LAST TIME: Identity Paths

(id) $\mathrm{refl}_A(a) : a =_A a$

p^{-1} $\mathrm{sym}(p) : b =_A a$ if $p : a =_A b$

$p \cdot q$ $\mathrm{trans}(p, q) : a =_A c$ if $p : a =_A b$ and $q : b =_A c$

We argued that this type had groupoid structure

$\mathrm{refl}^{-1} \equiv \mathrm{refl}$

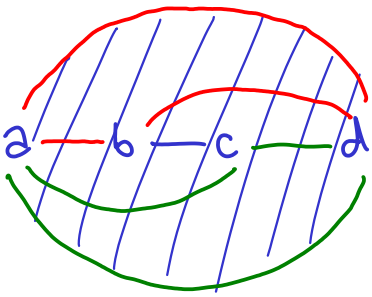
$p \cdot p^{-1} \equiv \mathrm{refl} \equiv p^{-1} \cdot p$

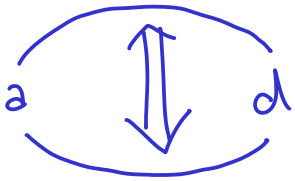
$\mathrm{refl} \cdot p \equiv p \equiv p \cdot \mathrm{refl}$

$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$

these are all equations, so now we have $p =_{a =_A b} q$

"higher identity type"





this structure continues to infinity

In category theory, there is a demand that these equalities hold "on the nose"; if we have this composition and that composition, we demand they be identical in the 1-category sense. This is where "weak" and "strong" comes from; equality comes from identifications, so the natural world treats them weakly, i.e. up to homotopy.

Functionality $\longrightarrow$ Functoriality

another way of saying this is to just move the $p : a =_A b$ quantifier to the conclusion.

Use elimination rule for identity type (J), which was developed on the basis of thinking of identity as an inductively defined type. Formally:

$$\frac{\begin{array}{l} x : A, y : A, z : x =_A y \vdash P(x,y,z) : \mathcal{U} \\ p : a =_A b \\ x : A \vdash p(x) : P(x,x,\mathrm{refl}) \end{array}}{J_{x,y,z.P}(x.p; q) : P(a,b,p)}$$

sufficient to show for refl

1) Motive
2) Path in question
3) "inductive step"/STS (sufficiency)
4) $\therefore$ motive is valid for q

$$J(x.p; \mathrm{refl}_A(a)) \equiv [a/x]p$$

PATH INDUCTION

We'll often say "Theorem, X (motive), and proof is by path induction on some path. We then (1) choose a motive and (2) do reflexivity. Ultimately we're writing a functional program that uses J. However we can still state it in "informal mathematics."

This is a notion of FUNCTORIALITY.

Functionality — everything respects def'l equivalence

eg) $a \equiv b$ implies $f(a) \equiv f(b) : B$ for any $f : A \to B$

Functions cannot distinguish between definitionally equal things. This makes sense, since we do not want f(2 + 2) != f(1 + 3)

eg) $a \equiv b$ implies $F(a) \equiv F(b) : \mathcal{U}$ for any $f : A \to \mathcal{U}$

Notice that this is just a restatement of the first statement, just with B = U.

But unfortunately, definitional equality doesn't give us as much as we want, for example, x + 0 and 0 + x are not definitionally equal, so these principles don't apply.

So functoriality needs to express that things respect HOMOTOPY.

Functoriality — everything respects homotopy

eg) $a =_A b$ implies $f(a) =_B f(b)$

This is a beefier statement than functionality; it tells us something about proofs. IF we have a proof a = b, then we have a proof f(a) = f(b).

eg) if $p : a =_A b$ then

$ap\ f\ b : f(a) =_B f(b)$

"path application"
"action on paths"

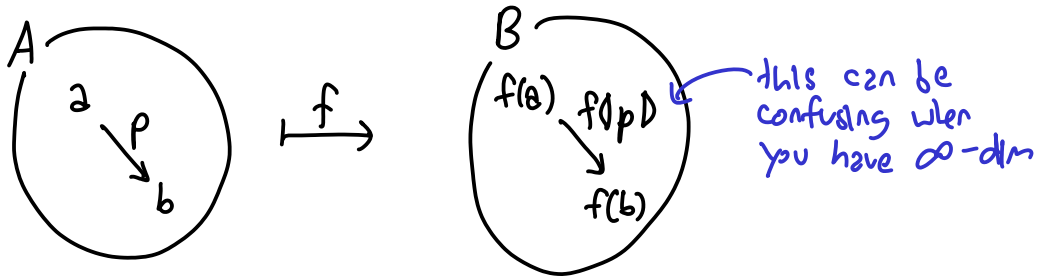
The term functor comes from category, where maps on categories must map both objects as well as morphisms, while preserving various coherence properties. The situation is analogous here:

normal 1-category theory

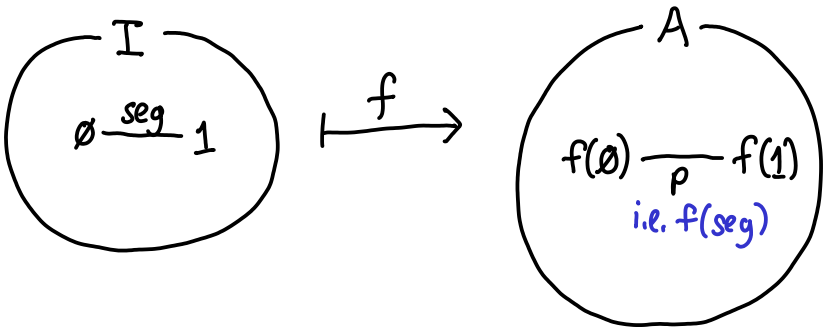
$f(a)$	0-cell (point) application	$ap_0\ f\ a$
$f(p)$	1-cell application	$ap_1\ f\ a$ (or simply ap)

But we can keep going on here, the analogy in CT being infinity-category theory, where all of the higher morphisms transfer too.

To sum up, a function is ALSO a functor which will transform paths.



aside: Unit Interval



Thus, I -> A allows you to PICK OUT a path from the space A. We know, in fact, that I -> A is equivalent to the total path space of A.

ASIDE: in homotopy theory, the interval would actually contain all of the real numbers. This kind of homotopy is called analytic, because it is built up from real numbers. We are taking a more functional programming approach; the analytic approach can be thought of as the "assembly language" of homotopy theory; we're manually doing the "memory management" of handling the real numbers. What I really care about is that I have reflexivity, that I have transitivity, etc. This is called the SYNTHETIC approach, we are doing synthetic homotopy theory. What's interesting is that we can do a lot of elementary (but nontrivial) homotopy theory, while never mentioning real numbers, or topological spaces, etc. It's all intrinsic in the type theory. Type theory captures an INTRINSIC view; we've abstracted away from instances of homotopy theory not limited to topological spaces, but other structures. Paths show up everywhere, and type theory is the abstract theory of that. It's good computer science, since identifying an abstract type, and identifying an induction principle; and as it turns out, mathematicians never noticed this. I like this method, because it is very natural from CS view.

Q/A: Think that the function from 1 -> A (where 1 is the discrete set of one point), this identifies a point in A. So we've generalized this to work with a path.

Functoriality is a THEOREM.

The reason path induction is valid, is everything that we can write down satisfies this requirement. Every function can be interpreted into functions on topological spaces, and there is no way to write a discontinous function. Intuitively, in freshman calculus, how was the example always defined? Well, you say a function is zero until one, and then it jumps up. There's a case analysis there. Well, from an FP perspective, how do you do that? There's no way to tell if a real number is greater than or equal to zero. So the idea is that to do something discontinous, you need to decide something (trichotomy); but this is precisely what constructive mathematics denies. Everything I write down is constructive math, so it will respect paths.

This is great: constructivity is motivated by computability. But it also is essential to dealing with mathematics of a very abstract nature, which has nothing to do with computability. Constructivity is at the core. Set theory has baked in too many wrong assumptions, you're at a dead end. Type theory lets us define these structures.

Theorem. If $f : A \to B$ and $p : a =_A b$, then $\mathrm{ap}\, f\, a : f(a) =_B f(b)$ (exists; $P(a,b)$)

Proof. By path induction on p

note: inductive hypothesis is the hardest part.

1) motive: $x : A, y : A \vdash f(x) =_B f(y) : \mathcal{U}$

2) STS, $x : A \vdash \mathrm{refl}_B(f(x)) : f(x) =_B f(x)$

3) $\therefore \mathrm{ap}\, f\, p : f(a) =_B f(b)$ ∎

(therefore)

Q: Why is f(x) = f(y) : U?
A: Well, closure properties of universes always allows us to form this type.

Q: Why doesn't p show up in the motive?
A: It could, but our conclusion doesn't say anything about the path itself. Our proof will, however!

by the way, we know

$$\mathrm{ap}\, f\, \mathrm{refl}_A(a) \equiv \mathrm{refl}_B(f(a))$$

due to J's calculation rule.

Path in space A can be transformed to path in space B, where the null path in A will turn into the null path in B. If you don't say this, for all we know, we could take self-loops in A into a loop which is not refl.

This is a simple example of proof relevant mathematics. We are thinking of them as operations on proof data structures.

Q: Why do we care that refl maps to refl? Is it functoriality?
A: Yes, essentially. You can in fact prove that ap acts functoriality in p, so it will preserve composition and inverses, and this proof requires that calculation rule. These proofs will be done by path induction, in which case I'll have reflexivity. And so I need to see that transporting these reflexivities will work.

Exercise: show $ap\ f\ p^{-1} = (ap\ f\ p)^{-1}$

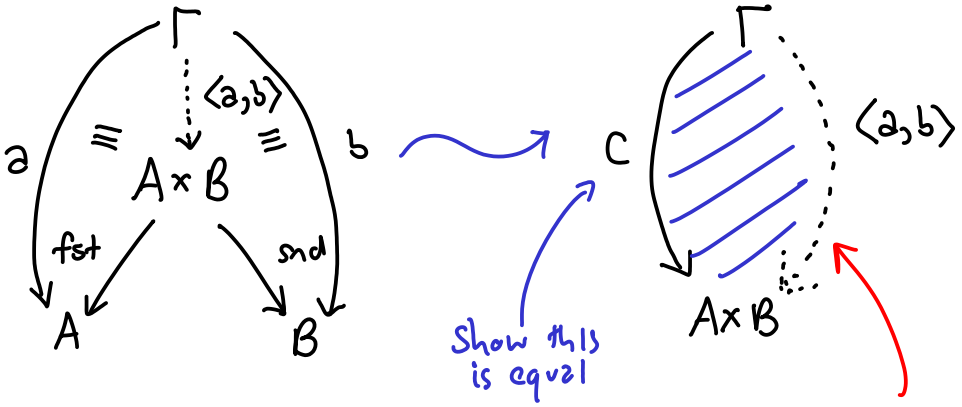
$ap\ f\ (p \cdot q) = ap\ f\ p \cdot ap\ f\ q$

neat facts)

1) if $a_1 =_A a_2$ & $b_1 =_B b_2$ then $\langle a_1, b_1 \rangle =_{A \times B} \langle a_2, b_2 \rangle$

p: q: ap (ap pair p) q: ↑ curried

2) recall: the question was unicity



Exercise: exhibit the cell given this axiom.

Axiom: $c =_{A \times B} \overbrace{\langle fst(c), snd(c) \rangle}^{\text{a proxy}}$ universal condition for products

Paolo: This "axiom" can actually be provable. But see below.

not definitional! c could be some unknown process.

In the HoTT book, products are defined one way, so this is an axiom, but then there is another theorem (pattern matching) which now must be an axiom.
In the HoTT book, products are treated positively, but I want to treat it negatively.

I am explaining the consequences of path induction, in the sense that everything respects paths. Here is another consequence:

Idea: If $F : A \to \mathcal{U}$ and $a =_A b$ then $F(a) \stackrel{?}{=} F(b)$

(This question gets to the heart of univalence.)

I would like to say that if a and b are equal, then F(a) and F(b) are in some sense the same type. E.g. because there is a proof x + y = y + x, then I'd like to know Vec(x + y) ?= Vec(y + x). Recall this is not true definitionally.

What does it mean for two types to be equivalent?

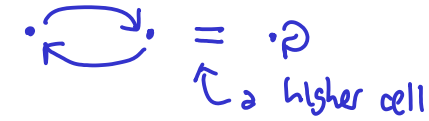
Q: I see why this is true, but what is the reason you can't just use the J rule to prove this, by fiddling with universe levels?
A: You're anticipating what I'm going to say.

Want $F(a) \simeq F(b)$

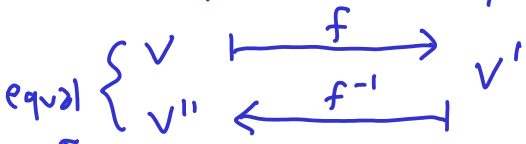
e.g. $\Sigma f : F(a) \to F(b).\ \mathrm{isequiv}(f)$

invertible up to higher homotopy

This equivalence means I can TRANSPORT a type from F(a) to F(b), and back again.



Vec(x+y) Vec(y+x)



not identical!

Q: Does this only work for index types?
A: This works for arbitrary types. Just make F the constant type.

this is not useful; the more path does not let us do anything.

Notice: if $a =_A b$, then $\boxed{\mathrm{ap}\ F\ p} : F(a) =_{\mathcal{U}} F(b)$

Problem: what is the relationship between paths in U, and equivalences? The equivalences let us move things from A to B, which has additional structure (a map backward).

Q: How is this different from isomorphisms in CT
A: This is equivalence in CT; e.g. an equivalence of categories.

Idea: turn paths into equivalences. So idtoequiv(ap F p) will actually let us do the mapping from one type to the other, and back.

index

If $p : a =_A b$ then $\underbrace{\text{transport}^F(p)}_{p_*} : F(a) \to F(b)$

s.t. $\text{transport}^F(\text{refl}) \equiv id$

The notation is not the best because you usually need to know F, but usually you know what it is.

define this with path induction

later I'll show it's an equivalence

1) Motive: $x : A, y : A \vdash F(x) \to F(y) : \mathcal{U}$

2) STS. $x : A \vdash \lambda u.\, u : F(x) \to F(y)$

3) $\therefore J(x.\lambda u.u^{F(x)}; p) : F(a) \to F(b)$

Ex) show 1) $\text{refl}_* = id$
2) $(p^{-1})_* = p_*^{-1}$
3) $(p \cdot q)_* = p_* \circ q_*$

} these don't typecheck

Moreover, transport is an equivalence.

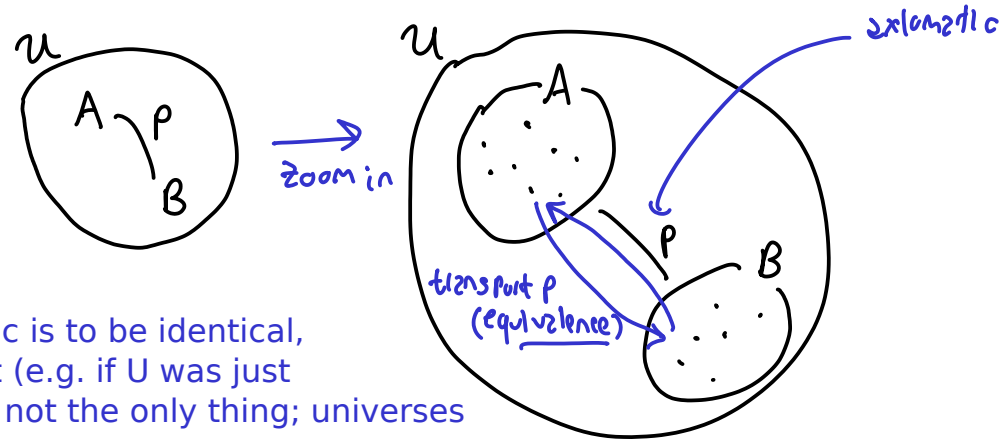
We have glossed over what it means for something to be an equivalence. It has a pre and a post inverse up to higher homotopy.

Principle of Univalence

What do we know?

$$A =_{\mathcal{U}} B \longrightarrow A \simeq B$$

via transport



Being homotopic is to be identical, if we were a set (e.g. if U was just Nat). But Nat is not the only thing; universes let us zoom in.

What does univalence say?

$$A =_{\mathcal{U}} B \quad \simeq \quad A \simeq B$$

Previously, there may have been a PAUCITY of paths in U, because the only constructor we gave you was refl. But univalence says, "Oh no, there is a boatload of paths in U, all you need is an equivalence." And if I had a path, when I turn it into an equivalence and back, it'll be the same type (up to higher homotopy). If I had a universe with homotopy propositions (HProp, these are things are empty or have one element), this would mean that interprovable propositions are equal. For HProps (known as -1-types), ~~interprovable HProps are equa~~l.

$$A =_{HProp} B \quad \simeq \quad A \Leftrightarrow B$$

this is extremely useful.

(generic programming)

Or, if we have homotopy sets (or 0-types, a space whose path structure is degenerate, since the only paths are self-loops), then we can say bijective sets are equal. Intuitively, there is no fine structure to elements; either it's the same or not. That's what sets are. Sets are a degenerate form of types.

Generic programming is all about the functoriality of data constructors.
And since HoTT is all about functoriality, you can make this all work.

The univalence axiom has lots of implications.

The interesting thing is we treat the proposition as a type, and we show that this type is equivalent to the space of equivalences. This is beautiful: it generalizes the conventional mathematical notion, because it speaks of the equivalence of proofs. From a CS point of view, this is where all the action is.

Claim: Agda is vastly better for HoTT then Coq.