AdapTT: Functoriality for Dependent Type Casts

POPL 2026

Arthur Adjedj, **Meven Lennon-Bertrand**, Thibaut Benjamin & Kenji Maillard

Arthur Adjedj

Thibaut Benjamin

Kenji Maillard

Agda

LEAN

ROCQ

Idris

Many forms of **type casting**

Many forms of **type casting**

“It wouldn’t work without a nice notation system.”

– Damien Pous, 2 days ago

Many forms of **type casting** and they're quite ~~broken~~ **complicated**.

"It wouldn't work without a nice notation system."
– Damien Pous, 2 days ago

Parametrized coercions #2455

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

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

Polarities: subtyping for datatypes #65

Closed

New issue

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

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

What is type casting, anyway?

Adapters= **“the data along which you can cast”**

Adapters= **“the data along which you can cast”**

For the type theorists:

$$\textsc{Cast}\ \frac{a : A \Rightarrow A' \qquad t : A}{t\langle a \rangle : A'}$$

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

For the type theorists:

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

For the category theorists: like natural models/CwF, but with **Cat**-valued presheaves

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

A reinvention of (split) comprehension categories: see Coraglia, **Najmaei**.

Many examples

A whole family of type theories:

- **Subtyping**:
  $A \Rightarrow B$ means "A is a subtype of B" $\longrightarrow$ **Uniqueness**!

A whole family of type theories:

- **Subtyping**:
  $A \Rightarrow B$ means "A is a subtype of B" $\longrightarrow$ Uniqueness!

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

Many examples

A whole family of type theories:

- **Subtyping**:
  $A \Rightarrow B$ means "A is a subtype of B" $\longrightarrow$ Uniqueness!
- **Observational equality**
- **Cast calculi for gradual typing**

  } Non-unique, non-full

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

Many examples

A whole family of type theories:

- **Subtyping**:
  $A \Rightarrow B$ means "A is a subtype of B" $\longrightarrow$ Uniqueness!
- **Observational equality**
- **Cast calculi for gradual typing**

  (Observational equality and cast calculi: Non-unique, non-full)
- …
- **Full function space**:
  given any $f : A \to B$ we get $\underline{f} : A \Rightarrow B$ (and $t\langle f \rangle \equiv f\ t$)

Structural casts and functorial types

Coercive subtyping:
cast along subtyping derivations

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

Structural casts everywhere

Coercive subtyping:
cast along subtyping derivations

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

Observational equality:
cast along equality proofs

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

Gradual typing:
casts always allowed, can fail

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

Coercive sub
cast along sub

$\Gamma \vdash a' : A'$

$_{,A}\, a) : B'$

Observationa
cast along equa

$\Gamma \vdash a' : A'$

$(e_A, a')) : B'$

Gradual typi
casts always a

$A' \rangle\, a') : B'$

Coercive sub
cast along sub

$\Gamma \vdash a' : A'$

$_{,A}\, a) : B'$

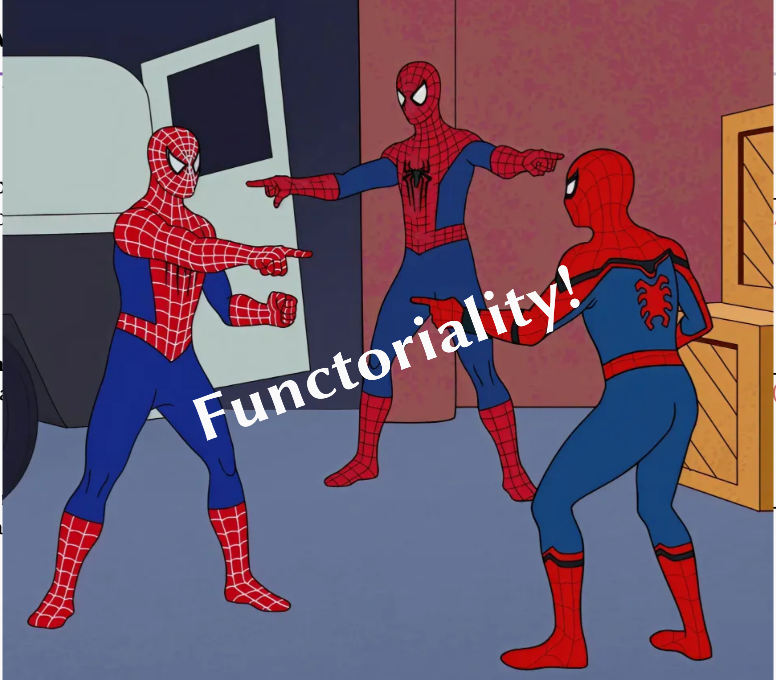
Observationa
cast along equ

$\Gamma \vdash a' : A'$

$(e_A, a')) : B'$

Gradual typi
casts always a

$A'\rangle\, a') : B'$



Functorial type formers

Type former F:

- acts on objects and **arrows**
- **preserves identities and composition** (critical! [ESOP24])

Functorial type formers

Type former F:

- acts on objects and arrows
- preserves identities and composition (critical! [ESOP24])

$$F_\Gamma : ?? \to \mathbf{Ty}_\Gamma$$

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

Functorial type formers

Type former F:

- acts on objects and arrows
- preserves identities and composition (critical! [ESOP24])

$$F_\Gamma : ?? \to \mathbf{Ty}_\Gamma$$

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

What is in a type former

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

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

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

- type variables: $\Gamma_{\mathrm{List}} := X : \mathrm{Ty} \qquad \Gamma_{\rightarrow} := (X : \mathrm{Ty}), (Y : \mathrm{Ty})$
- **dependent** type variables: $\Gamma_{\Pi}, \Gamma_{\Sigma}, \Gamma_{\mathrm{W}} := (X : \mathrm{Ty}), (Y : X.\,\mathrm{Ty})$

What is in a type former

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

- type variables: $\Gamma_{\mathrm{List}} := X : \mathrm{Ty}$ $\qquad \Gamma_{\to} := (X : \mathrm{Ty}), (Y : \mathrm{Ty})$
- **dependent** type variables: $\Gamma_{\Pi}, \Gamma_{\Sigma}, \Gamma_{\mathrm{W}} := (X : \mathrm{Ty}), (Y : X.\,\mathrm{Ty})$
- term variables: $\Gamma_{=} := (X : \mathrm{Ty}), (x : X), (y : X)$ $\qquad \Gamma_{\mathrm{Vec}} := (X : \mathrm{Ty}), (n : \mathrm{N})$

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

- type variables: $\Gamma_{\mathrm{List}} := X\colon \mathsf{Ty}$ $\qquad$ $\Gamma_{\to} := (X\colon \mathsf{Ty}), (Y\colon \mathsf{Ty})$
- **dependent** type variables: $\Gamma_{\Pi}, \Gamma_{\Sigma}, \Gamma_{\mathrm{W}} := (X\colon \mathsf{Ty}), (Y\colon X.\,\mathsf{Ty})$
- term variables: $\Gamma_{=} := (X\colon \mathsf{Ty}), (x\colon X), (y\colon X)$ $\qquad$ $\Gamma_{\mathrm{Vec}} := (X\colon \mathsf{Ty}), (n\colon \mathsf{N})$

Yoneda magic:

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

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

Adapters!

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

Adapters!

We need **variance** information:

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

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:

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

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:

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

A very general rule for **transformations**:

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

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

Adapters!

We need variance information:

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

A very general rule for **transformations**:

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

All types are functors

Type former F:

- acts on objects and arrows
- preserves identities and composition (critical! [ESOP24])

$$F_\Gamma : \mathbf{Sub}(\Gamma, \Gamma_F) \to \mathbf{Ty}_\Gamma$$

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

Categorically

- A **2**-category $\mathbf{Ctx}$ of contexts, substitutions and transformations
- A **2**-functor $\mathrm{Ty} : \mathbf{Ctx} \to \mathbf{Cat}$
- A dependent **2**-functor $\mathrm{Tm} : (\Gamma : \mathbf{Ctx}) \to (\mathrm{Ty}(\Gamma) \to \mathbf{Set})$
- Local representability data (term and **type** variables)
- a **2**-functor $\cdot^{-} : \mathbf{Ctx}^{\mathrm{co}} \to \mathbf{Ctx}$

Lots of data and equations to unpack!

Categorically

- A 2-category **Ctx** of contexts, substitutions and transformations
- A 2-functor $\mathrm{Ty} : \mathbf{Ctx} \to \mathbf{Cat}$
- A dependent 2-functor $\mathrm{Tm} : (\Gamma : \mathbf{Ctx}) \to (\mathrm{Ty}(\Gamma) \to \mathbf{Set})$
- Local representability data (term and type variables)
- a 2-functor $\cdot^{-} : \mathbf{Ctx}^{\mathrm{co}} \to \mathbf{Ctx}$

Lots of data and equations to unpack!

Coming back: **models in presheaves** over a model of AdapTT

Our favourite type formers

Functorial type formers

Type former F:

- acts on objects and arrows
- preserves identities and composition (critical! [ESOP24])

$$F_\Gamma : \mathbf{Sub}(\Gamma, \Gamma_F) \to \mathbf{Ty}_\Gamma$$

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

Type specification recipe

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

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

Π, Σ: easy fit

$$(f\langle \Pi\, a.b \rangle)\ u \equiv (f\ \ u\langle a \rangle)\langle b[u\langle a \rangle] \rangle$$

Π, Σ: easy fit

$$(f\langle \Pi\, a.b \rangle)\ u \equiv (f\ \ u\langle a \rangle)\langle b[u\langle a \rangle] \rangle$$

Parameterised, indexed inductive types: a **theory of signatures** to generically derive everything

Π, Σ: easy fit

$$(f\langle \Pi\, a.b \rangle)\ u \equiv (f\ \ u\langle a \rangle)\langle b[u\langle a \rangle] \rangle$$

Parameterised, indexed inductive types: a theory of signatures to generically derive everything

All the hard work was already done!

Wrapping up

Type former F:

- acts on objects and arrows
- preserves identities and composition (critical! [ESOP24])

$$F_\Gamma : \mathbf{Sub}(\Gamma, \Gamma_F) \to \mathbf{Ty}_\Gamma$$

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

There's more cooking

- Meta-theory (normalisation)
- Alternative presentation of type variables
- Experimental implementation
- Instances of the framework (cumulativity, subset types, records...)
- Relation to binary parametricity?

Get in touch if you're interested!

Adapters: a foundation for dependent type-casting

Structural casts ↔ Functorial type formers

Thanks to cool 2-categorical structure

General indexed inductive types

Thanks!