : \\
&\quad (M : \mathsf{Tm}(A))(\phi : \mathbb{P})\,(e : \mathsf{Ne}\,A\,\phi\,M)\,(M^{\bullet} : \phi \to \mathsf{Tm}^{\bullet}(A, M)) \\
&\quad \to \{\mathsf{Tm}^{\bullet}(A, M) \mid \phi \hookrightarrow M^{\bullet}\}
\end{aligned}
$$

Informally: you just provide the answer when the neutral doesn't help.