An observational proof assistant for higher-dimensional mathematics

Michael Shulman
University of San Diego

j.w.w. Thorsten Altenkirch, Ambrus Kaposi,
Elif Uskuplu, and Chaitanya Leena Subramaniam

North American Annual Meeting
of the Association for Symbolic Logic
May 15, 2025

Narya

Narya is an experimental interval-free proof assistant
for higher-dimensional mathematics.

`https://github.com/gwaithimirdain/narya`

- Higher observational type theory
- Internally parametric type theory
- Displayed type theory

Outline

1. Dependent type theory
2. Higher observational type theory
3. Some technical details

Core language

The core language of Narya is a dependent type theory like those of Rocq, Agda, and Lean.

- Dependent function types `(x:A) → B x` with application `f a` and abstraction `x ↦ f x` (note syntax).
- A universe `Type` with `Type : Type`. No universe checking!
- No unification or implicit arguments yet.
- However, verbosity is decreased by bidirectional typechecking:
  - Un-annotated `x ↦ x` checks at any type `A → A`.
  - Un-annotated `f a` synthesizes `B a` if `f` synthesizes `(x:A) → B x` and `a` checks at `A`.
  - To write a redex, you must ascribe the function: `((x ↦ x) : A → A) a`
- NbE-based typechecker with η-conversion: `f` $\equiv$ `x ↦ f x`

Records

```
def Σ (A : Type) (B : A → Type) : Type := sig (
  fst : A,
  snd : B fst)
```

- An unlabeled tuple like `(a, b)` checks at any record with the right number of fields (of the right type).
- A labeled tuple like `(fst := a, snd := b)` checks at any record with fields having those names (and types).
- Fields are projected out with indices or labels, and spaces:

  `p .0` `p .fst`
- A projection synthesizes, if the head `p` synthesizes a record.
- Records satisfy η-conversion: `p` $\equiv$ `(p .fst, p .snd)`.

Parsing field projections

Field projections are left-associative like function applications:

`f (g a).fld b` means `((f (g a)) .fld) b`

This is the correct choice when application is juxtaposition.
It allows chained method calls without parentheses:

```
object .methodOne x y z
  .methodTwo a b
  .methodThree c d e
  .methodFour
```

Datatypes

```
def List (A : Type) : Type := data [
| nil.
| cons. (x : A) (xs : List A) ]
```

- A constructor like `cons. b bs` checks at any datatype with that constructor name, if its arguments check correctly.
- Constructors end with `.`, dually to how fields begin with `.`
- Numerals `0`, `1`, ... parse to `zero.`, `suc. zero.`, ...

- Matches with recursion, using an ML-like syntax:

```
def len (A : Type) (xs : List A) : ℕ := match xs [
| nil. ↦ 0
| cons. _ xs ↦ 1 + len xs ]
```

  but *with an ending delimiter* `]`.
- No termination or positivity checking!

Matches, variables, and contexts

You can match on any (synthesizing) term, not just a variable:

```
def last (A : Type) (xs : List A) : Maybe A
  := match reverse xs [
| nil. ↦ none.
| cons. x _ ↦ some. x ]
```

If you do match on a variable, the goal and the context refine:

```
def suc_pred (n : ℕ) (H : n ≠ 0) : suc. (pred n) = n
  := match n [
| zero. ↦ match H refl [ ]
| suc. k ↦ refl (suc. k) ]
```

- In the `zero.` branch, the type of `H` is `0 ≠ 0`.
- In the `suc.` branch, the goal reduces to `suc. k = suc. k`.

When matching on a non-variable, you can refine the goal explicitly with `match M return x ↦ P x` (and "convoy" the context).

Fancy datatypes

Datatypes can be indexed:

```
def Vec (A : Type) : ℕ → Type ≔ data [
| nil. : Vec A 0
| cons. (n : ℕ) (x : A) (xs : Vec A n) : Vec A (n+1) ]
```

Indexed matches refine the goal and context if all indices are distinct variables (more restrictive than Agda `--without-K`: no unification yet).

Families of definitions can be mutually inductive and/or recursive, including induction-induction and induction-recursion:

```
def ctx : Type ≔ data [
| empty.
| ext. (Γ : ctx) (A : ty Γ) ]

and ty (Γ : ctx) : Type ≔ data [
| base.
| pi. (A : ty Γ) (B : ty (ext. Γ A)) ]
```

Fancy matches

Matches can be deep:

```
def half (n : ℕ) : ℕ ≔ match n [
| zero. ↦ zero.
| suc. zero. ↦ zero.
| suc. (suc. n) ↦ suc. (half n) ]
```

Matches can be multiple:

```
def conj (x y : Bool) : Bool ≔ match x, y [
| true., true. ↦ true.
| true., false. ↦ false.
| false., _ ↦ false. ]
```

- Both expanded at parse time into sequential matches.
- The programmer controls and can easily see the case tree.
- Exact splits are required.

Codatatypes

Codatatypes are defined using a "self" variable:

```
def Stream (A : Type) : Type ≔ codata [
| s .head : A
| s .tail : Stream A ]
```

```
def M (A : Type) (B : A → Type) : Type ≔ codata [
| s .recv : A
| s .send : B (s .recv) → M A B ]
```

- Methods of a codatatype are called like fields of a record:

  `s .head` `s .tail b`

- Codatatypes are inhabited by copatterns and corecursion:

  ```
  def zeros : Stream ℕ ≔ [
  | .head ↦ 0
  | .tail ↦ zeros ]
  ```

- No productivity or positivity checking yet!

Canonical types in case trees

A novel feature: canonical types can be defined inside a match.

```
def Covec (A : Type) (n : ℕ) : Type := match n [
| zero. ↦ sig ()
| suc. n ↦ sig (
    car : A,
    cdr : Covec A n) ]
```

Then

- `Covec A 0` is a unit type
- `Covec A 1` is a record type isomorphic to `A`
- `Covec A 2` is a record type isomorphic to `A × A`
- etc.

but they are all canonical and don't reduce to anything.

Mixfix notations

User-defined mixfix notations with precedence and associativity.

```
notation 1 plus : x "+" y ... := plus x y

notation 2 times : x "*" y ... := times x y

notation 0 ite : "if" b "then" x "else" y := ite b x y

notation 1.5 types : Γ "⊢" a "::" A := types Γ a A

notation 3 interp : "⟦" M "⟧" := interp M
```

- Unicode operators are allowed, delimited by spaces.
- ASCII operators don't require spaces: `x+y`, `x-y`, `x:A`.
- Special built-ins like `→`, `↦`, `:=` don't require spaces, and have ASCII equivalents `->`, `|->`, `:=`.

Namespaces and imports

Uses Yuujinchou by Favonia/RedPRL for namespacing.

- These both define `foo.bar`, which is in namespace `foo`:

```
def foo.bar := ...
```

```
section foo :=
  def bar := ...
end
```

- Patch `foo` by defining new constants starting with `foo.`
- Import other source files, with a powerful suite of modifiers:

```
import "file" | in foo renaming bar baz
```

- Notations are stored in the `notations` namespace:

```
import "file1" | except notations
import "file2" | union (renaming . file2, only notations)
```

After typechecking a file, a compiled version is written to disk for faster future loading.

Interactive coding and proof

- ProofGeneral Emacs mode allows progressive processing and undoing while working on a source file, like Rocq.
- Can also leave holes `?`, view their inferred types and contexts, and fill them later, like Agda.
- Can split in a hole, automatically inserting abstractions, tuples, or comatches based on its type.
- No tactics yet.

- An opinionated automatic formatter reformats files on load and commands when processed interactively (by default), also changing ASCII `->` into Unicode `→` etc.

Outline

1. Dependent type theory
2. Higher observational type theory
3. Some technical details

Identity types

First principle of HOTT

For `x : A` and `y : A`, we have an identity type `Id A x y`.

Second principle of HOTT

For `x : A`, we have a reflexivity term `refl x : Id A x x`, which synthesizes if `x` does.

We regard the definition of `Id A` (and hence also `Id (Id A)`, `Id (Id (Id A))` etc.) as part of the definition of `A`.

Similarly, the definition of `refl x` is part of the definition of `x`.

For example...

Observational identity types

The type `Id (A × B) p q` behaves as if it were defined by

```
def Id_A×B_p_q : Type ≔ sig (
  fst : Id A (p .fst) (q .fst)
  snd : Id B (p .snd) (q .snd))
```

This makes it behave almost exactly like `Id A (p .fst) (q .fst) × Id B (p .snd) (q .snd)`:

1. Given `r : Id (A × B) p q`, we have

   `r .fst : Id A (p .fst) (q .fst)`

   `r .snd : Id B (p .snd) (q .snd)`

2. Given `s : Id A (p .fst) (q .fst)`
   and `t : Id B (p .snd) (q .snd)`, we have

   `(s, t) : Id (A × B) p q`

Observational reflexivity

Similarly, reflexivity is defined on each term-former:

$$\begin{array}{rcl} \texttt{refl (a, b)} & \equiv & \texttt{(refl a, refl b)} \\ \texttt{refl (p .fst)} & \equiv & \texttt{(refl p) .fst} \\ \texttt{refl (p .snd)} & \equiv & \texttt{(refl p) .snd} \end{array}$$

These compute from left to right. However, since records have η-conversion, it follows that for any $\texttt{p : A} \times \texttt{B}$ we have

$$\texttt{refl p} \equiv \texttt{(refl (p .fst), refl (p .snd))}$$

More observational identity types

- `Id (List A) xs ys` behaves as if it were defined by

```
def Id_ListA : List A → List A → Type := data [
| nil. : Id_ListA nil. nil.
| cons. : {x₀ x₁ : A} (x₂ : Id A x₀ x₁)
    {xs₀ xs₁ : List A} (xs₂ : Id_ListA xs₀ xs₁)
    → Id_ListA (cons. x₀ xs₀) (cons. x₁ xs₁) ]
```

These constructors check at it, and can be matched against.
This makes encode-decode proofs marginally simpler.

- `Id (Stream A) s t` behaves as if it were defined by

```
def Id_StreamA (s t : Stream A) : Type := codata [
| u .head : Id A (s .recv) (t .recv)
| u .tail : Id_StreamA (s .tail) (t .tail) ]
```

These destructors apply to it, and can be comatched against.
Equality in codatatypes is bisimulation, by definition.

Observational function types

`Id (A → B) f g` behaves as if it were defined to equal

$$\texttt{\{x}_0\ \texttt{x}_1\ \texttt{: A\}}\ \texttt{(x}_2\ \texttt{: Id A x}_0\ \texttt{x}_1\texttt{)} \to \texttt{Id B (f x}_0\texttt{) (g x}_1\texttt{)}.$$

This is a little more complicated than the naïvely expected

$$\texttt{(x : A)} \to \texttt{Id B (f x) (g x)}.$$

- Once we have transport, we can prove them equivalent.
- The former is required for parametricity (when no transport).
- For `f : A → B`, we get that `refl f` is "ap", with type

$$\texttt{\{x}_0\ \texttt{x}_1\ \texttt{: A\}}\ \texttt{(x}_2\ \texttt{: Id A x}_0\ \texttt{x}_1\texttt{)} \to \texttt{Id B (f x}_0\texttt{) (f x}_1\texttt{)}.$$

Third principle of HOTT

All constructions preserve equality, in separately defined ways.

Computation with ap

Supposing $\texttt{a}_2 : \texttt{Id A a}_0\ \texttt{a}_1$, we have:

$$\texttt{refl ((x} \mapsto \texttt{(f x, g x)) : A} \to \texttt{B} \times \texttt{C) a}_2 \equiv \texttt{(refl f a}_2\texttt{, refl g a}_2\texttt{)}$$

Supposing $\texttt{p}_2 : \texttt{Id (A} \times \texttt{B)}\ \texttt{p}_0\ \texttt{p}_1$, we have:

$$\texttt{refl ((x} \mapsto \texttt{x .fst) : A} \times \texttt{B} \to \texttt{A) p}_2 \equiv \texttt{p}_2\ \texttt{.fst}$$

$$\texttt{refl ((x} \mapsto \texttt{x .snd) : A} \times \texttt{B} \to \texttt{B) p}_2 \equiv \texttt{p}_2\ \texttt{.snd}$$

Implemented internally using "higher-dimensional substitutions".

On "definitions" of the universe

Traditional perspective ("meaning explanations" and "lower" OTT)

The universe is inductively defined by type-formers as constructors.

- The Tarski eliminator `El` is (inductive-)recursively defined.
- `Id` is also recursively defined with clauses for each constructor, e.g. `Id (A × B)` ≡ `Id A × Id B`.
- Justifies type-case, not univalence.

HOTT perspective

The universe is coinductively defined by `El` and `Id` as destructors.

- `Id` is coinductive: `Id` of a type is another family of types.
- Each type-former is defined corecursively, by specifying `El` (i.e. its intro/elim/...rules) and `Id` (which are generally other instances of the same type-former).
- Justifies univalence, as we will see...

Bisimulations of types

- Equality in a coinductively defined type is "bisimulation".
- Intuitively, two systems are bisimilar if there is a correspondence between their "states" such that
  1. Any state in one system corresponds to some state in the other.
  2. The "subsequent behavior" (coinductive destructors) of corresponding pairs of states are also bisimilar, coinductively.

Fourth principle of HOTT

`Id Type A B` behaves like it consists of bisimulations:

```
def isBisim (A B:Type) (R:A→B→Type) : Type ≔ codata [
| x .trr : A → B
| x .liftr : (a:A) → R a (x .trr a)
| x .trl : B → A
| x .liftl : (b:B) → R (x .trl b) b
| x .id :* (a₀ a₁:A) (b₀ b₁:B) (r₀:R a₀ b₀) (r₁:R a₁ b₁)
    → isBisim (Id A a₀ a₁) (Id B b₀ b₁) ...  ]
```

where `...` is some correspondence induced by `R` (TBD).

Behavior of `Id Type`

Introduction

For $\texttt{R : A} \to \texttt{B} \to \texttt{Type}$ and $\texttt{Rb : isBisim A B R}$, we have

$$\texttt{glue A B R Rb : Id Type A B}.$$

Elimination

Any $\texttt{A}_2 \texttt{ : Id Type A}_0 \texttt{ A}_1$ gives rise to:

- Given $\texttt{a}_0 \texttt{ : A}_0$ and $\texttt{a}_1 \texttt{ : A}_1$, have $\texttt{A}_2 \texttt{ a}_0 \texttt{ a}_1 \texttt{ : Type}$.
- Given $\texttt{a}_0 \texttt{ : A}_0$, have $\texttt{A}_2 \texttt{ .trr a}_0 \texttt{ : A}_1$.
- Given $\texttt{a}_0 \texttt{ : A}_0$, have $\texttt{A}_2 \texttt{ .liftr a}_0 \texttt{ : A}_2 \texttt{ a}_0 \texttt{ (A}_2 \texttt{ .trr a}_0\texttt{)}$.
- Given $\texttt{a}_1 \texttt{ : A}_1$, have $\texttt{A}_2 \texttt{ .trl a}_1 \texttt{ : A}_0$.
- Given $\texttt{a}_1 \texttt{ : A}_1$, have $\texttt{A}_2 \texttt{ .liftl a}_1 \texttt{ : A}_2 \texttt{ (A}_2 \texttt{ .trl a}_1\texttt{) a}_1$.
- Something TBD for "$\texttt{A}_2$ `.id`"...

Behavior of `Id Type`

Introduction

For `R : A → B → Type` and `Rb : isBisim A B R`, we have

`glue A B R Rb : Id Type A B`.

Elimination

Any `A₂ : Id Type A₀ A₁` gives rise to:

- Given `a₀ : A₀` and `a₁ : A₁`, have `A₂ a₀ a₁ : Type`.
- `trr`, `liftr`, `trl`, `liftl`

Computation

For `a : A` and `b : B`, the type `glue A B R Rb a b` behaves like

```
def glueABR_a_b : Type ≔ sig ( unglue : R a b )
```

and `glue A B R Rb .trr` etc. compute to the fields of `Rb`.

Behavior of `Id Type`

Introduction

For $\texttt{R : A} \to \texttt{B} \to \texttt{Type}$ and $\texttt{Rb : isBisim A B R}$, we have

$$\texttt{glue A B R Rb : Id Type A B}.$$

Elimination

Any $\texttt{A}_2 \texttt{ : Id Type } \texttt{A}_0 \texttt{ A}_1$ gives rise to:

- Given $\texttt{a}_0 \texttt{ : A}_0$ and $\texttt{a}_1 \texttt{ : A}_1$, have $\texttt{A}_2 \texttt{ a}_0 \texttt{ a}_1 \texttt{ : Type}$.
- $\texttt{trr}$, $\texttt{liftr}$, $\texttt{trl}$, $\texttt{liftl}$

Consistency

For $\texttt{A : Type}$ with $\texttt{a : A}$ and $\texttt{b : A}$, we have

$$\texttt{Id A a b} \equiv \texttt{refl A a b}.$$

Heterogeneous equality

For `B : A → Type`, if `a₂ : Id A a₀ a₁` we have

`refl B a₂ : Id Type (B a₀) (B a₁)`.

Thus, for `b₀ : B a₀` and `b₁ : B a₁`, we have a type

`refl B a₂ b₀ b₁ : Type`

of dependent or heterogeneous equalities.

These figure in `Id` of dependent records and functions:

- For `B:A→Type`, `Id (Σ A B) p q` behaves like

```
def Id_ΣAB_p_q : Type := sig (
  fst : Id A (p .fst) (q .fst),
  snd : refl B fst (p .snd) (q .snd))
```

- For `B:A→Type`, `Id ((x:A)→B x) f g` behaves like

```
{x₀ x₁ : A} (x₂ : Id A x₀ x₁) → refl B x₂ (f x₀) (g x₁)
```

Co-univalence

Theorem

If $\texttt{A}_2 : \texttt{Id Type A}_0\ \texttt{A}_1$, *then* $\texttt{A}_2\ \texttt{.trr} : \texttt{A}_0 \to \texttt{A}_1$ *and* $\texttt{A}_2\ \texttt{.trl} : \texttt{A}_1 \to \texttt{A}_0$ *are inverse equivalences.*

Proof.

Given $\texttt{a}_0 : \texttt{A}_0$, have $\texttt{A}_2\ \texttt{.liftr a}_0 : \texttt{A}_2\ \texttt{a}_0\ (\texttt{A}_2\ \texttt{.trr a}_0)$ and

$$\texttt{A}_2\ \texttt{.liftl}\ (\texttt{A}_2\ \texttt{.trr a}_0) : \texttt{A}_2\ (\texttt{A}_2\ \texttt{.trl}\ (\texttt{A}_2\ \texttt{.trr a}_0))\ (\texttt{A}_2\ \texttt{.trr a}_0)$$

Therefore, we have an induced bisimulation between

$$\texttt{Id A}_0\ \texttt{a}_0\ (\texttt{A}_2\ \texttt{.trl}\ (\texttt{A}_2\ \texttt{.trr a}_0))$$ and
$$\texttt{Id A}_1\ (\texttt{A}_2\ \texttt{.trr a}_0)\ (\texttt{A}_2\ \texttt{.trr a}_0).$$

Thus, since $\texttt{refl}\ (\texttt{A}_2\ \texttt{.trr a}_0)$ inhabits the latter, the former is also inhabited. The other direction is dual. □

Univalence

Theorem

If `f : A → B` *is an equivalence, then*

$$\texttt{(a b} \mapsto \texttt{Id B (f a) b) : A} \to \texttt{B} \to \texttt{Type}$$

is a bisimulation. Hence we get `ua f : Id Type A B`.

Main idea of proof.

If `f : A → B` is an equivalence, so is each

$$\texttt{refl f } \{\texttt{a}_0\} \ \{\texttt{a}_1\} : \texttt{Id A a}_0 \ \texttt{a}_1 \to \texttt{Id B (f a}_0\texttt{) (f a}_1\texttt{)}$$

Therefore, we can use corecursion. □

The more things change...

On the n-Category Café blog comments, December 2009:

Peter Lumsdaine: ...asking a [space] to be contractible involves arbitrary high dimensions, and I'm not sure what kind of language ...would let you talk about that...a type may have infinitely high non-trivial structure... [but] you can only work with...finite-dimensional approximations.

Me: One way...to talk about contractibility is by coinduction. If A is contractible...(1) A is inhabited, and (2) for any $x, y \in A$, $\hom_A(x, y)$ is contractible. Moreover, contractibility is maximal with this property...

Peter Lumsdaine: ...you might well also want to talk about [equivalences] defined coinductively, as...in e.g. [Eugenia] Cheng "An ω-category with all duals is an ω-groupoid".

Then Voevodsky came along and we went on a 15-year detour...

Status of the implementation

- Run `narya -hott` to get the HOTT implementation.
- Everything I've described so far is implemented except that the fields `trr`, etc. don't compute.
- Warning: highly in flux, syntax is likely to change.

- Omitting `-hott` gives an internally parametric type theory, with `Id` but no `trr`.
  - Here we can define a notion of "fibrancy" that models HOTT.
  - Everything is proven to compute except `trr` on the universe.
- Can vary the `-arity` and `-internal`-ness of parametricity.
  - "External" unary case is a version of displayed type theory, in which we can define semi-simplicial types (jww Kolomatskaia).
- Eventually `-hott` will always be on, and other "directions" of parametricity can coexist with it.

Remarks on the implementation

- 27,000 lines of OCaml
- Normalization by evaluation
- Dependently typed programming
  - Intrinsically well-scoped De Bruijn indices
  - Type-level dimensions
- Algebraic effects
- Thanks to Favonia/RedPRL
  - Yuujinchou: imports, scoping
  - Asai: error reporting
  - Algaeff: algebraic effects
  - Bwd: backwards lists

Outline

1. Dependent type theory
2. Higher observational type theory
3. Some technical details

What's missing

1. The complete type of `id` in `isBisim`.
2. Something to represent `id` acting on $\texttt{A}_2 : \texttt{Id Type A}_0\ \texttt{A}_1$.
3. A *lie.

Completing* the definition of bisimulation

Any `R:A→B→Type` induces

```
refl R : {a₀ a₁:A} (a₂:Id A a₀ a₁) {b₀ b₁:B} (b₂:Id B b₀ b₁)
         → Id Type (R a₀ b₀) (R a₁ b₁)
```

So if we have `a₀ a₁ : A` and `b₀ b₁ : B`, and also `r₀:R a₀ b₀` and `r₁:R a₁ b₁`, then

```
(a₂ b₂ ↦ refl R a₂ b₂ r₀ r₁) : Id A a₀ a₁ → Id B b₀ b₁ → Type
```

and this is what appears in the method `id` of `isBisim`:

```
def isBisim (A B:Type) (R:A→B→Type) : Type ≔ codata [
| x .trr : A → B
| x .liftr : (a:A) → R a (x .trr a)
| x .trl : B → A
| x .liftl : (b:B) → R (x .trl b) b
| x .id :* (a₀ a₁:A) (b₀ b₁:B) (r₀:R a₀ b₀) (r₁:R a₁ b₁)
    → isBisim (Id A a₀ a₁) (Id B b₀ b₁)
        (a₂ b₂ ↦ refl R a₂ b₂ r₀ r₁) ]
```

Squares

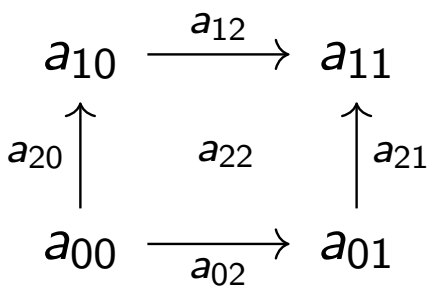
Given `A : Type` we have `Id A : A → A → Type`, thus

```
refl (Id A) : {a00 a01 : A} (a02 : Id A a00 a01)
              {a10 a11 : A} (a12 : Id A a10 a11)
    → Id Type (Id A a00 a10) (Id A a01 a11)
```

So if we have `a00 a01 a02 a10 a11 a12` and also `a20 : Id A a00 a10` and `a21 : Id A a01 a11`, we get

```
refl (Id A) a02 a12 a20 a21 : Type
```

whose elements are squares in `A`:



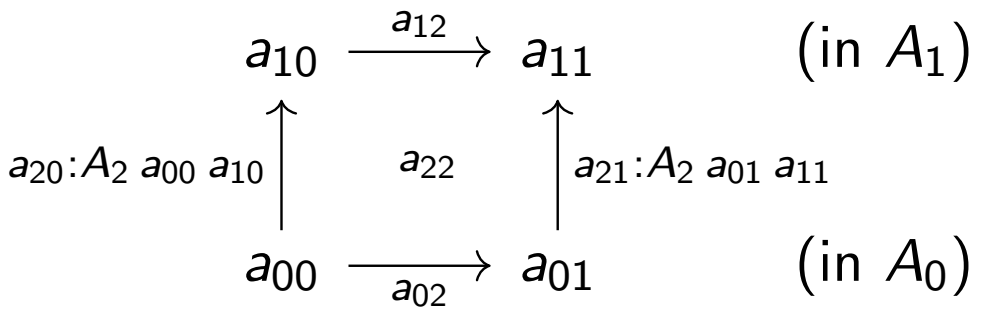
`refl (Id A) a02 a12 .trr` and so on give left-right box-filling.

Heterogeneous squares

Similarly, for `A₂ : Id Type A₀ A₁`, we have

```
refl ((x y ↦ A₂ x y) : A₀ → A₁ → Type)
  : {a₀₀ a₀₁ : A₀} (a₀₂ : Id A₀ a₀₀ a₀₁)
    {a₁₀ a₁₁ : A₁} (a₁₂ : Id A₁ a₁₀ a₁₁)
    → Id Type (A₂ a₀₀ a₁₀) (A₂ a₀₁ a₁₁)
```

giving types of heterogeneous squares

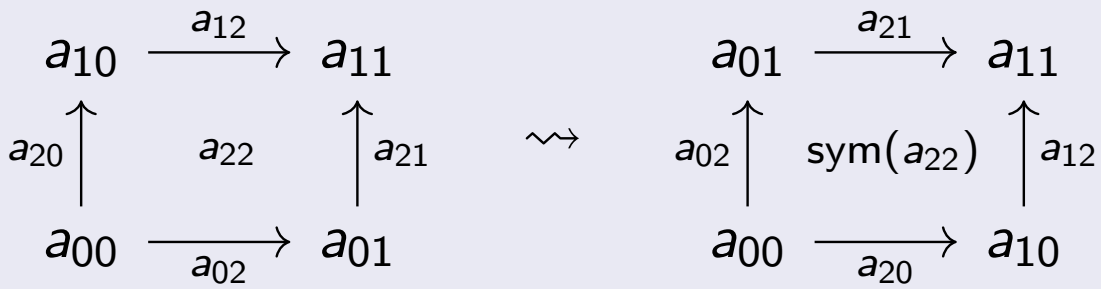


with left-right filling. But for "`A₂ .id`" we want top-bottom filling.

Symmetry

Fifth (and last) principle of HOTT

Every square has an associated symmetric/transposed square:



We define these separately for each construction of squares. They are functorial and coherent, and generalize to heterogeneous squares.

For `A₂ : Id Type A₀ A₁`, the last bisimulation method "`A₂ .id`" is represented by `sym (refl A₂)`.

(We also have to deal with squares that are heterogeneous in both directions. This is why our definition of `isBisim` is a *lie, and it leads to a surprisingly deep and beautiful rabbit hole of "higher coinductive types", which I omit today.)

Id-elimination

Just as in cubical type theory, from top-bottom box-filling and transport we derive Martin-Löf Id-elimination, with typal β-rule:

```
def J (A : Type) (a : A) (P : (y : A) → Id A a y → Type)
  (pa : P a (refl a)) (b : A) (p : Id A a b)
  : P b p
  := let sq := refl ((y ↦ Id A a y) : A → Type) p in
     refl P (sq .trr (refl a))
       (sym (sq .liftr (refl a))) .trr pa
```

```
def Jβ (A : Type) (a : A) (P : (y : A) → Id A a y → Type)
  (pa : P a (refl a))
  : Id (P a (refl a)) pa (J A a P pa a (refl a))
  := ...
```