Controlling unfolding in type theory

Daniel Gratzer **Jonathan Sterling** Carlo Angiuli
Thierry Coquand Lars Birkedal

Aarhus University

December 14, 2022

Funded by
the European Union

Congratulations, Dr. Loïc Pujet!

table of contents

definitional equality is terrible...
but it does something important.

definitional equality is terrible...
but it does something important.

definitional equality is the ***boundary***
between machine and human.

definitional equality is terrible...
but it does something important.

definitional equality is the ***boundary***
between machine and human.

failures of definitional equality are not negotiable: when it says "no", you know for sure your code is wrong. **feature, not bug!**

definitional equality is terrible…
but it does something important.

definitional equality is the ***boundary***
between machine and human.

failures of definitional equality are not negotiable: when it says "no", you know for sure your code is wrong. **feature, not bug!**

e.g. if **reflexivity** fails, you do not need
to ask the non-question "What if I break
this into several **reflexivity** steps?" 😵‍💫

what is *conversion*?

what is *conversion*? conversion is the *implementation* of definitional equality as part of an elaborator or type checker.

what is *conversion*? conversion is the *implementation* of definitional equality as part of an elaborator or type checker.

well-known design constraint: it is *not optional* for conversion to be both **sound** and **complete** for definitional equality.

what is *conversion*? conversion is the *implementation* of definitional equality as part of an elaborator or type checker.

well-known design constraint: it is *not optional* for conversion to be both **sound** and **complete** for definitional equality.

incompleteness usually means **failures of transitivity** and/or subject reduction. in that direction lies irrational debates with the type checker, muddling the division of labor between human and machine. tail wags dog!

what is *conversion*? conversion is the *implementation* of definitional equality as part of an elaborator or type checker.

well-known design constraint: it is *not optional* for conversion to be both **sound** and **complete** for definitional equality.

incompleteness usually means **failures of transitivity** and/or subject reduction. in that direction lies irrational debates with the type checker, muddling the division of labor between human and machine. tail wags dog!

less well-known design constraint: elaboration / type checking must **respect** definitional equality.

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

1. assume $\Gamma\ \mathit{ctx}$;
2. assume that R is a finite mapping of names “x” to well-typed terms $\Gamma \vdash R.\mathrm{tm}\,(\text{“}x\text{”}) : R.\mathrm{tp}\,(\text{“}x\text{”})$;
3. assume $\Gamma \vdash A\ \mathit{type}$;
4. assume “*e*” is raw code (*e.g.* string or sexpr);
5. guarantee M is a well-typed term $\Gamma \vdash M : A$.

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

1. assume Γ *ctx*;
2. assume that R is a finite mapping of names “x” to well-typed terms $\Gamma \vdash R.\mathrm{tm}\,(\text{“}x\text{”}) : R.\mathrm{tp}\,(\text{“}x\text{”})$;
3. assume $\Gamma \vdash A$ *type*;
4. assume “*e*” is raw code (*e.g.* string or sexpr);
5. guarantee M is a well-typed term $\Gamma \vdash M : A$.

incorrect: it does not respect definitional equivalence

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;

$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;
2. assume $R : \mathrm{Name} \longrightarrow_{\mathit{fin.}} \mathbf{El}\,\Gamma$;

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;
2. assume $R : \mathrm{Name} \longrightarrow_{\mathit{fin.}} \mathbf{El}\,\Gamma$;
3. assume $A \in \mathbf{Tp}\,\Gamma$;

$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;
2. assume $R : \mathrm{Name} \longrightarrow_{\mathit{fin.}} \mathbf{El}\,\Gamma$;
3. assume $A \in \mathbf{Tp}\,\Gamma$;
4. assume $\text{“}e\text{”} \in \mathrm{Code}$;

$$\Gamma; R \vdash A \ni \text{"e"} \rightsquigarrow M$$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;
2. assume $R : \mathrm{Name} \longrightarrow_{\mathit{fin.}} \mathbf{El}\,\Gamma$;
3. assume $A \in \mathbf{Tp}\,\Gamma$;
4. assume $\text{“}e\text{”} \in \mathrm{Code}$;
5. guarantee $M \in \mathbf{El}\,\Gamma$ with $\pi_\Gamma M = A$.

$$\Gamma; R \vdash A \ni \text{“e”} \rightsquigarrow M$$

Let $(\mathscr{C}, \pi : \mathbf{El} \longrightarrow \mathbf{Tp})$ be the *syntactic cwf*.

1. assume $\Gamma \in \mathbf{ob}_{\mathscr{C}}$;
2. assume $R : \mathsf{Name} \longrightarrow_{\mathit{fin.}} \mathbf{El}\,\Gamma$;
3. assume $A \in \mathbf{Tp}\,\Gamma$;
4. assume $\text{“}e\text{”} \in \mathsf{Code}$;
5. guarantee $M \in \mathbf{El}\,\Gamma$ with $\pi_\Gamma M = A$.

correct: it respects definitional equivalence
(the syntactic cwf is quotiented)

why is this important?

definitional equality is the ground truth in the user's naïve mental model for elaboration (*cf*. "notional machines" in computer science education).

if elaboration disrespects definitional equality, the user is forced to think of their environment as *code* rather than *meaning*, which means **we have failed** to facilitate clear thinking (our singular goal).

unstable elaboration *qua* tactic execution is a huge source of brittleness in mechanized proofs in dependent type theory.

unfoldable definitions and definitional equality

a major source of instability is *unfoldable definitions* in type theoretic proof assistants, *e.g*. when the goal contains a defined term.

although *definiendum* is equal to *definiens*, elaboration / tactics can tell the difference between them: see proof scripts that fail after unfolding something.

one way to fix this is to have the elaborator always unfold everything, but this is bad for usability and for performance.

our solution: a calculus of top-level definitions for which the definiendum equals its definiens only in *certain* contexts.

table of contents

type theory with controlled unfolding

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n :\equiv n$$
$$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$$

type theory with controlled unfolding

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n :\equiv n$$
$$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$$

data vec n A **where**

 vnil : vec ze A

 vcons : $A \to \text{vec}\ n\ A \to \text{vec}\ (\text{su}\ n)\ A$

type theory with controlled unfolding

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n :\equiv n$$
$$\mathsf{su}\; m + n :\equiv \mathsf{su}\,(m + n)$$

data vec n A **where**

$$\mathsf{vnil} : \mathsf{vec}\;\mathsf{ze}\;A$$
$$\mathsf{vcons} : A \to \mathsf{vec}\;n\;A \to \mathsf{vec}\,(\mathsf{su}\;n)\;A$$

$$(\oplus) : \mathsf{vec}\;m\;A \to \mathsf{vec}\;n\;A \to \mathsf{vec}\,(m + n)\;A$$
$$\mathsf{vnil} \oplus v :\equiv \boxed{?: \mathsf{vec}\,(\mathsf{ze} + n)\;A}$$
$$\mathsf{vcons}\;a\;u \oplus v :\equiv \boxed{?: \mathsf{vec}\,(\mathsf{su}\;m + n)\;A}$$

type theory with controlled unfolding

$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathsf{ze} + n :\equiv n$
$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$

data vec n A **where**
vnil : vec ze A
vcons : $A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$

$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$\mathsf{vnil} \oplus v :\equiv v$ **Error:** $\mathsf{ze} + n \not\equiv n$
$\mathsf{vcons}\ a\ u \oplus v :\equiv$ $? : \mathsf{vec}\ (\mathsf{su}\ m + n)\ A$

type theory with controlled unfolding

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n :\equiv n$$
$$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$$

data vec n A **where**

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

$$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$$
$$\mathsf{vnil} \oplus v :\equiv \boxed{?: \mathsf{vec}\ (\mathsf{ze} + n)\ A}$$
$$\mathsf{vcons}\ a\ u \oplus v :\equiv \boxed{?: \mathsf{vec}\ (\mathsf{su}\ m + n)\ A}$$

type theory with controlled unfolding

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n :\equiv n$$
$$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$$

data vec n A **where**

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

$(\oplus)$ **unfolds** $(+)$

$$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$$
$$\mathsf{vnil} \oplus v :\equiv \boxed{?: \mathsf{vec}\ n\ A}$$
$$\mathsf{vcons}\ a\ u \oplus v :\equiv \boxed{?: \mathsf{vec}\ (\mathsf{su}\ (m + n))\ A}$$

type theory with controlled unfolding

$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathsf{ze} + n :\equiv n$
$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$

data vec n A **where**
  vnil : vec ze A
  vcons : $A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$\mathsf{vnil} \oplus v :\equiv v$
$\mathsf{vcons}\ a\ u \oplus v :\equiv$ $\boxed{?: \mathsf{vec}\ (\mathsf{su}\ (m + n))\ A}$

type theory with controlled unfolding

$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathsf{ze} + n :\equiv n$
$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$

data $\mathsf{vec}\ n\ A$ **where**
$\mathsf{vnil} : \mathsf{vec}\ \mathsf{ze}\ A$
$\mathsf{vcons} : A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$\mathsf{vnil} \oplus v :\equiv v$
$\mathsf{vcons}\ a\ u \oplus v :\equiv \mathsf{vcons}\ a\ \boxed{? : \mathsf{vec}\ (m + n)\ A}$

type theory with controlled unfolding

$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathsf{ze} + n :\equiv n$
$\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)$

data vec n A **where**
  vnil : vec ze A
  vcons : $A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$\mathsf{vnil} \oplus v :\equiv v$
$\mathsf{vcons}\ a\ u \oplus v :\equiv \mathsf{vcons}\ a\ (u \oplus v)$

more complex dependencies

$$\begin{aligned}
&\mathrm{map} : (A \to B) \to \mathrm{vec}\ n\ A \to \mathrm{vec}\ n\ B \\
&\mathrm{map}\ f\ \mathrm{vnil} :\equiv \mathrm{vnil} \\
&\mathrm{map}\ f\ (\mathrm{vcons}\ a\ u) :\equiv \mathrm{vcons}\ (f\ a)\ (\mathrm{map}\ f\ u)
\end{aligned}$$

more complex dependencies

$$\mathsf{map} : (A \longrightarrow B) \longrightarrow \mathsf{vec}\ n\ A \longrightarrow \mathsf{vec}\ n\ B$$
$$\mathsf{map}\ f\ \mathsf{vnil} :\equiv \mathsf{vnil}$$
$$\mathsf{map}\ f\ (\mathsf{vcons}\ a\ u) :\equiv \mathsf{vcons}\ (f\ a)\ (\mathsf{map}\ f\ u)$$

$$\begin{aligned}&\mathsf{map}\text{-}\oplus\\ &\quad : (f : A \longrightarrow B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A)\\ &\quad \longrightarrow \mathsf{map}\ f\ (u \oplus v) = \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v\end{aligned}$$
$$\mathsf{map}\text{-}\oplus\ f\ \mathsf{vnil}\ v :\equiv$$
$$\quad ? : \mathsf{map}\ f\ (\mathsf{vnil} \oplus v) = \mathsf{map}\ f\ \mathsf{vnil} \oplus \mathsf{map}\ f\ v$$

...

more complex dependencies

$$\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ n\ B \\
&\mathsf{map}\ f\ \mathsf{vnil} :\equiv \mathsf{vnil} \\
&\mathsf{map}\ f\ (\mathsf{vcons}\ a\ u) :\equiv \mathsf{vcons}\ (f\ a)\ (\mathsf{map}\ f\ u)
\end{aligned}$$

map-$\oplus$ **unfolds** map

$$\begin{aligned}
&\mathsf{map}\text{-}\oplus \\
&\quad : (f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \\
&\quad \to \mathsf{map}\ f\ (u \oplus v) = \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v \\
&\mathsf{map}\text{-}\oplus\ f\ \mathsf{vnil}\ v :\equiv \\
&\quad \boxed{? : \mathsf{map}\ f\ (\mathsf{vnil} \oplus v) = \mathsf{vnil} \oplus \mathsf{map}\ f\ v}
\end{aligned}$$

...

more complex dependencies

$$\mathsf{map} : (A \longrightarrow B) \longrightarrow \mathsf{vec}\ n\ A \longrightarrow \mathsf{vec}\ n\ B$$
$$\mathsf{map}\ f\ \mathsf{vnil} :\equiv \mathsf{vnil}$$
$$\mathsf{map}\ f\ (\mathsf{vcons}\ a\ u) :\equiv \mathsf{vcons}\ (f\ a)\ (\mathsf{map}\ f\ u)$$

$\mathsf{map}\text{-}\oplus$ **unfolds** $\mathsf{map}; (\oplus)$

$\mathsf{map}\text{-}\oplus$

$\quad : (f : A \longrightarrow B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A)$

$\quad \longrightarrow \mathsf{map}\ f\ (u \oplus v) = \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v$

$\mathsf{map}\text{-}\oplus\ f\ \mathsf{vnil}\ v :\equiv$

$\quad \boxed{?: \mathsf{map}\ f\ v = \mathsf{map}\ f\ v}$

...

more complex dependencies

$$
\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ n\ B\\
&\mathsf{map}\ f\ \mathsf{vnil} :\equiv \mathsf{vnil}\\
&\mathsf{map}\ f\ (\mathsf{vcons}\ a\ u) :\equiv \mathsf{vcons}\ (f\ a)\ (\mathsf{map}\ f\ u)
\end{aligned}
$$

map-⊕ **unfolds** map; (⊕)

$$
\begin{aligned}
&\mathsf{map}\text{-}\oplus\\
&\quad : (f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A)\\
&\quad \to \mathsf{map}\ f\ (u \oplus v) = \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v\\
&\mathsf{map}\text{-}\oplus\ f\ \mathsf{vnil}\ v :\equiv\\
&\quad \mathsf{refl}
\end{aligned}
$$

...

more complex dependencies

$$
\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ n\ B \\
&\mathsf{map}\ f\ \mathsf{vnil} :\equiv \mathsf{vnil} \\
&\mathsf{map}\ f\ (\mathsf{vcons}\ a\ u) :\equiv \mathsf{vcons}\ (f\ a)\ (\mathsf{map}\ f\ u)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{map}\text{-}\oplus\ \textbf{unfolds}\ \mathsf{map}; (\oplus) \\
&\mathsf{map}\text{-}\oplus \\
&\quad : (f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \\
&\quad \to \mathsf{map}\ f\ (u \oplus v) = \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v \\
&\mathsf{map}\text{-}\oplus\ f\ \mathsf{vnil}\ v :\equiv \\
&\quad \mathsf{refl} \\
&\cdots
\end{aligned}
$$

Note that unfolding $(\oplus)$ must transitively unfold $(+)$, since the body of $(\oplus)$ is only well-typed under this equation.

abbreviations

we recover the behavior of naïve definitional extensions (which always unfold) using *abbreviations*.

abbreviation singleton
singleton : $A \longrightarrow$ vec (su ze) A
singleton a :≡ vcons a vnil

abbreviations

we recover the behavior of naïve definitional extensions (which always unfold) using *abbreviations*.

abbreviation singleton
singleton : $A \to$ vec (su ze) A
singleton a :≡ vcons a vnil

test : singleton 5 = vcons 5 vnil
test :≡ ? : vcons 5 vnil = vcons 5 vnil

abbreviations

we recover the behavior of naïve definitional extensions (which always unfold) using *abbreviations*.

abbreviation singleton
singleton : $A \longrightarrow$ vec (su ze) A
singleton a :≡ vcons a vnil

test : singleton 5 = vcons 5 vnil
test :≡ refl

unfolding within types

what if the *type* of a definition needs to unfold something?

$$\oplus\text{-L} : (u : \mathsf{vec}\ n\ A) \to \mathsf{vnil} \oplus u = {\color{red}u}$$

Error: $\mathsf{ze} + n \not\equiv n$

unfolding within types

what if the *type* of a definition needs to unfold something?

$\oplus$-L **unfolds** $(+)$ // adding this doesn't help
$\oplus\text{-L} : (u : \text{vec}\ n\ A) \to \text{vnil} \oplus u = u$ **Error:** $\text{ze} + n \not\equiv n$

remember that **unfolds** applies to the definiens, not the type.

unfolding within types

idea: define definiendum's type separately from definiens.

$$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \to \mathbf{type}$$
$$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = \boxed{? : \mathsf{vec}\ (\mathsf{ze} + n)\ A}$$

unfolding within types

idea: define definiendum's type separately from definiens.

$\oplus$-L-tp **unfolds** $(+)$
$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \longrightarrow \textbf{type}$
$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = \boxed{?: \mathsf{vec}\ n\ A}$

unfolding within types

idea: define definiendum's type separately from definiens.

$\oplus$-L-tp **unfolds** $(+)$
$\oplus\text{-L-tp} : \text{vec}\ n\ A \longrightarrow \textbf{type}$
$\oplus\text{-L-tp}\ u :\equiv \text{vnil} \oplus u = u$

unfolding within types

idea: define definiendum's type separately from definiens.

$\oplus$-L-tp **unfolds** $(+)$

$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \longrightarrow \mathbf{type}$

$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = u$

$\oplus\text{-L} : (u : \mathsf{vec}\ n\ A) \longrightarrow \oplus\text{-L-tp}\ u$

$\oplus\text{-L}\ u :\equiv \boxed{?\ : \oplus\text{-L-tp}\ u}$

unfolding within types

idea: define definiendum's type separately from definiens.

$\oplus$-L-tp **unfolds** $(+)$
$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \longrightarrow \mathbf{type}$
$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = u$

$\oplus$-L **unfolds** $\oplus$-L-tp
$\oplus\text{-L} : (u : \mathsf{vec}\ n\ A) \longrightarrow \oplus\text{-L-tp}\ u$
$\oplus\text{-L}\ u :\equiv \boxed{? : \mathsf{vnil} \oplus u = u}$

unfolding within types

idea: define definiendum's type separately from definiens.

$\oplus$-L-tp **unfolds** $(+)$
$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \longrightarrow \textbf{type}$
$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = u$

$\oplus$-L **unfolds** $\oplus$-L-tp; $(\oplus)$
$\oplus\text{-L} : (u : \mathsf{vec}\ n\ A) \longrightarrow \oplus\text{-L-tp}\ u$
$\oplus\text{-L}\ u :\equiv \boxed{? : u = u}$

unfolding within types

idea: define definiendum's type separately from definiens.

abbreviation $\oplus$-L-tp
$\oplus$-L-tp **unfolds** $(+)$
$\oplus\text{-L-tp} : \mathsf{vec}\ n\ A \longrightarrow \mathbf{type}$
$\oplus\text{-L-tp}\ u :\equiv \mathsf{vnil} \oplus u = u$

$\oplus$-L **unfolds** $(\oplus)$
$\oplus\text{-L} : (u : \mathsf{vec}\ n\ A) \longrightarrow \oplus\text{-L-tp}\ u$
$\oplus\text{-L}\ u :\equiv \boxed{? : u = u}$

unfolding within types

idea: define definiendum's type separately from definiens.

abbreviation $\oplus$-L-tp
$\oplus$-L-tp **unfolds** $(+)$
$\oplus$-L-tp : vec n A $\longrightarrow$ **type**
$\oplus$-L-tp u :≡ vnil $\oplus$ u $=$ u

$\oplus$-L **unfolds** $(\oplus)$
$\oplus$-L : (u : vec n A) $\longrightarrow$ $\oplus$-L-tp u
$\oplus$-L u :≡ refl

table of contents

MLTT over a bounded meet-semilattice

we consider a version of MLTT/LF extended by an (external) $(\top, \wedge)$-semilattice of propositions $\mathscr{P}$. For each $p \in \mathscr{P}$ we add:

- a new context-former Γ, p;
- the (implicit) dependent product $\Gamma \vdash \{p\}\, A$ for each $\Gamma, p \vdash A$;
- the *extension type* $\Gamma \vdash \{A \mid p \hookrightarrow a\}$ for each $\Gamma \vdash A$ and partial element $\Gamma, p \vdash a : A$.

(you could think of this as a type theory fibered over the category of $(\top, \wedge)$-semilattices)

extension types in MLTT+$\mathscr{P}$

$\boxed{\Gamma \vdash \{A \mid p \hookrightarrow a\}}$ is the subtype of A that contains an element $\boxed{\Gamma \vdash u : A}$ exactly when $\boxed{\Gamma, p \vdash u \equiv a : A}$ holds.

(Of course, no core type theory has “true” subtypes because these are not algebraic; but we leave the coercions implicit.)

top-level definitions will be elaborated to have extension type; the condition p will govern whether the definition is “unfolded”.

our idea: elaborate controlled unfolding to MLTT+$\mathscr{P}$

controlled unfolding is a *linguistic feature*, not a hack.

our overall goal to provide users with a ***reliable mental model*** for the relationship between their *code* and its *meaning*.

this mental model is a "notional elaborator":

i.e. an informal translation of code into MLTT+$\mathscr{P}$

that respects the definitional equality of MLTT+$\mathscr{P}$ terms in its environment

table of contents

a document is elaborated to a $(\top, \wedge)$-semilattice $\mathscr{P}$ together with a sequence of declarations Σ in MLTT+$\mathscr{P}$.

for convenience, we describe the latter simultaneously through declarations of the following form:

prop $p \leq q$
// given $q \in \mathscr{P}$, extends $(\mathscr{P}, \Sigma)$ to $(\mathscr{P}[p, p \leq q], \Sigma_{|\mathscr{P}[p,p\leq q]})$

prop $p = q$
// given $q \in \mathscr{P}$, extends $(\mathscr{P}, \Sigma)$ to $(\mathscr{P}[p, p = q], \Sigma_{|\mathscr{P}[p,p=q]})$

const $x : A$
// given $\Sigma; \Gamma \vdash_{\mathscr{P}} A$, extends $(\mathscr{P}, \Sigma)$ to $(\mathscr{P}, (\Sigma, x : A))$

(don't worry about pattern matching; we are really using eliminators.)

warm-up: elaborating $(+)$

we elaborate the following top-level definition:

$$
\begin{aligned}
&(+) : \mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{N} \\
&\mathsf{ze} + n :\equiv n \\
&\mathsf{su}\ m + n :\equiv \mathsf{su}\ (m + n)
\end{aligned}
$$

warm-up: elaborating $(+)$

to the following core signature:

$\delta_+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\delta_+ \; \mathsf{ze} \; n :\equiv n$
$\delta_+ \; (\mathsf{su} \; m) \; n :\equiv \mathsf{su} \; (\delta_+ \; m \; n)$

prop $p_+ \leq \top$
const $(+) : \{\mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid p_+ \hookrightarrow \delta_+\}$

elaborating top-level unfoldings

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \longrightarrow \mathsf{vec}\ n\ A \longrightarrow \mathsf{vec}\ (m+n)\ A$
$\mathsf{vnil} \oplus v :\equiv v$
$\mathsf{vcons}\ a\ u \oplus v :\equiv \mathsf{vcons}\ a\ (u \oplus v)$

elaborating top-level unfoldings

$$\delta_\oplus : \{p_+\}\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \longrightarrow \mathsf{vec}\ (m + n)\ A$$
$$\delta_\oplus\ \mathsf{vnil}\ v :\equiv v$$
$$\delta_\oplus\ (\mathsf{vcons}\ a\ u)\ v :\equiv \mathsf{vcons}\ a\ (\delta_\oplus\ u\ v)$$

$$\textbf{prop}\ p_\oplus \leq p_+$$
$$\textbf{const}\ (\oplus) : \{\mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A \mid p_\oplus \hookrightarrow \delta_\oplus\}$$

elaborating abbreviations

abbreviation $\oplus$-L-tp
$\oplus$-L-tp **unfolds** $(+)$
$\oplus$-L-tp : vec n A $\longrightarrow$ **type**
$\oplus$-L-tp u :≡ vnil $\oplus$ u $=$ u

elaborating abbreviations

$$\delta_{\oplus\text{-L-tp}} : \{p_+\}\ (u : \mathsf{vec}\ n\ A) \longrightarrow \textbf{type}$$
$$\delta_{\oplus\text{-L-tp}}\ u :\equiv \mathsf{vnil} \oplus u = u$$

$$\textbf{prop}\ p_{\oplus\text{-L-tp}} = p_+$$
$$\textbf{const}\ \oplus\text{-L-tp} : \{\mathsf{vec}\ n\ A \longrightarrow \textbf{type} \mid p_{\oplus\text{-L-tp}} \hookrightarrow \delta_{\oplus\text{-L-tp}}\}$$

table of contents

metatheory of MLTT+$\mathscr{P}$

Theorem (Normalization)

There is an syntactically defined set of normal forms that can be placed in bijection with equivalence classes of types/terms in MLTT+$\mathscr{P}$.

Corollary (Decidability)

If $\mathscr{P}$ is decidable, then definitional equality in MLTT+$\mathscr{P}$ is decidable.

Corollary (Inversion)

Type constructors Π, Σ, etc. are injective in MLTT+$\mathscr{P}$.

Corollary (Conservativity)

MLTT+$\mathscr{P}$ is conservative over MLTT.

synthetic NbE/NbG for MLTT+$\mathscr{P}$

all results follow from a **synthetic Tait computability** argument; as with cubical type theory (Sterling and Angiuli, 2021), the most important ingredient is *stabilized neutrals*.

- **new:** a constructive version of STC, computational content is a form of normalization-by-evaluation;
- **oops:** fixes a bug in the normalization structure of neutral types in my v1 of thesis (thanks Thierry & Daniel!)

implementation in **cooltt**

to test the usability of controlled unfolding, we implemented it in our experimental cubical proof assistant **cooltt**:

- **cooltt** already had extension types;
- **hack:** use the cubical interval; $\mathbb{I} \hookrightarrow \mathbb{F}$ to simulate $\mathscr{P}$;
- **Favonia** added inequalities to OCaml library **kado**,[1] which we use to solve the interval theory;
- **result:** very quick and non-invasive implementation! usability is promising, more experience needed;
- all definitions are **abbreviation** by default for backward-compatibility.

[1] `https://github.com/RedPRL/kado`

implementation in Agda

Amélia Liao and **Jesper Cockx** have implemented a version of controlled unfolding in **Agda**![2]

rather than using extension types, Agda computes dependency reachability directly. (this is a reasonable implementation strategy that will **also** work for systems like Coq!)

implementation semi-blocked on how to reconcile with the existing **abstract** feature.

[2] https://github.com/agda/agda/pull/6354

some thoughts on the future of type theory

there is sometimes a divide between hacking proof assistants, and thinking deeply about the syntax & semantics of type theory. we should strive to do both well.

we think too much in terms of code, and are constantly breaking subject reduction and transitivity of conversion, or proposing elaborations that disrespect definitional equality...

meanwhile, users increasingly demand predictability and reliability, and constantly complain about “brittleness”.

I believe most desirable things can be done in a way that respects definitional equality, but we must be creative and give ourselves the space to move slowly and deliberately.

thanks, and congratulations again!

table of contents

bibliography I

Gratzer, Daniel, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal (Oct. 2022). *Controlling unfolding in type theory*. DOI: 10.48550/arXiv.2210.05420. arXiv: 2210.05420 [cs.LO].

Norell, Ulf (Sept. 2007). "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology.

RedPRL Development Team (2020). *cooltt*. URL: https://www.github.com/RedPRL/cooltt.

Riehl, Emily and Michael Shulman (2017). "A type theory for synthetic ∞-categories". In: *Higher Structures* 1 (1), pp. 147–224. arXiv: 1705.07442 [math.CT]. URL: https://journals.mq.edu.au/index.php/higher_structures/article/view/36.

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

Sterling, Jonathan and Carlo Angiuli (July 2021). "Normalization for Cubical Type Theory". In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–15. DOI: 10.1109/LICS52264.2021.9470719. arXiv: 2101.11479 [cs.LO].

Uemura, Taichi (2021). "Abstract and Concrete Type Theories". PhD thesis. Amsterdam: Universiteit van Amsterdam. URL: https://www.illc.uva.nl/cms/Research/Publications/Dissertations/DS-2021-09.text.pdf.