Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities

Carlo Angiuli[1] Favonia[2] Robert Harper[1]

[1]Carnegie Mellon University

[2]University of Minnesota

CSL 2018

Equality and dependency

This work is about equality in dependent type theory.

The goal of my talk is to explain the long title.

Equality and dependency

This work is about equality in dependent type theory.

A general-purpose constructive logic and programming language used in many proof assistants. (Coq, Agda, Lean, Nuprl . . .)

The goal of my talk is to explain the long title.

Equality and dependency

Types are indexed by (dependent on) terms.

$\mathbf{List}\ A\ n$	type of lists of length n
$(n : \mathbf{nat}) \to \mathbf{List}\ A\ n$	dependent function type

$$\mathbf{append} : (n_1, n_2 : \mathbf{nat}) \to \mathbf{List}\ A\ n_1 \to \mathbf{List}\ A\ n_2 \to \mathbf{List}\ A\ (n_1 + n_2)$$

If you concatenate two lists of length one, is the result directly a list of length two?

Equality and dependency

Types are indexed by (dependent on) terms.

$\mathbf{List}\ A\ n$	type of lists of length n
$(n : \mathbf{nat}) \to \mathbf{List}\ A\ n$	dependent function type

$$\mathbf{append} : (n_1, n_2 : \mathbf{nat}) \to \mathbf{List}\ A\ n_1 \to \mathbf{List}\ A\ n_2 \to \mathbf{List}\ A\ (n_1 + n_2)$$

If $\ell : \mathbf{List}\ A\ (1+1)$,

- $\ell : \mathbf{List}\ A\ 2$?
- "$\mathbf{coe}_{(1+1=2)}(\ell)$" : $\mathbf{List}\ A\ 2$?

If you concatenate two lists of length one, is the result directly a list of length two?

Equality and dependency

Types are indexed by (dependent on) terms.

$\mathbf{List}\ A\ n$	type of lists of length n
$(n : \mathbf{nat}) \to \mathbf{List}\ A\ n$	dependent function type

$$\mathbf{append} : (n_1, n_2 : \mathbf{nat}) \to \mathbf{List}\ A\ n_1 \to \mathbf{List}\ A\ n_2 \to \mathbf{List}\ A\ (n_1 + n_2)$$

If $\ell : \mathbf{List}\ A\ (1 + 1)$,

- $\ell : \mathbf{List}\ A\ 2$? ✓
- "$\mathbf{coe}_{(1+1=2)}(\ell)$" $: \mathbf{List}\ A\ 2$? ✗

If you concatenate two lists of length one, is the result directly a list of length two?

Extensional type theory

Two main variations on type/term equality.

In extensional type theory, $\mathbf{Eq}_A(a_1, a_2)$:

- Rewriting along equalities is silent (no $\mathbf{coe}$).
- Equality of functions is extensional:
  $\mathbf{Eq}_{\mathbf{nat}\to\mathbf{nat}\to\mathbf{nat}}((\lambda n_1, n_2.n_1 + n_2), (\lambda n_1, n_2.n_2 + n_1))$

Intensional type theory

Intensional type theory has two notions of equality:

- Definitional equality ($a_1 \equiv a_2 : A$) is syntactic ($\alpha\beta(\eta)$) and silent.
- Intensional identity ($\mathbf{Id}_A(a_1, a_2)$) requires explicit $\mathbf{co}$ercions.

Intensional type theory

$\mathbf{Id}_A(a_1, a_2)$ doesn't interact properly with type formers:

- Not extensional for functions: can't prove $\mathbf{Id}_{\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}}((\lambda n_1, n_2.n_1 + n_2), (\lambda n_1, n_2.n_2 + n_1))$.
- Identity of identities is not trivial: can't prove $\mathbf{Id}_{(\mathbf{Id}_A(a_1, a_2))}(p_1, p_2)$.

Homotopy type theory leverages the latter property of the $\mathbf{Id}$ type.

Intensional type theory

$\mathbf{Id}_A(a_1, a_2)$ doesn't interact properly with type formers:

- Not extensional for functions: can't prove $\mathbf{Id}_{\mathbf{nat}\to\mathbf{nat}\to\mathbf{nat}}((\lambda n_1, n_2.n_1 + n_2), (\lambda n_1, n_2.n_2 + n_1))$.
- Identity of identities is not trivial: can't prove $\mathbf{Id}_{(\mathbf{Id}_A(a_1,a_2))}(p_1, p_2)$.

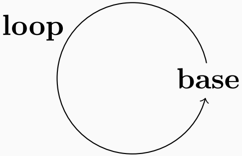
Make lemonade from these lemons: add non-trivial ~~identities~~ paths.

Homotopy type theory leverages the latter property of the $\mathbf{Id}$ type.

Homotopy type theory

Higher inductive types with path generators.

$$\mathbf{S}^1 : \mathcal{U}$$
$$\mathbf{base} : \mathbf{S}^1$$
$$\mathbf{loop} : \mathbf{Id}_{\mathbf{S}^1}(\mathbf{base}, \mathbf{base})$$



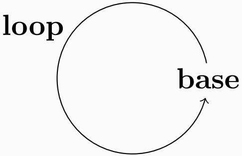
Synthetic treatment of homotopy groups, cohomology, ...

Homotopy type theory

Higher inductive types with path generators.

$$\mathbf{S}^1 : \mathcal{U}$$
$$\mathbf{base} : \mathbf{S}^1$$
$$\mathbf{loop} : \mathbf{Id}_{\mathbf{S}^1}(\mathbf{base}, \mathbf{base})$$

$$\mathbf{loop}^2 : \mathbf{Id}_{\mathbf{S}^1}(\mathbf{base}, \mathbf{base})$$
$$\mathbf{loop}^{-1} : \mathbf{Id}_{\mathbf{S}^1}(\mathbf{base}, \mathbf{base})$$
$$\vdots$$



Synthetic treatment of homotopy groups, cohomology, ...

Homotopy type theory

Univalence: A, B homotopy-equivalent $\iff \mathbf{Id}_{\mathcal{U}}(A, B)$.
Makes "mathematics up to isomorphism" fully precise.

$$\mathbf{bool} \to A \qquad \simeq \qquad A \times A$$

$$f \xmapsto{\quad\mathbf{iso}\quad} \langle f\ \mathbf{true}, f\ \mathbf{false} \rangle$$

Coercions across univalence can't be silent, because isomorphic types have different elements. Neither can one avoid specifying a particular isomorphism, because different ones induce different coercions.

Homotopy type theory

Univalence: A, B homotopy-equivalent $\iff \mathbf{Id}_{\mathcal{U}}(A, B)$.
Makes "mathematics up to isomorphism" fully precise.

$$\mathbf{bool} \to A \qquad\qquad \simeq \qquad\qquad A \times A$$

$$f \xmapsto{\quad\mathbf{iso}\quad} \langle f\ \mathbf{true}, f\ \mathbf{false} \rangle$$

$$\ell : \mathbf{List}\ (\mathbf{bool} \to A)\ n \implies \text{“}\mathbf{coe}_{\mathbf{iso}}(\ell)\text{”} : \mathbf{List}\ (A \times A)\ n$$

Coercions across univalence can't be silent, because isomorphic types have different elements. Neither can one avoid specifying a particular isomorphism, because different ones induce different coercions.

Homotopy type theory

Univalence: A, B homotopy-equivalent $\iff \mathbf{Id}_{\mathcal{U}}(A, B)$.
Makes "mathematics up to isomorphism" fully precise.

$$\mathbf{bool} \to A \qquad \simeq \qquad A \times A$$

$$\begin{tikzcd} f \arrow[r, mapsto, "\mathbf{iso}"] \arrow[dr, mapsto, "\color{red}\mathbf{iso}'"'] & \langle f\ \mathbf{true}, f\ \mathbf{false} \rangle \\ & \langle f\ \mathbf{false}, f\ \mathbf{true} \rangle \end{tikzcd}$$

$$\ell : \mathbf{List}\ (\mathbf{bool} \to A)\ n \implies \text{“}\mathbf{coe}_{\color{red}\mathbf{iso}}(\ell)\text{”} : \mathbf{List}\ (A \times A)\ n$$

Coercions across univalence can't be silent, because isomorphic types have different elements. Neither can one avoid specifying a particular isomorphism, because different ones induce different coercions.

Constructivity?

Univalence/HITs added as axioms without computational meaning.

"$\mathbf{coe_{iso}}(\ell)$" : $\mathbf{List}\ (A \times A)\ n$ doesn't compute to a list of pairs.

Constructivity?

Univalence/HITs added as axioms without computational meaning.

"$\mathbf{coe}_{\mathbf{iso}}(\ell)$" : $\mathbf{List}\ (A \times A)\ n$ doesn't compute to a list of pairs.

Definition (Canonicity)

If $\cdot \vdash M : \mathbf{bool}$, then M computes to (and is silently equal to) either $\mathbf{true}$ or $\mathbf{false}$.

Contributions

Type theory with univalence/HITs and also canonicity!

- Second such type theory. (Cohen et al., 2016)
- Novel ("Cartesian cubical") method.

Has both silent, extensional equality ($\mathbf{Eq}_A(a_1, a_2)$) and non-silent paths ($\mathbf{Path}_A(a_1, a_2)$) mediating univalence/HITs.

- First "two-level" type theory with canonicity.
- Which equalities can or cannot be silent?

Computational type theory

Computational type theory

Inspired by Nuprl, we build our type theory around a PER semantics in which proofs are programs.

- *Constructive mathematics and computer programming* (Martin-Löf, 1979)
- *A non-type-theoretic definition of Martin-Löf's types* (Allen, 1987)
- Logical relations (Tait, 1967), ...

These ideas have cropped up in many different guises, but our development is closest to Martin-Löf's meaning explanations, and to Allen's PER semantics.

Computational type theory

Untyped syntax; operational semantics on closed terms.

$$\begin{aligned} M := {} & (a : A) \to B \mid \lambda a.M \mid \mathbf{app}(M, N) \\ & \mid (a{:}A) \times B \mid \langle M, N \rangle \mid \mathbf{fst}(M) \mid \mathbf{snd}(M) \\ & \mid \mathbf{bool} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}_{b.A}(M; T, F) \mid \cdots \end{aligned}$$

$$\frac{}{\mathbf{bool\ val}} \qquad \frac{}{\mathbf{true\ val}} \qquad \frac{}{\mathbf{false\ val}}$$

$$\frac{M \longmapsto M'}{\mathbf{if}_{b.A}(M; T, F) \longmapsto \mathbf{if}_{b.A}(M'; T, F)}$$

$$\frac{}{\mathbf{if}_{b.A}(\mathbf{true}; T, F) \longmapsto T} \qquad \frac{}{\mathbf{if}_{b.A}(\mathbf{false}; T, F) \longmapsto F} \qquad \cdots$$

Booleans

Types classify (closed) programs according to their behaviors.

Definition

- $M \in \mathbf{bool}$ if $M \longmapsto^* \mathbf{true}$ or $M \longmapsto^* \mathbf{false}$.
- $M \doteq N \in \mathbf{bool}$ if $M, N \longmapsto^* \mathbf{true}$ or $M, N \longmapsto^* \mathbf{false}$.

Types are partial equivalence relations closed under evaluation.

Notice that canonicity holds by definition. The hard part is making sure that *all* the constructs of our type theory have computational meaning; $\mathbf{true}$ and $\mathbf{false}$ trivially do.

Booleans

Types classify (closed) programs according to their behaviors.

Definition

- $M \in \mathbf{bool}$ if $M \longmapsto^* \mathbf{true}$ or $M \longmapsto^* \mathbf{false}$.
- $M \doteq N \in \mathbf{bool}$ if $M, N \longmapsto^* \mathbf{true}$ or $M, N \longmapsto^* \mathbf{false}$.

Types are partial equivalence relations closed under evaluation.

Canonicity is true by construction!

Notice that canonicity holds by definition. The hard part is making sure that *all* the constructs of our type theory have computational meaning; $\mathbf{true}$ and $\mathbf{false}$ trivially do.

Functions

Open terms are regarded as functions (via substitution).

Definition

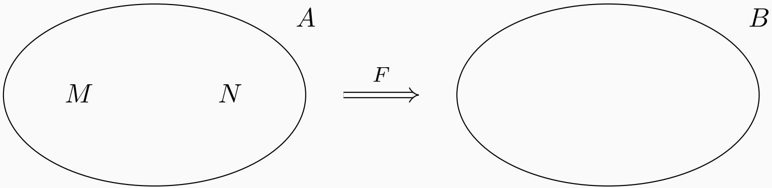
$\lambda a.M \in A \to B$ when for any $N_1 \doteq N_2 \in A$,
$M[N_1/a] \doteq M[N_2/a] \in B$.

Functions map silently equal arguments to silently equal results.

Paths?

How do functions act on paths (non-silent equalities)?

Given $F \in A \to B$ and $P \in \mathbf{Path}_A(M, N)$:

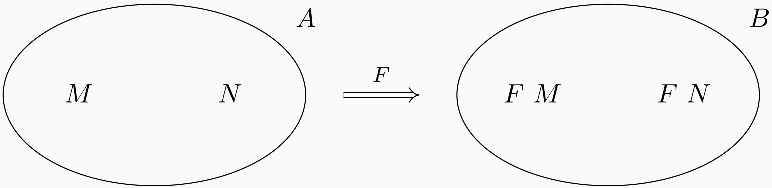


$F\ P$ does not make type sense, because P is a path, not an element of A.

Paths?

How do functions act on paths (non-silent equalities)?

Given $F \in A \to B$ and $P \in \mathbf{Path}_A(M, N)$:

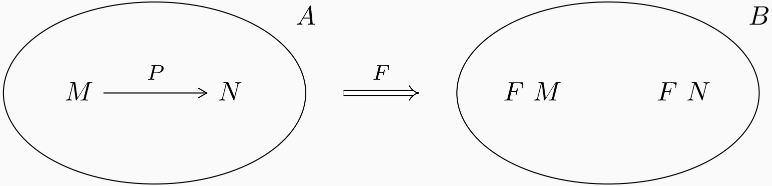


$F\ P$ does not make type sense, because P is a path, not an element of A.

Paths?

How do functions act on paths (non-silent equalities)?

Given $F \in A \to B$ and $P \in \mathbf{Path}_A(M, N)$:

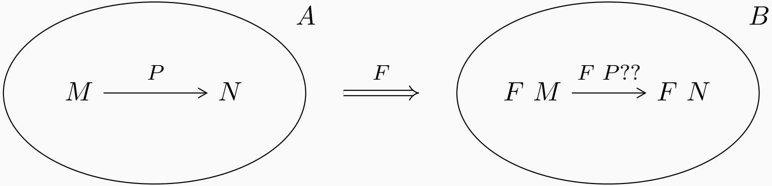


$F\ P$ does not make type sense, because P is a path, not an element of A.

Paths?

How do functions act on paths (non-silent equalities)?

Given $F \in A \to B$ and $P \in \mathbf{Path}_A(M, N)$:



$F\ P$ does not make type sense, because P is a path, not an element of A.

Interval variables

Represent P with formal dependence on interval variable x.

$$M = P(0) \xrightarrow{\quad P(x) \quad} P(1) = N$$

$F\ P$ makes sense since we can weaken F by x.

$$(F\ P)\langle 0/x\rangle \xrightarrow{\quad F\ P \quad} (F\ P)\langle 1/x\rangle$$

Interval variables

Represent P with formal dependence on interval variable x.

$$M = P(0) \xrightarrow{\quad P(x) \quad} P(1) = N$$

$F\ P$ makes sense since we can weaken F by x.

$$F\ P\langle 0/x\rangle \xrightarrow{\quad F\ P \quad} F\ P\langle 1/x\rangle$$

Interval variables

Represent P with formal dependence on interval variable x.

$$M = P(0) \xrightarrow{\quad P(x) \quad} P(1) = N$$

$F\ P$ makes sense since we can weaken F by x.

$$F\ M \xrightarrow{\quad F\ P \quad} F\ N$$

Interval variables

Add primitive paths for HITs and univalence ($E \in A \simeq B$).

$$\mathbf{base} \xrightarrow{\mathbf{loop}_x} \mathbf{base}$$

$$A \xrightarrow{\mathbf{V}_x(A, B, E)} B$$

$$\vdots$$

Terms can depend on an arbitrary number of interval variables.

Interval variables

If $M(x, y)$, then:

- Can degenerate M by weakening by z.
- Can compute faces by instantiating x, y at $0, 1$.
- Can compute the diagonal by contracting x and y.

These interval variables induce cubical structure.

Interval variables

If $M(x, y)$, then:

- Can degenerate M by weakening by z.
- Can compute faces by instantiating x, y at $0, 1$.
- Can compute the diagonal by contracting x and y.

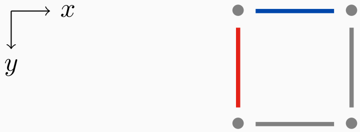
$M\langle 0/x\rangle$

These interval variables induce cubical structure.

Interval variables

If $M(x, y)$, then:

- Can degenerate M by weakening by z.
- Can compute faces by instantiating x, y at $0, 1$.
- Can compute the diagonal by contracting x and y.



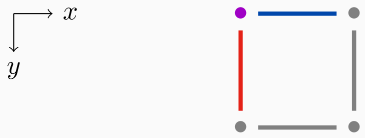
$M\langle 0/x \rangle$ $\quad$ $M\langle 0/y \rangle$

These interval variables induce cubical structure.

Interval variables

If $M(x, y)$, then:

- Can degenerate M by weakening by z.
- Can compute faces by instantiating x, y at $0, 1$.
- Can compute the diagonal by contracting x and y.



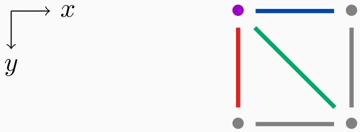
$$M\langle 0/x\rangle\langle 0/y\rangle \;=\; M\langle 0/y\rangle\langle 0/x\rangle$$

These interval variables induce cubical structure.

Interval variables

If $M(x, y)$, then:

- Can degenerate M by weakening by z.
- Can compute faces by instantiating x, y at $0, 1$.
- Can compute the diagonal by contracting x and y.



$$M\langle 0/x\rangle\langle 0/y\rangle \;=\; M\langle 0/y\rangle\langle 0/x\rangle$$

These interval variables induce cubical structure.

Now you're computing with cubes!

Cubical operational semantics

Extend syntax; allow evaluating terms with free interval variables.

$$r := 0 \mid 1 \mid x$$
$$M := \cdots \mid \mathbf{base} \mid \mathbf{loop}_{r} \mid \cdots$$

$$\mathbf{base} \xrightarrow{\mathbf{loop}_x} \mathbf{base}$$

$$\frac{}{\mathbf{base}\ \mathbf{val}} \qquad \frac{}{\mathbf{loop}_x\ \mathbf{val}}$$

Cubical operational semantics

Extend syntax; allow evaluating terms with free interval variables.

$$\begin{aligned} r &:= 0 \mid 1 \mid x \\ M &:= \cdots \mid \mathbf{base} \mid \mathbf{loop}_{\color{red}r} \mid \cdots \end{aligned}$$

$$\mathbf{loop}_0 \doteq \mathbf{base} \xrightarrow{\mathbf{loop}_x} \mathbf{base} \doteq \mathbf{loop}_1$$

$$\frac{}{\mathbf{base}\ \mathbf{val}} \qquad \frac{}{\mathbf{loop}_x\ \mathbf{val}} \qquad \frac{}{\mathbf{loop}_0 \longmapsto \mathbf{base}} \qquad \frac{}{\mathbf{loop}_1 \longmapsto \mathbf{base}}$$

Cubical PERs

Every type now has a PER of n-dimensional elements at each n:

$$M \doteq N \in A \; [x_1, \ldots, x_n]$$

Cubical PERs

Presheaf over finite-product category generated by $1 \rightrightarrows \mathbb{I}$.

Hence, Cartesian cubical type theory.

$$\{M \mid M \in A\ [x,y]\}$$

$$\{M \mid M \in A\ [x]\}$$

$$\{M \mid M \in A\ [\cdot]\}$$

Cubical PERs

Presheaf over finite-product category generated by $1 \rightrightarrows \mathbb{I}$.

Hence, Cartesian cubical type theory.

$$\begin{tikzcd}
\{M \mid M \in A\ [x,y]\} \arrow[d, bend right, "\langle 0/y \rangle"'] \arrow[d, "\langle x/y \rangle\ \cdots"] \\
\{M \mid M \in A\ [x]\} \arrow[d, bend right, "\langle 0/x \rangle"'] \arrow[d, bend left, "\langle 1/x \rangle"] \\
\{M \mid M \in A\ [\cdot]\} \arrow[u]
\end{tikzcd}$$

Cubical PERs

Presheaf over finite-product category generated by $1 \rightrightarrows \mathbb{I}$.

Hence, Cartesian cubical type theory.

$$
\begin{tikzcd}
\{M \mid M \in A\ [x,y]\} \arrow[d, bend right, "\langle 0/y \rangle"'] \arrow[d, "\langle x/y \rangle\ \cdots"] \\
\{M \mid M \in A\ [x]\} \arrow[d, bend right, "\langle 0/x \rangle"'] \arrow[d, bend left, "\langle 1/x \rangle"] \\
\{M \mid M \in A\ [\cdot]\} \arrow[u]
\end{tikzcd}
$$

Must be closed under both evaluation and reindexing, and these must commute (up to $\doteq$).

Paths

Elements of path type are functions out of the interval.

$$\frac{P \in A \ [\Psi, x]}{\langle x \rangle P \in \mathbf{Path}_A(P\langle 0/x \rangle, P\langle 1/x \rangle) \ [\Psi]} \qquad \frac{M \in \mathbf{Path}_A(P_0, P_1) \ [\Psi]}{M@r \in A \ [\Psi]}$$

Coercion

Respect for paths is implemented by a coercion operator.

$$\frac{A\ \mathbf{type}\ [\Psi, x] \qquad M \in A\langle r/x\rangle\ [\Psi]}{\mathbf{coe}_{x.A}^{r\rightsquigarrow r'}(M) \in A\langle r'/x\rangle\ [\Psi]}$$

$$\begin{tikzcd}
M \arrow[d, phantom, "\rotatebox{90}{$\in$}"] & \\
A\langle 0/x\rangle \arrow[r, "A"'] & A\langle 1/x\rangle
\end{tikzcd}$$

Coercion

Respect for paths is implemented by a coercion operator.

$$\frac{A\ \mathbf{type}\ [\Psi, x] \qquad M \in A\langle r/x\rangle\ [\Psi]}{\mathbf{coe}_{x.A}^{r \rightsquigarrow r'}(M) \in A\langle r'/x\rangle\ [\Psi]}$$

$$\begin{tikzcd}
M \arrow[r, dashed] \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] & \mathbf{coe}_{x.A}^{0 \rightsquigarrow 1}(M) \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] \\
A\langle 0/x\rangle \arrow[r, "A"'] & A\langle 1/x\rangle
\end{tikzcd}$$

Coercion

Respect for paths is implemented by a coercion operator.

$$\frac{A \ \mathbf{type}\ [\Psi, x] \qquad M \in A\langle r/x\rangle\ [\Psi]}{\mathbf{coe}_{x.A}^{r \rightsquigarrow r'}(M) \in A\langle r'/x\rangle\ [\Psi]}$$

$$\begin{tikzcd}
M \arrow[rr, "\mathbf{coe}_{x.A}^{0 \rightsquigarrow x}(M)"] & & \mathbf{coe}_{x.A}^{0 \rightsquigarrow 1}(M) \\
\rotatebox[origin=c]{-90}{$\in$} & \rotatebox[origin=c]{-90}{$\in$} & \rotatebox[origin=c]{-90}{$\in$} \\
A\langle 0/x\rangle \arrow[rr, "A"'] & & A\langle 1/x\rangle
\end{tikzcd}$$

Coercion

$$\begin{array}{ccc} \ell & & \\ \rotatebox{90}{$\in$} & & \\ \mathbf{List}\ (\mathbf{bool} \to A)\ n & \qquad\qquad\qquad & \mathbf{List}\ (A \times A)\ n \end{array}$$

Coercion

$$
\begin{tikzcd}
\ell \arrow[rr, dashed] & & \mathbf{coe}^{\color{red}{0 \rightsquigarrow 1}}_{x.\mathbf{List}\ \mathbf{V}_x(\ldots,\mathbf{iso})\ n}(\ell) \\
\pitchfork & & \pitchfork \\
\mathbf{List}\ (\mathbf{bool} \to A)\ n \arrow[rr, "\mathbf{List}\ \mathbf{V}_x(\ldots,\mathbf{iso})\ n"'] & & \mathbf{List}\ (A \times A)\ n
\end{tikzcd}
$$

Coercion

But exact equality doesn't respect paths!

$$\begin{tikzcd}
\mathbf{refl} \arrow[rr, dashed] & & ?? \\
\rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} \\
\mathbf{Eq}_{\mathcal{U}}(A, A) \arrow[rr, "\mathbf{Eq}_{\mathcal{U}}(A, \mathbf{V}_x(A, B, \mathbf{iso}))"'] & & \mathbf{Eq}_{\mathcal{U}}(A, B)
\end{tikzcd}$$

Coercion

But exact equality doesn't respect paths!

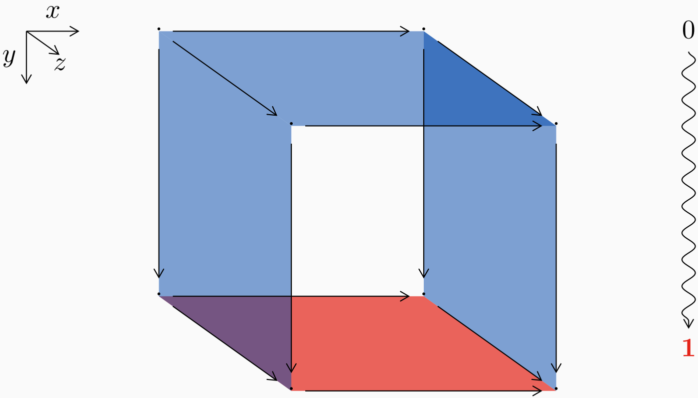
$$\begin{tikzcd}
\mathbf{refl} \arrow[rr, dashed] & & ?? \\
\rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} \\
\mathbf{Eq}_{\mathcal{U}}(A, A) \arrow[rr, "\mathbf{Eq}_{\mathcal{U}}(A, \mathbf{V}_x(A, B, \mathbf{iso}))"'] & & \mathbf{Eq}_{\mathcal{U}}(A, B)
\end{tikzcd}$$

We must stratify types into two levels:

- Kan types (with coercion), and
- pretypes (without coercion).

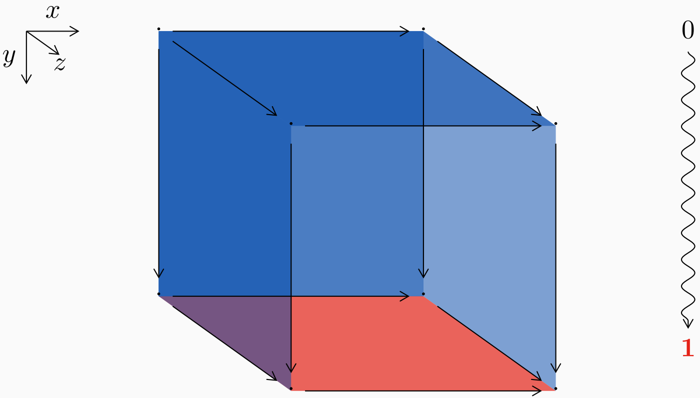
Kan composition

To implement coercion at every type, we also need:



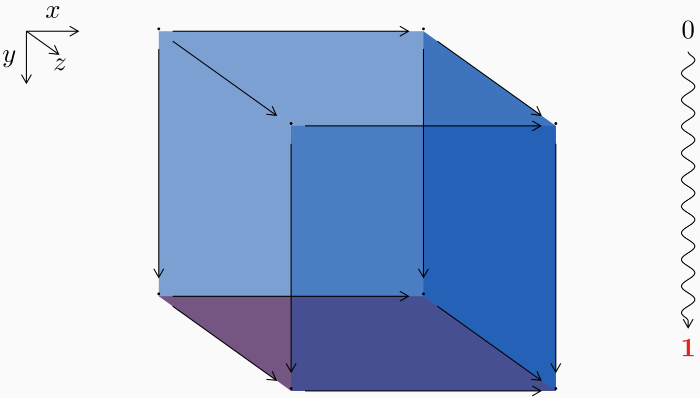
Kan composition

To implement coercion at every type, we also need:



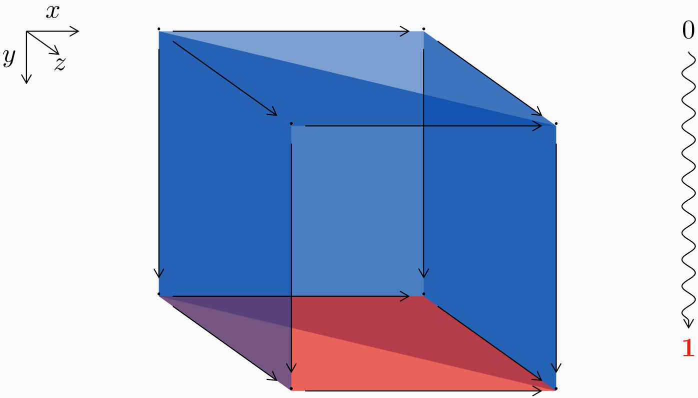
Kan composition

To implement coercion at every type, we also need:



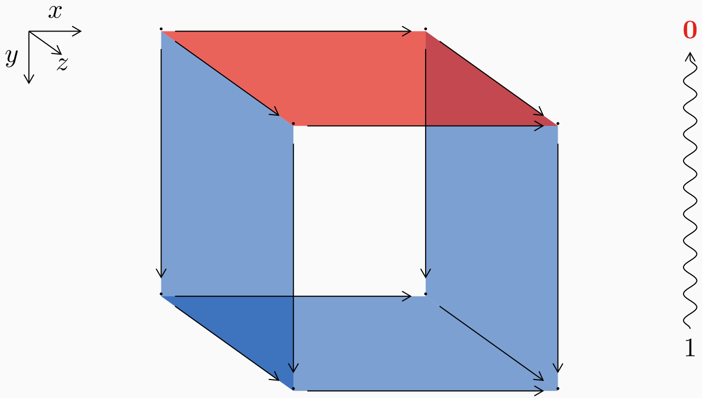
Kan composition

To implement coercion at every type, we also need:



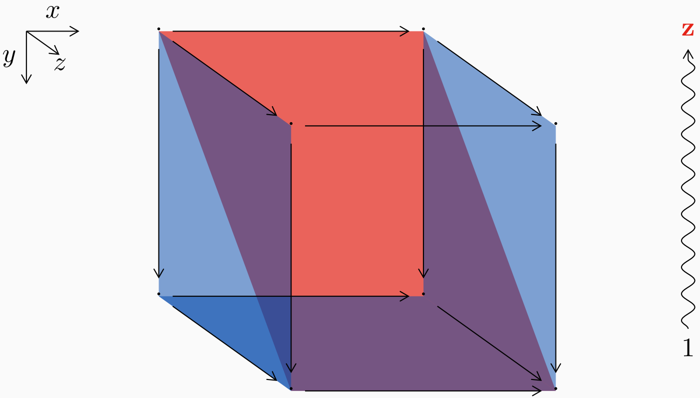
Kan composition

To implement coercion at every type, we also need:



Kan composition

To implement coercion at every type, we also need:



Kan composition in $\mathcal{U}$

Compositions of types must be a new type-former.

What are its elements? How do you coerce and compose in it?

$$?? \quad \in \quad \begin{tikzcd} \cdot \arrow[r, "B"] \arrow[d, "A"'] & \cdot \arrow[d, "C"] \\ \cdot \arrow[r, dashed] & \cdot \end{tikzcd}$$

Conclusion

Cartesian Cubical Computational Type Theory:
Constructive Reasoning with Paths and Equalities

De Morgan cubical type theory

Cubical Type Theory: a constructive interpretation of the univalence axiom (Cohen, Coquand, Huber, Mörtberg, 2016)

More cubical structure and less Kan structure.

$$
\begin{tikzcd}
a \arrow[r, "p"] \arrow[d, "p"'] \arrow[dr, phantom, "p\langle x \vee y/x\rangle"] & b \arrow[d] \\
b \arrow[r] & b
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[r] \arrow[d] \arrow[dr, phantom, "p\langle x \wedge y/x\rangle"] & a \arrow[d, "p"] \\
a \arrow[r, "p"'] & b
\end{tikzcd}
\qquad
b \xrightarrow{p\langle 1-x/x\rangle} a
$$

Two-level type theory

Homotopy Type System (HTS) of Voevodsky (2013).

Want to internally define type-valued presheaves, but functoriality-up-to-paths requires infinite coherence data.

We have defined semi-simplicial types in **RedPRL**!

Implementations

Two prototype tactic-based proof assistants: **RedPRL** and **redtt**.

- Developed by Sterling, Favonia, Angiuli, Cavallo, et al.
- Open-source, available on `github.com/RedPRL`.
- **RedPRL**: à la Nuprl, direct reasoning about untyped terms.
- **redtt**: typed core language of proofs.

Thanks!