Adapters

A type-theoretic foundation for type casting

CASH Seminar – October 17th 2025

Meven Lennon-Bertrand
Joint w/ Arthur Adjedj, Thibaut Benjamin, Théo Laurent, Kenji Maillard

Have many form of **type casting**...

Have many form of **type casting**... and they're quite **complicated**.

Parametrized coercions #2455

New issue

Open

coqbot opened on Dec 6, 2010 — Member

Note: the issue was created automatically with bugzilla2github tool

Original bug ID: BZ#2455
From: **@robbertkrebbers**
Reported version: trunk
CC: **@Eelis**, **@JasonGross**

coqbot on Dec 6, 2010 — Member, Author

Comment author: **@robbertkrebbers**

Creating a parametrized coercion does not work. For example, I want to create a coercion from the positive elements of an arbitrary ordered ring into that ring.

(* Running Coq trunk r13689 *)

Section test.
(* Imagine R to be an ordered Ring *)
Context (R : Type).

(* Here we define the positive elements using a sigma type *)
(* In this example we are lazy and just take them all :) *)
Definition Pos := sig (fun x : R => True).

Assignees

No one assigned

Labels

part: coercions

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

Code with agent mode

No branches or pull requests

Notifications Customize

Subscribe

You're not receiving notifications from this thread.

Coercions again #403

New issue

Closed

leodemoura opened on Apr 14, 2021 — Member

The Lean 4 coercions work much better than the Lean 3 ones, but they are still brittle and based on TC resolution.
"Bad" instances often trigger non-termination.
We can't support coercions from `A` to a subtype of `A` without allowing TC to invoke tactics, and we really don't want TC to invoke arbitrary tactics since it would make the system more complex, the caching mechanism will be less effective, and users will probably abuse the feature and create performance problems.
Finally, the TC rules are to strict and prevent us from finding a coercion for

```
structure Foo (A : Sort _) := (foo : A)
structure Bar (A : Sort _) extends Foo A := (bar : A)
instance {A} : Coe (Bar A) (Foo A) := {coe := Bar.toFoo}
def getFoo {A} (F : Foo A) := F.foo
def bar : Bar Nat := {foo := 0, bar := 1}

#check getFoo bar -- fails because the expected type `Foo ?A` contains a metavariable.
```

One option is to write an extensible coercion resolution procedure.
Users would still be able to define (non-dependent) coercions using `instance`s, but the search and support for dependent coercions from `Prop` to `Bool` and `A` to subtype of `A` would be handwritten.

leodemoura added refactoring on Apr 14, 2021

Assignees
No one assigned

Labels
refactoring

Type
No type

Projects
No projects

Milestone
No milestone

Relationships
None yet

Development
Code with agent mode
No branches or pull requests

Notifications Customize

The `norm_cast` family of tactics.

A full description of the tactic, and the use of each theorem category, can be found at https://arxiv.org/abs/2001.10594.

```
def Lean.Elab.Tactic.NormCast.proveEqUsing
      (s : Meta.SimpTheorems) (a b : Expr) :
   MetaM (Option Meta.Simp.Result)
```
source

Proves `a = b` using the given simp set.

▸ Equations

```
def Lean.Elab.Tactic.NormCast.proveEqUsingDown
      (a b : Expr) :
   MetaM (Option Meta.Simp.Result)
```
source

Proves `a = b` by simplifying using move and squash lemmas.

▸ Equations

```
def Lean.Elab.Tactic.NormCast.mkCoe
      (e ty : Expr) :
   MetaM Expr
```
source

Constructs the expression `(e : ty)`.

▸ Equations

```
def Lean.Elab.Tactic.NormCast.isCoeOf?
      (e : Expr) :
   MetaM (Option Expr)
```
source

Checks whether an expression is the coercion of some other expression, and if so returns that expression.

▸ Equations

return to top

source

▸ Imports
▸ Imported by

Lean.Elab.Tactic.NormCast.proveEqUsing
Lean.Elab.Tactic.NormCast.proveEqUsingDown
Lean.Elab.Tactic.NormCast.mkCoe
Lean.Elab.Tactic.NormCast.isCoeOf?
Lean.Elab.Tactic.NormCast.isNumeral?
Lean.Elab.Tactic.NormCast.splittingProcedure
Lean.Elab.Tactic.NormCast.prove
Lean.Elab.Tactic.NormCast.upwardAndElim
Lean.Elab.Tactic.NormCast.numeralToCoe
Lean.Elab.Tactic.NormCast.
elabNormCastConfig
Lean.Elab.Tactic.NormCast.derive
Lean.Elab.Tactic.NormCast.elabModCast
Lean.Elab.Tactic.NormCast.normCastTarget
Lean.Elab.Tactic.NormCast.normCastHyp
Lean.Elab.Tactic.NormCast.evalNormCast0
Lean.Elab.Tactic.NormCast.evalConvNormCast
Lean.Elab.Tactic.NormCast.evalPushCast
Lean.Elab.Tactic.NormCast.elabAddElim

Polarities: subtyping for datatypes #65

New issue

Closed

catalin-hritcu opened on Nov 29, 2014 — Member

`(x:int{x>1} * y:int{y>1})` is not a subtype of `(int * int)`

catalin-hritcu added kind/bug on Nov 29, 2014

24 remaining items

Load more

nikswamy on Mar 14, 2022 — Collaborator

Addressing this issue requires more research. Closing as a wontfix until then.

nikswamy closed this as completed on Mar 14, 2022

Assignees

No one assigned

Labels

component/language-design
component/metatheory
component/typechecker
hard
kind/enhancement
status/wont-fix

Relationships

None yet

Development

Code with agent mode

No branches or pull requests

Disable all subtyping by default? #4474

New issue

Closed

jespercockx opened on Feb 23, 2020 — Member

In the light of historic and current issues involving subtyping (e.g. #1579 #2170 #2440 #3986 #4175 #4390 #4401) I am starting to wonder whether it is a good idea to have subtyping enabled by default in Agda. All dependent type theories with subtyping that are know either use coercive subtyping or restrict it to a very specific setting (i.e. cumulativity). On the other hand, Agda now has a notion of material subtyping that is used for several features: irrelevance, erasure, sized types, cumulativity, and cohesion. In particular, it seems that we do not yet fully understand how constraint solving and metavariables in such a setting are supposed to work.

In this light, I would like to discuss whether it is a good idea to have (material) subtyping enabled by default. Maybe it would be better to have a general flag `--no-subtyping` that disables material subtyping across the board? Things like irrelevance and erasure should still function with this option, though it might be necessary to eta-expand some functions by hand. Sized types and cumulativity would obviously not be compatible with this flag.

What do you think? Is this a good idea or do we need a less radical solution?

👍 2

jespercockx added subtyping type: discussion on Feb 23, 2020

nad on Feb 23, 2020 — Contributor

I just asked you a similar question.

> In particular, it seems that we do not yet fully understand how constraint solving and metavariables in such a setting are supposed to work.

Assignees
No one assigned

Labels
subtyping type: discussion

Type
No type

Projects
No projects

Milestone
2.6.1
Closed on Mar 16, 2020, 100% complete

Relationships
None yet

Development
Code with agent mode
No branches or pull requests

Notifications Customize

Dependent types in one slide

Types can contain terms: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{Vect}\, A\, n}$

Dependent types in one slide

Types can contain terms: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{Vect}\, A\, n}$

A change with **many** consequences...

Dependent types in one slide

Types can contain terms: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{Vect}\, A\, n}$

A change with **many** consequences...

Type well-formation $\Gamma \vdash A$

Substitution during typing $\dfrac{\Gamma \vdash f : \Pi\, x{:}\, A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash fu : B[u/x]}$

Dependent types in one slide

Types can contain terms: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{Vect}\, A\, n}$

A change with **many** consequences...

Type well-formation $\Gamma \vdash A$

Substitution during typing $\dfrac{\Gamma \vdash f : \Pi\, x{:}\, A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash fu : B[u/x]}$

Conversion/definitional equality $\dfrac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}$

Equivalence, congruent, and contains (at least) β rules $\dfrac{\Gamma, x{:}\, A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda\, x{:}\, A.t)\, u \equiv t[u/x] : B[u/x]}$

Dependent types in one slide

Types can contain terms: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{Vect}\, A\, n}$ A change with **many** consequences...

Type well-formation $\Gamma \vdash A$ Substitution during typing $\dfrac{\Gamma \vdash f : \Pi\, x{:}\, A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash fu : B[u/x]}$

Conversion/definitional equality $\dfrac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}$

Equivalence, congruent, and contains (at least) β rules $\dfrac{\Gamma, x{:}\, A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda\, x{:}\, A.t)\, u \equiv t[u/x] : B[u/x]}$

Typing depends on the equational theory of the language!

Warming-up with subtyping

w/ T. Laurent, K. Maillard @ ESOP '24

Implicit subtyping

What users™ want:

$$\textsc{Sub}\ \frac{\Gamma \vdash_{\mathsf{sub}} t : A \qquad \Gamma \vdash_{\mathsf{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\mathsf{sub}} t : A'}$$

Implicit subtyping

What users™ want:

$$\textsc{Sub}\ \frac{\Gamma \vdash_{\mathsf{sub}} t : A \qquad \Gamma \vdash_{\mathsf{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\mathsf{sub}} t : A'}$$

Think of $\preccurlyeq$ as (set) inclusion.

$$\begin{array}{ccc} t & = & t \\ \cap & & \cap \\ \mathsf{Tm}(A) & \subseteq & \mathsf{Tm}(A') \end{array}$$

Implicit subtyping

What users™ want:

$$\textsc{Sub}\ \frac{\Gamma \vdash_{\mathsf{sub}} t : A \qquad \Gamma \vdash_{\mathsf{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\mathsf{sub}} t : A'}$$

Think of $\preccurlyeq$ as (set) inclusion.

$$\begin{array}{ccc} t & = & t \\ \cap & & \cap \\ \mathsf{Tm}(A) & \subseteq & \mathsf{Tm}(A') \end{array}$$

Type theorists hate this:

- limited model: $A' \subseteq A \bigwedge B \subseteq B' \nRightarrow A \to B \subseteq A' \to B'$
- difficult meta-theory
- annoying to implement

Explicit subtyping

What type theorists want you to do:

$$\textsc{Coe}\ \frac{\Gamma \vdash_{\text{coe}} t : A \qquad \Gamma \vdash_{\text{coe}} A \preccurlyeq A'}{\Gamma \vdash_{\text{coe}} \mathrm{coe}_{A,A'}\, t : A'}$$

Explicit coe is much easier to model/study/implement!

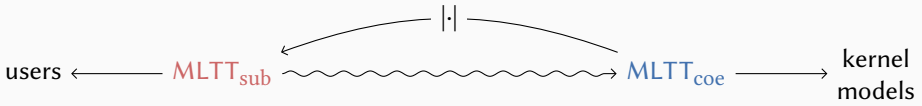
What type theorists want you to do:

$$\textsc{Coe}\ \frac{\Gamma \vdash_{\mathrm{coe}} t : A \qquad \Gamma \vdash_{\mathrm{coe}} A \preccurlyeq A'}{\Gamma \vdash_{\mathrm{coe}} \mathrm{coe}_{A,A'}\, t : A'}$$

Explicit coe is much easier to model/study/implement!

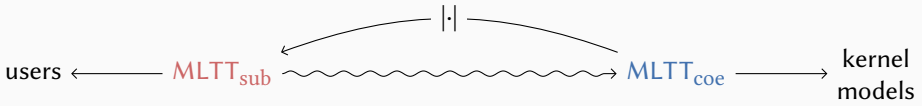
This is the work of a compiler!

Let's compile, then!



$$\textsc{Sub}\ \frac{\Gamma \vdash_{\text{sub}} t : A \qquad \Gamma \vdash_{\text{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\text{sub}} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{\text{coe}} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{\text{coe}} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{\text{coe}} \text{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

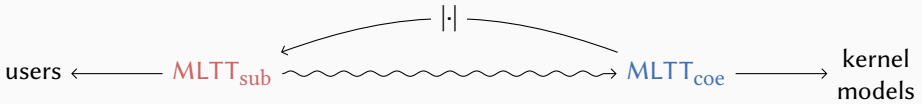
Let's compile, then!



$$\textsc{Sub}\ \frac{\Gamma \vdash_{\text{sub}} t : A \qquad \Gamma \vdash_{\text{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\text{sub}} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{\text{coe}} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{\text{coe}} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{\text{coe}} \text{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

```
Unable to unify "t" with "t".
```

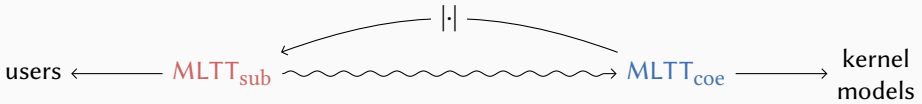
Let's compile, then!



$$\textsc{Sub}\ \frac{\Gamma \vdash_{\text{sub}} t : A \qquad \Gamma \vdash_{\text{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\text{sub}} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{\text{coe}} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{\text{coe}} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{\text{coe}} \text{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

```
Unable to unify "t" with "t".
Set Printing Coercions.
Unable to unify "t'" with "t'".
```

Let's compile, then!

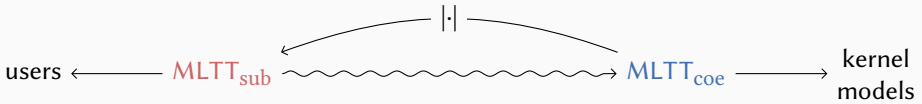


$$\textsc{Sub}\ \frac{\Gamma \vdash_{\text{sub}} t : A \qquad \Gamma \vdash_{\text{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\text{sub}} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{\text{coe}} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{\text{coe}} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{\text{coe}} \text{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

```
Unable to unify "t" with "t".
Set Printing Coercions.
Unable to unify "t'" with "t''".
```

Compilation should be **unambiguous**.
Necessary for compilation to **preserve typing**.

Let's compile, then!



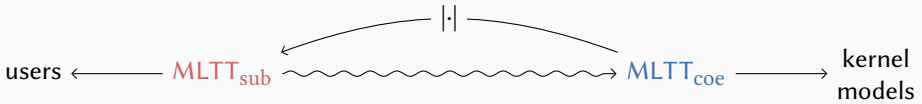
$$\textsc{Sub}\ \frac{\Gamma \vdash_{\text{sub}} t : A \qquad \Gamma \vdash_{\text{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\text{sub}} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{\text{coe}} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{\text{coe}} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{\text{coe}} \text{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

```
Unable to unify "t" with "t".
Set Printing Coercions.
Unable to unify "t'" with "t'".
```

Compilation should be unambiguous.
Necessary for compilation to preserve typing.

Coherence: If $|t| = |u|$ then $t \equiv u$.

Let's compile, then!



$$\textsc{Sub}\ \frac{\Gamma \vdash_{sub} t : A \qquad \Gamma \vdash_{sub} A \preccurlyeq A'}{\Gamma \vdash_{sub} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{coe} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{coe} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{coe} \mathrm{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

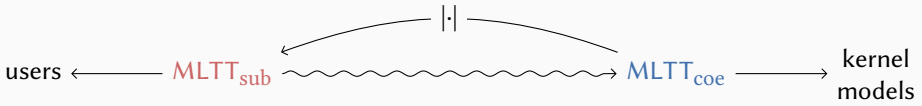
```
Unable to unify "t" with "t".
Set Printing Coercions.
Unable to unify "t'" with "t''".
```

Compilation should be unambiguous.
Necessary for compilation to preserve typing.

Coherence: If $|t| = |u|$ then $t \equiv u$.

What equations do we need?

Let's compile, then!



$$\textsc{Sub}\ \frac{\Gamma \vdash_{sub} t : A \qquad \Gamma \vdash_{sub} A \preccurlyeq A'}{\Gamma \vdash_{sub} t : A'} \quad \rightsquigarrow \quad \textsc{Coe}\ \frac{\tilde{\Gamma} \vdash_{coe} \tilde{t} : \tilde{A} \qquad \tilde{\Gamma} \vdash_{coe} \tilde{A} \preccurlyeq \tilde{A}'}{\tilde{\Gamma} \vdash_{coe} \mathrm{coe}_{\tilde{A},\tilde{A}'}\, \tilde{t} : \tilde{A}'}$$

```
Unable to unify "t" with "t".
Set Printing Coercions.
Unable to unify "t'" with "t'".
```

Compilation should be unambiguous.
Necessary for compilation to preserve typing.

Coherence: If $|t| = |u|$ then $t \equiv u$.

A **type-theoretic** question

What equations do we need?

New equations for structural subtyping

Structural subtyping

Focus = **structural** subtyping:

$$\frac{A' \preccurlyeq A \qquad B \preccurlyeq B'}{A \to B \preccurlyeq A' \to B'} \qquad \frac{A \preccurlyeq A'}{\mathrm{List}\, A \preccurlyeq \mathrm{List}\, A'} \qquad \frac{A \preccurlyeq A' \qquad B \preccurlyeq B'}{A \times B \preccurlyeq A' \times B'} \qquad \ldots$$

What equations do we need?

What equations do we need?

Computation equations:

$$\begin{aligned}
\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\,[] &\equiv [] \\
\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\,(a :: l) &\equiv (\mathrm{coe}_{A,A'}\ a) :: \mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\ l \\
(\mathrm{coe}_{A\to B,A'\to B'}\ f)\ u &\equiv \mathrm{coe}_{B,B'}(f\ (\mathrm{coe}_{A',A}\ u)) \\
&\vdots
\end{aligned}$$

What equations do we need?

Computation equations:

$$\begin{aligned}
\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\,[] &\equiv [] \\
\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\,(a :: l) &\equiv (\mathrm{coe}_{A,A'}\ a) :: \mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\ l \\
(\mathrm{coe}_{A\to B,A'\to B'}\ f)\ u &\equiv \mathrm{coe}_{B,B'}(f\ (\mathrm{coe}_{A',A}\ u)) \\
&\vdots
\end{aligned}$$

Functoriality equations:

$$\begin{aligned}
\mathrm{coe}_{A',A''}\ \mathrm{coe}_{A,A'}\ l &\equiv \mathrm{coe}_{A,A''}\ l \\
\mathrm{coe}_{A,A}\ l &\equiv l \\
&\vdots
\end{aligned}$$

New equations for neutrals

$$\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\ l := \mathrm{map}_{\mathrm{List}}\ \mathrm{coe}_{A,A'}\ l \qquad ?$$

New equations for neutrals

$$\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\ l := \mathrm{map}_{\mathrm{List}}\ \mathrm{coe}_{A,A'}\ l \qquad ?$$

Not in vanilla CIC/MLTT, where

$$\begin{array}{rcl} \mathrm{map}_{\mathrm{List}}\ f\ (\mathrm{map}_{\mathrm{List}}\ g\ x) & \not\equiv & \mathrm{map}_{\mathrm{List}}(f \circ g)\ x \\ \mathrm{map}_{\mathrm{List}}\ \mathrm{id}\ x & \not\equiv & x \end{array}$$

New equations for neutrals

$$\mathrm{coe}_{\mathrm{List}\ A,\mathrm{List}\ A'}\ l := \mathrm{map}_{\mathrm{List}}\ \mathrm{coe}_{A,A'}\ l \qquad ?$$

Not in vanilla CIC/MLTT, where

$$\begin{array}{rcl} \mathrm{map}_{\mathrm{List}}\ f\ (\mathrm{map}_{\mathrm{List}}\ g\ x) & \not\equiv & \mathrm{map}_{\mathrm{List}}(f \circ g)\ x \\ \mathrm{map}_{\mathrm{List}}\ \mathrm{id}\ x & \not\equiv & x \end{array}$$

Can we add these functoriality equations in?

DRAFT

New Equations for Neutral Terms

A Sound and Complete Decision Procedure, Formalized

Guillaume Allais Conor McBride
University of Strathclyde
{guillaume.allais, conor.mcbride}@strath.ac.uk

Pierre Boutillier
PPS - Paris Diderot
pierre.boutillier@pps.univ-paris-diderot.fr

Abstract

The definitional equality of an intensional type theory is its test of type compatibility. Today's systems rely on ordinary evaluation semantics to compare expressions in types, frustrating users with type errors arising when evaluation fails to identify two 'obviously' equal terms. If only the machine could decide a richer theory! We propose a way to decide theories which supplement evaluation with 'ν-rules', rearranging the neutral parts of normal forms, and report a successful initial experiment.

We study a simple λ-calculus with primitive fold, map and append operations on lists and develop in Agda a sound and complete decision procedure for an equational theory enriched with monoid, functor and fusion laws.

Keywords Normalization by Evaluation, Logical Relations, Simply-Typed Lambda Calculus, Map Fusion

1. Introduction

The programmer working in intensional type theory is no stranger to 'obviously true' equations she wishes held *definitionally* for her program to typecheck without having to chase down ill-typed terms and brutally coerce them. In this article, we present one way to relax definitional equality, thus accommodating some of her longings. We distinguish three types of fundamental relations between terms.

The first denotes computational rules: it is untyped, *oriented* and denoted by $\rightsquigarrow$ in its one step version or $\rightsquigarrow^*$ when the reflexive transitive congruence closure is considered. In Table 1, we introduce a few such rules which correspond to the equations the programmer writes to define functions. They are referred to as δ (for *definitions*) and ι (for pattern-matching on *inductive* data) rules and hold computationally just like the more common β-rule.

The second is the judgmental equality ($\equiv$): it is typed, tractable

```
map : (a → b) → list a → list b
map f []        ⇝ []
map f (x :: xs) ⇝ f x :: map f xs

(++) : list a → list a → list a
[]       ++ ys ⇝ ys
x :: xs ++ ys ⇝ x :: (xs ++ ys)

fold : (a → b → b) → b → list a → b
fold c n []        ⇝ n
fold c n (x :: xs) ⇝ c x (fold c n xs)
```

Table 1. $\delta\iota$-rules - computational

$$\begin{array}{lll} \Gamma & \vdash \mathtt{f} \equiv \lambda\ \mathtt{x}\ \mathtt{.}\ \mathtt{f}\ \mathtt{x} & : \mathtt{a} \to \mathtt{b} \\ \Gamma & \vdash \mathtt{p} \equiv (\pi_1\ \mathtt{p}\ ,\ \pi_2\ \mathtt{p}) & : \mathtt{a} * \mathtt{b} \\ \Gamma & \vdash \mathtt{u} \equiv () & : \mathtt{1} \end{array}$$

Table 2. η-rules - canonicity

fied judgmentally. Table 3 shows a kit for building computationally inert *neutral* terms growing layers of thwarted progress around a variable which we dub the 'nut', together with some equations on neutral terms which held only propositionally – until now. This paper shows how to extend the judgmental equality with these new 'ν-rules'. We gain, for example, that `map swap . map swap` $\equiv$ `id`, where `swap` swaps the elements of a pair.

New I

A Sound and

Guillaume Allais Con
University of Strathc
{guillaume.allais, conor.mcbride

Abstract

The definitional equality of an intensional t
of type compatibility. Today's systems rely o
semantics to compare expressions in types,
type errors arising when evaluation fails to ide
equal terms. If only the machine could decid
propose a way to decide theories which supple
'ι-rules', rearranging the neutral parts of nor
a successful initial experiment.

We study a simple λ-calculus with primit
pend operations on lists and develop in Agda
decision procedure for an equational theory e
functor and fusion laws.

Keywords Normalization by Evaluation, Lo
Typed Lambda Calculus, Map Fusion

1. Introduction

The programmer working in intensional type
to 'obviously true' equations she wishes held
program to typecheck without having to chase
and brutally coerce them. In this article, we pr
definitional equality, thus accommodating s
We distinguish three types of fundamental rel

The first denotes computational rules: it is
denoted by $\rightsquigarrow$ in its one step version or $\rightsquigarrow^*$ w
sitive congruence closure is considered. In Ta
few such rules which correspond to the equat
writes to define functions. They are referred t
and ι (for pattern-matching on *inductive* data
putationally just like the more common β-rul

The second is the judgmental equality ($=$

Decidability of Conversion for Type Theory in Type Theory

ANDREAS ABEL, Gothenburg University, Sweden
JOAKIM ÖHMAN, IMDEA Software Institute, Spain
ANDREA VEZZOSI, Chalmers University of Technology, Sweden

Type theory should be able to handle its own meta-theory, both to justify its foundational claims and to obtain a verified implementation. At the core of a type checker for intensional type theory lies an algorithm to check equality of types, or in other words, to check whether two types are convertible. We have formalized in Agda a practical conversion checking algorithm for a dependent type theory with one universe à la Russell, natural numbers, and η-equality for Π types. We prove the algorithm correct via a Kripke logical relation parameterized by a suitable notion of equivalence of terms. We then instantiate the parameterized fundamental lemma twice: once to obtain canonicity and injectivity of type formers, and once again to prove the completeness of the algorithm. Our proof relies on inductive-recursive definitions, but not on the uniqueness of identity proofs. Thus, it is valid in variants of intensional Martin-Löf Type Theory as long as they support induction-recursion, for instance, Extensional, Observational, or Homotopy Type Theory.

CCS Concepts: • **Theory of computation → Type theory**; *Proof theory*;

Additional Key Words and Phrases: Dependent types, Logical relations, Formalization, Agda

ACM Reference Format:
Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (January 2018), 29 pages. https://doi.org/10.1145/3158111

1 INTRODUCTION

DRAFT

New I

A Sound and

Guillaume Allais Con

Martin-Löf *à la* Coq

Arthur Adjedj
ENS Paris Saclay, Université Paris-Saclay
Gif-sur-Yvette, France

Meven Lennon-Bertrand
University of Cambridge
Cambridge, United Kingdom

Kenji Maillard
Inria
Nantes, France

Pierre-Marie Pédrot
Inria
Nantes, France

Loïc Pujet
University of Stockholm
Stockholm, Sweden

Abstract

We present an extensive mechanization of the metatheory of Martin-Löf Type Theory (MLTT) in the Coq proof assistant. Our development builds on pre-existing work in Agda to show not only the decidability of conversion, but also the decidability of type checking, using an approach guided by bidirectional type checking. From our proof of decidability, we obtain a certified and executable type checker for a full-fledged version of MLTT with support for Π, Σ, ℕ, and **Id** types, and one universe. Our development does not rely on impredicativity, induction-recursion or any axiom beyond MLTT extended with indexed inductive types and a handful of predicative universes, thus narrowing the gap between the object theory and the metatheory to a mere difference in universes. Furthermore, our formalization choices are geared towards a modular development that relies on Coq's features, *e.g.* universe polymorphism and metaprogramming with tactics.

Keywords: Dependent type system, Bidirectional typing, Log-

checker is spent on establishing meta-theoretic properties, which are necessary to ensure termination of the type checker but have little to do with its concrete implementation.

Acknowledging this tension leads to two radically different approaches. On the one hand, one can simply postulate normalization, to better concentrate on the difficulties faced when certifying a realistic type-checker. The most ambitious project to date that follows this approach is Meta-Coq [Sozeau, Anand, et al. 2020; Sozeau, Forster, et al. 2023], which formalizes a nearly complete fragment of Coq's type system and provides a certified type checker aiming for execution in a realistic context, after extraction. On the other hand, one can concentrate on normalization and decidability of conversion, which are the most difficult theoretical problems. The most advanced formalizations on that end are Abel, Öhman, et al. [2017] and Wieczorek and Biernacki [2018]. The first, in Agda, shows decidability of conversion, but does not provide an executable conversion checker. The second, in Coq, certifies a conversion checker designed for execution after extraction, but supports a type theory that is

Artifacts Evaluated – Functional

ype Theory

al claims and to obtain
an algorithm to check
ve formalized in Agda
se à la Russell, natural
relation parameterized
damental lemma twice:
e completeness of the
ness of identity proofs.
rt induction-recursion,

Agda

Type Theory in Type
oi.org/10.1145/3158111

Theorems!

$\mathrm{MLTT}_{\mathrm{coe}}$ is a nice type theory

We can design $\mathrm{MLTT}_{\mathrm{coe}}$ with all these equations and good type theoretic properties:

- logical consistency
- decidability of conversion and type-checking

Featuring List (⚙), Π, Σ, W, +, =... (🖋)

Theorems!

$\mathrm{MLTT}_{\mathrm{coe}}$ is a nice type theory

We can design $\mathrm{MLTT}_{\mathrm{coe}}$ with all these equations and good type theoretic properties:

- logical consistency
- decidability of conversion and type-checking

Featuring List (⚙), Π, Σ, W, +, =... (🖋)

And it is a good target

There is an elaboration $\mathrm{MLTT}_{\mathrm{sub}} \rightsquigarrow \mathrm{MLTT}_{\mathrm{coe}}$ which preserves conversion and typing.
Moreover, it is an inverse of erasure, and elaboration is thus coherent.

"To compile structural implicit subtyping, you need **exactly** functoriality equations."

Theorems!

$\mathrm{MLTT}_{\mathrm{coe}}$ is a nice type theory

We can design $\mathrm{MLTT}_{\mathrm{coe}}$ with all these equations and good type theoretic properties:

- logical consistency
- decidability of conversion and type-checking

Featuring List (⚙), Π, Σ, W, +, =... (🖋)

And it is a good target

There is an elaboration $\mathrm{MLTT}_{\mathrm{sub}} \rightsquigarrow \mathrm{MLTT}_{\mathrm{coe}}$ which preserves conversion and typing.
Moreover, it is an inverse of erasure, and elaboration is thus coherent.

"To compile structural implicit subtyping, you need **exactly** functoriality equations."

But missing a **general framework**.

Structural casts and functorial types

w/ A. Adjedj, T. Benjamin & K. Maillard @ POPL '26

(Structural) casts everywhere

Explicit subtyping:
cast along subtyping derivations.

$$\frac{\Gamma \vdash A' \preccurlyeq A \quad \Gamma \vdash B \preccurlyeq B'}{\Gamma \vdash A \to B \preccurlyeq A' \to B' \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'}$$
$$\Gamma \vdash (\mathrm{coe}_{A \to B, A' \to B'}\, f)\, a' \equiv \mathrm{coe}_{B,B'}(f\ \mathrm{coe}_{A',A}\, a) : B'$$

(Structural) casts everywhere

Explicit subtyping:
cast along subtyping derivations.

$$\frac{\dfrac{\Gamma \vdash A' \preccurlyeq A \quad \Gamma \vdash B \preccurlyeq B'}{\Gamma \vdash A \to B \preccurlyeq A' \to B'} \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'}{\Gamma \vdash (\mathrm{coe}_{A \to B, A' \to B'}\, f)\, a' \equiv \mathrm{coe}_{B,B'}(f\ \mathrm{coe}_{A',A}\, a) : B'}$$

Observational equality:
cast along equality proofs.

$$\frac{\dfrac{\Gamma \vdash e_A : A' = A \quad \Gamma \vdash e_B : B = B'}{\Gamma \vdash e' := \ldots : A \to B = A' \to B'} \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'}{\Gamma \vdash \mathrm{transp}_{A \to B, A' \to B'}(e', f)\, a' \equiv \mathrm{transp}_{B,B'}(e_B, f\ \mathrm{transp}_{A',A}(e_A, a')) : B'}$$

Dynamic typing:
cast always allowed (might fail).

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a' : A'}{\Gamma \vdash (\langle A' \to B' \Leftarrow A \to B \rangle\, f)\, a' \equiv \langle B' \Leftarrow B \rangle (f\, \langle A \Leftarrow A' \rangle\, a') : B'}$$

(Structural) casts everywhere

Explicit subtyping:
cast along subtyping derivations.

$$\frac{\Gamma \vdash A' \preccurlyeq A \quad \Gamma \vdash B \preccurlyeq B'}{\Gamma \vdash A \to B \preccurlyeq A' \to B' \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'} \\ \Gamma \vdash (\mathrm{coe}_{A \to B, A' \to B'}\, f)\, a' \equiv \mathrm{coe}_{B,B'}(f\ \mathrm{coe}_{A',A}\, a) : B'$$

Observational equality:
cast along equality proofs.

$$\frac{\Gamma \vdash e_A : A' = A \quad \Gamma \vdash e_B : B = B'}{\Gamma \vdash e' := \ldots : A \to B = A' \to B' \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'} \\ \Gamma \vdash \mathrm{transp}_{A \to B, A' \to B'}(e', f)\, a' \equiv \mathrm{transp}_{B,B'}(e_B, f\ \mathrm{transp}_{A',A}(e_A, a')) : B'$$

Dynamic typing:
cast always allowed (might fail).

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a' : A'}{\Gamma \vdash (\langle A' \to B' \Leftarrow A \to B \rangle\, f)\, a' \equiv \langle B' \Leftarrow B \rangle (f\, \langle A \Leftarrow A' \rangle\, a') : B'}$$

There's a general pattern!

(Structural) casts everywhere

Explicit subtyping:
cast along subtyping derivations.

$$\frac{\Gamma \vdash A' \preccurlyeq A \quad \Gamma \vdash B \preccurlyeq B'}{\Gamma \vdash A \to B \preccurlyeq A' \to B' \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'}{\Gamma \vdash (\mathrm{coe}_{A \to B, A' \to B'}\, f)\, a' \equiv \mathrm{coe}_{B,B'}(f\ \mathrm{coe}_{A',A}\, a) : B'}$$

Observational equality:
cast along equality proofs.

$$\frac{\Gamma \vdash e_A : A' = A \quad \Gamma \vdash e_B : B = B'}{\Gamma \vdash e' := \ldots : A \to B = A' \to B' \qquad \Gamma \vdash f : A \to B \quad \Gamma \vdash a' : A'}{\Gamma \vdash \mathrm{transp}_{A \to B, A' \to B'}(e', f)\, a' \equiv \mathrm{transp}_{B,B'}(e_B, f\ \mathrm{transp}_{A',A}(e_A, a')) : B'}$$

Dynamic typing:
cast always allowed (might fail).

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a' : A'}{\Gamma \vdash (\langle A' \to B' \Leftarrow A \to B \rangle\, f)\, a' \equiv \langle B' \Leftarrow B \rangle (f\, \langle A \Leftarrow A' \rangle\, a') : B'}$$

There's a general pattern! **Functoriality?**

Understanding functorial type formers

Functor = a mapping between two categories:

- acts on objects and **arrows**
- preserves identities and composition

Understanding functorial type formers

Functor = a mapping between two categories:

- acts on objects and arrows
- preserves identities and composition

1. Make types into a category
2. Describe the source of type formers
3. Show functoriality for our favourite type formers
4. Profit!

Functor = a mapping between two categories:

- acts on objects and arrows
- preserves identities and composition

1. **Make types into a category**
2. Describe the source of type formers
3. Show functoriality for our favourite type formers
4. Profit!

The category of types and adapters

Adapters = "the data along which you can cast"

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a \rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A \rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a \rangle \equiv t\langle a \rangle\langle a' \rangle : A''}$$

The category of types and adapters

Adapters = "the data along which you can cast"

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a \rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A \rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a \rangle \equiv t\langle a \rangle\langle a' \rangle : A''}$$

A family of type theories:

- **Subtyping**: $A \Rightarrow B$ corresponds to $A \preccurlyeq B$. **Uniqueness**!

The category of types and adapters

Adapters = **"the data along which you can cast"**

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a \rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A \rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a \rangle \equiv t\langle a \rangle\langle a' \rangle : A''}$$

A family of type theories:

- Subtyping: $A \Rightarrow B$ corresponds to $A \preccurlyeq B$. Uniqueness!
- **Observational equality**: given $e : A = B$ we get $\underline{e} : A \Rightarrow B$
- **Dynamic typing**: $A \Rightarrow B$ always inhabited

Adapters = "the data along which you can cast"

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a\rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A\rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a\rangle \equiv t\langle a\rangle\langle a'\rangle : A''}$$

A family of type theories:

- Subtyping: $A \Rightarrow B$ corresponds to $A \preccurlyeq B$. Uniqueness!
- Observational equality: given $e : A = B$ we get $\underline{e} : A \Rightarrow B$
- Dynamic typing: $A \Rightarrow B$ always inhabited
- **Full function space**: given any $f : A \to B$ we get $\underline{f} : A \Rightarrow B$ (and $t\langle f\rangle \equiv f\ t$)

The category of types and adapters

Adapters = "the data along which you can cast"

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a \rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A \rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a \rangle \equiv t\langle a \rangle \langle a' \rangle : A''}$$

A family of type theories:

- Subtyping: $A \Rightarrow B$ corresponds to $A \preccurlyeq B$. Uniqueness!
- Observational equality: given $e : A = B$ we get $\underline{e} : A \Rightarrow B$
- Dynamic typing: $A \Rightarrow B$ always inhabited

} Need non-uniqueness too

- Full function space: given any $f : A \to B$ we get $\underline{f} : A \Rightarrow B$ (and $t\langle f \rangle \equiv f\ t$)

The category of types and adapters

Adapters = "the data along which you can cast"

$$\frac{a : A \Rightarrow A' \qquad t : A}{t\langle a\rangle : A'} \qquad \frac{t : A}{t\langle \mathrm{id}_A\rangle \equiv t : A} \qquad \frac{a : A \Rightarrow A' \qquad a' : A' \Rightarrow A'' \qquad t : A}{t\langle a' \circ a\rangle \equiv t\langle a\rangle\langle a'\rangle : A''}$$

A family of type theories:

- Subtyping: $A \Rightarrow B$ corresponds to $A \preccurlyeq B$. Uniqueness!
- Observational equality: given $e : A = B$ we get $\underline{e} : A \Rightarrow B$
- Dynamic typing: $A \Rightarrow B$ always inhabited

} Need non-uniqueness too

- Full function space: given any $f : A \to B$ we get $\underline{f} : A \Rightarrow B$ (and $t\langle f\rangle \equiv f\ t$)

Abstractly:

- a category $\mathbf{Ty}_\Gamma$ of types and adapters
- $\mathrm{Tm}_\Gamma : \mathbf{Ty}_\Gamma \to \mathbf{Set}$ is a functor

(CwF-style reinvention of comprehension categories, see also Coraglia's and Najmaei's work.)

Understanding functorial type formers

Functor = a mapping between two categories:

- acts on objects and arrows
- preserves identities and composition

1. Make types into a category
2. **Describe the source of type formers**
3. Show functoriality for our favourite type formers
4. Profit!

What is in a type former

```
Inductive List (X : Type) : Type := …
Inductive W (X : Type) (Y : X → Type) : Type := …
```

A type former is specified by a **context**

What is in a type former

```
Inductive List (X : Type) : Type := …
Inductive W (X : Type) (Y : X → Type) : Type := …
```

A type former is specified by a **context** , with

- type variables: $\Gamma_{\text{List}} := X : \text{Ty} \qquad \Gamma_{\to} := (X : \text{Ty}), (Y : \text{Ty})$

What is in a type former

```
Inductive List (X : Type) : Type := …
Inductive W (X : Type) (Y : X → Type) : Type := …
```

A type former is specified by a **context** , with

- type variables: $\Gamma_{\text{List}} := X : \text{Ty} \qquad \Gamma_{\to} := (X : \text{Ty}), (Y : \text{Ty})$
- *dependent* type variables: $\Gamma_{\text{W}} := (X : \text{Ty}), (Y : X.\ \text{Ty})$

What is in a type former

```
Inductive List (X : Type) : Type := …
Inductive W (X : Type) (Y : X → Type) : Type := …
```

A type former is specified by a **context** , with

- type variables: $\Gamma_{\text{List}} := X : \text{Ty}$ $\quad\quad$ $\Gamma_{\to} := (X : \text{Ty}), (Y : \text{Ty})$
- *dependent* type variables: $\Gamma_{\text{W}} := (X : \text{Ty}), (Y : X.\,\text{Ty})$
- term variables: $\Gamma_{=} := (X : \text{Ty}), (x : X), (y : X)$

What is in a type former

```
Inductive List (X : Type) : Type := …
Inductive W (X : Type) (Y : X → Type) : Type := …
```

A type former is specified by a **context** , with

- type variables: $\Gamma_{\text{List}} := X : \text{Ty} \qquad \Gamma_{\to} := (X : \text{Ty}), (Y : \text{Ty})$
- *dependent* type variables: $\Gamma_{\text{W}} := (X : \text{Ty}), (Y : X.\,\text{Ty})$
- term variables: $\Gamma_{=} := (X : \text{Ty}), (x : X), (y : X)$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A, B) : \Gamma_{\to}}$$

Transformations: where the magic happens

But wait, we have a way to relate two substitutions $\Gamma \vdash \sigma, \tau : \Gamma_{\rightarrow}$: *Adapters!*

Transformations: where the magic happens

But wait, we have a way to relate two substitutions $\Gamma \vdash \sigma, \tau : \Gamma_{\to}$: *Adapters!*

We need **variance** information, though:

$$\Gamma_{\to} := (X : \mathsf{Ty}_{-})(Y : \mathsf{Ty}_{+}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash a : A' \Rightarrow A \qquad \Gamma \vdash b : B \Rightarrow B'}{\Gamma \vdash a \to b : A \to B \Rightarrow A' \to B'}$$

But wait, we have a way to relate two substitutions $\Gamma \vdash \sigma, \tau : \Gamma_{\to}$: *Adapters!*

We need **variance** information, though:

$$\Gamma_{\to} := (X : \mathsf{Ty}_{-})(Y : \mathsf{Ty}_{+}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash a : A' \Rightarrow A \qquad \Gamma \vdash b : B \Rightarrow B'}{\Gamma \vdash (a, b) : (A, B) \Rightarrow_{\Gamma_{\to}} (A', B')}$$

But wait, we have a way to relate two substitutions $\Gamma \vdash \sigma, \tau : \Gamma_{\to}$: *Adapters!*

We need **variance** information, though:

$$\Gamma_{\to} := (X : \mathsf{Ty}_{-})(Y : \mathsf{Ty}_{+}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash a : A' \Rightarrow A \qquad \Gamma \vdash b : B \Rightarrow B'}{\Gamma \vdash (a, b) : (A, B) \Rightarrow_{\Gamma_{\to}} (A', B')}$$

A very general rule:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash \sigma, \tau : \Gamma \qquad \Delta \vdash \mu : \sigma \Rightarrow_{\Gamma} \tau}{\Delta \vdash A[\![\mu]\!] : A[\sigma] \Rightarrow A[\tau]}$$

All types are functors

But wait, we have a way to relate two substitutions $\Gamma \vdash \sigma, \tau : \Gamma_{\to}$: *Adapters!*

We need **variance** information, though:

$$\Gamma_{\to} := (X : \mathsf{Ty}_{-})(Y : \mathsf{Ty}_{+}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash a : A' \Rightarrow A \qquad \Gamma \vdash b : B \Rightarrow B'}{\Gamma \vdash (a, b) : (A, B) \Rightarrow_{\Gamma_{\to}} (A', B')}$$

A very general rule:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash \sigma, \tau : \Gamma \qquad \Delta \vdash \mu : \sigma \Rightarrow_{\Gamma} \tau}{\Delta \vdash A[\![\mu]\!] : A[\sigma] \Rightarrow A[\tau]}$$

All types are functors... obtained compositionally from functoriality of each type former.

AdapTT, categorically

- A 2-category **Ctx** of contexts, substitutions and transformations
- A 2-functor $\mathsf{Ty} : \mathbf{Ctx} \to \mathbf{Cat}$ (maps Γ to the category $\mathbf{Ty}_\Gamma$, $\cdot[\![\cdot]\!]$ is the action on 2-arrows)
- A "dependent 2-functor" $\mathsf{Tm} : (\Gamma \colon \mathsf{Ctx}) \to (\mathsf{Ty}(\Gamma) \to \mathbf{Set})$;
- Local representability (type and term variables);
- a 2-functor $\cdot^{-} : \mathbf{Ctx}^{\mathrm{co}} \to \mathbf{Ctx}$ to interpret negative variance.

... **lots** of data (and equations) to unpack!

Understanding functorial type formers

Functor = a mapping between two categories:

- acts on objects and arrows
- preserves identities and composition

1. Make types into a category
2. Describe the source of type formers
3. **Show functoriality for our favourite type formers**
4. Profit!

Making a type former functorial

Type specification recipe

1. Type formation rule $A \to B$
2. Constructor (λ) and eliminator (app)
3. Computation rule for each constructor-destructor combination (β)
4. Extensionality rule (η) (optional)

Making a type former functorial

Type specification recipe (**adapted**)

1. **Give the type former's context** ($\Gamma_{\to}$), and **derive**
   - **1.1** the type formation rule $A \to B$
   - **1.2** **the adapter formation rule** $a \to b$
2. Constructor (λ) and eliminator (app)
3. Computation rule for each constructor-destructor combination (β)
4. Extensionality rule (η) (optional)
5. **Computation rule for the adapter**

$$(f\langle a \to b\rangle)\, u \equiv (f\ u\langle a\rangle)\langle b\rangle$$

Our favourite type formers

$$\Gamma_\Pi := (X : \mathsf{Ty}_-), (Y : X.\,\mathsf{Ty}_+) \qquad \Gamma_\Sigma := (X : \mathsf{Ty}_+), (Y : X.\,\mathsf{Ty}_+)$$

Our favourite type formers

$$\Gamma_\Pi := (X\colon \mathsf{Ty}_-), (Y\colon X.\,\mathsf{Ty}_+) \qquad \Gamma_\Sigma := (X\colon \mathsf{Ty}_+), (Y\colon X.\,\mathsf{Ty}_+)$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, x\colon A' \vdash b : B[x\langle a\rangle/x] \Rightarrow B'}{\Delta \vdash \Pi\, a.b : \Pi(x\colon A).B \Rightarrow \Pi(x\colon A').B'} \qquad \frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, x\colon A \vdash b : B \Rightarrow B'[x\langle a\rangle/x]}{\Delta \vdash \Sigma\, a.b : \Sigma(x\colon A).B \Rightarrow \Sigma(x\colon A').B'}$$

Our favourite type formers

$$\Gamma_\Pi := (X\colon \mathsf{Ty}_-), (Y\colon X.\,\mathsf{Ty}_+) \qquad \Gamma_\Sigma := (X\colon \mathsf{Ty}_+), (Y\colon X.\,\mathsf{Ty}_+)$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, x\colon A' \vdash b : B[x\langle a\rangle/x] \Rightarrow B'}{\Delta \vdash \Pi\, a.b : \Pi(x\colon A).B \Rightarrow \Pi(x\colon A').B'} \qquad \frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, x\colon A \vdash b : B \Rightarrow B'[x\langle a\rangle/x]}{\Delta \vdash \Sigma\, a.b : \Sigma(x\colon A).B \Rightarrow \Sigma(x\colon A').B'}$$

Our favourite type formers

$$\Gamma_\Pi := (X\colon \mathsf{Ty}_-), (Y\colon X.\,\mathsf{Ty}_+) \qquad \Gamma_\Sigma := (X\colon \mathsf{Ty}_+), (Y\colon X.\,\mathsf{Ty}_+)$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, x\colon A' \vdash b : B[x\langle a\rangle/x] \Rightarrow B'}{\Delta \vdash \Pi\, a.b : \Pi(x\colon A).B \Rightarrow \Pi(x\colon A').B'} \qquad \frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, x\colon A \vdash b : B \Rightarrow B'[x\langle a\rangle/x]}{\Delta \vdash \Sigma\, a.b : \Sigma(x\colon A).B \Rightarrow \Sigma(x\colon A').B'}$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, (x\colon A') \vdash b : B[x\langle a\rangle/x] \Rightarrow B' \qquad \Delta \vdash f : \Pi(x\colon A).B \qquad \Delta \vdash u : A'}{\Delta \vdash f\langle \Pi\, a.b\rangle\, u \equiv (f\ u\langle a\rangle)\langle b[u/x]\rangle : B'[u/x]}$$

$$\frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, (x\colon A) \vdash b : B \Rightarrow B'[x\langle a\rangle/x] \qquad \Delta \vdash p : \Sigma(x\colon A).B}{\Delta \vdash \pi_1\ (p\langle \Sigma\, a.b\rangle) \equiv (\pi_1\ p)\langle a\rangle : A' \qquad \Delta \vdash \pi_2\ (p\langle \Sigma\, a.b\rangle) \equiv (\pi_2\ p)\langle b[\pi_1 p/x]\rangle : B'[(\pi_1 p)\langle a\rangle/x]}$$

Our favourite type formers

$$\Gamma_\Pi := (X\colon \mathsf{Ty}_-), (Y\colon X.\,\mathsf{Ty}_+) \qquad \Gamma_\Sigma := (X\colon \mathsf{Ty}_+), (Y\colon X.\,\mathsf{Ty}_+)$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, x\colon A' \vdash b : B[x\langle a\rangle/x] \Rightarrow B'}{\Delta \vdash \Pi\, a.b : \Pi(x\colon A).B \Rightarrow \Pi(x\colon A').B'} \qquad \frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, x\colon A \vdash b : B \Rightarrow B'[x\langle a\rangle/x]}{\Delta \vdash \Sigma\, a.b : \Sigma(x\colon A).B \Rightarrow \Sigma(x\colon A').B'}$$

$$\frac{\Delta \vdash a : A' \Rightarrow A \qquad \Delta, (x\colon A') \vdash b : B[x\langle a\rangle/x] \Rightarrow B' \qquad \Delta \vdash f : \Pi(x\colon A).B \qquad \Delta \vdash u : A'}{\Delta \vdash f\langle \Pi\, a.b\rangle\, u \equiv (f\ u\langle a\rangle)\langle b[u/x]\rangle : B'[u/x]}$$

$$\frac{\Delta \vdash a : A \Rightarrow A' \qquad \Delta, (x\colon A) \vdash b : B \Rightarrow B'[x\langle a\rangle/x] \qquad \Delta \vdash p : \Sigma(x\colon A).B}{\Delta \vdash \pi_1\,(p\langle \Sigma\, a.b\rangle) \equiv (\pi_1\ p)\langle a\rangle : A' \qquad \Delta \vdash \pi_2\,(p\langle \Sigma\, a.b\rangle) \equiv (\pi_2\ p)\langle b[\pi_1 p/x]\rangle : B'[(\pi_1 p)\langle a\rangle/x]}$$

A bit intense... but **clear guidelines**.

Understanding functorial type formers

Functor = a mapping between two categories:

- acts on objects and arrows
- preserves identities and composition

1. Make types into a category
2. Describe the source of type formers
3. Show functoriality for our favourite type formers
4. **Profit!**

Functorial inductive types

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons (a : A) (l : list A) : A
```

```
Inductive W (A : Type) (B : A -> Type) : Type :=
  | sup (a : A) (rec : B a -> W A B) : W A B
```

```
Inductive eq (A : Type) : A -> A -> Type :=
  | eq_refl (a : A) : eq A a a
```

Functorial inductive types

```
Inductive list (A : Type+) : Type+ :=
  | nil : list A
  | cons (a : A) (l : list A) : A
```

```
Inductive W (A : Type+) (B : A -> Type-) : Type :=
  | sup (a : A) (rec : B a -> W A B) : W A B
```

```
Inductive eq (A : Type) : A -> A -> Type :=
  | eq_refl (a : A) : eq A a a
```

Idea:

1. Include **variance** in the types' context
2. Derive the adapter's type from the context
3. **Derive the adapter's computation rule from the constructors' description**

Functorial inductive types

```
Inductive list (A : Type+) : Type+ :=
  | nil : list A
  | cons (a : A) (l : list A) : A
```

```
Inductive W (A : Type+) (B : A -> Type-) : Type :=
  | sup (a : A) (rec : B a -> W A B) : W A B
```

```
Inductive eq (A : Type) : A -> A -> Type :=
  | eq_refl (a : A) : eq A a a
```

Idea:

1. Include variance in the types' context
2. Derive the adapter's type from the context
3. Derive the adapter's computation rule from the constructors' description

All the hard work is done!

Wrapping up

What's cooking

- Meta-theory of AdapTT
- Alternative presentation of type variables?
- Experimental implementation
- Explore instances of the framework (cumulativity, subset types, records…)
- More category theory

Structural casts $\leftrightarrow$ Functorial type formers

Structural casts ↔ Functorial type formers

We can design **better coercions**...

Structural casts ↔ Functorial type formers

We can design **better coercions**...

... but we have to push conversion **beyond mere computation**!

Structural casts ↔ Functorial type formers

We can design better coercions...

... but we have to push conversion beyond mere computation!

THANKS!