Homotopical Patch Theory

Carlo Angiuli Edward Morehouse
Daniel R. Licata Robert Harper

Carnegie Mellon University

September 3, 2014

This talk is about doing (Darcs-like) patch theory inside of homotopy type theory using functorial semantics.

Functorial Semantics

Functorial Semantics

Define equational theories as functors out of some category.

A group is a product-preserving functor $\mathbb{G} \to \mathbf{Set}$.

This idea is due to Bill Lawvere—that equational theories (like group theory) are categories, and their models (like groups) are functors out of those categories. $\mathbb{G}$ defines what it means to be a group; a functor out of it *is* a group.

Functorial Semantics

The theory of groups, $\mathbb{G}$, is generated by:

$$C \in \mathrm{Ob}(\mathbb{G})$$

$$comp : C \times C \to C$$
$$e : 1 \to C$$
$$^{-1} : C \to C$$

such that $(\mathrm{id} \times e); comp = \mathrm{id}, \ldots$

$\mathbb{G}$ is a finite-product category generated by an object C, which represents the *carrier* of the group; and morphisms generated by the identity element of C, a binary composition operator, and a unary inverse operator; satisfying laws.

Functorial Semantics

A product-preserving functor $[\![-]\!] : \mathbb{G} \to \mathbf{Set}$ is

a set $[\![C]\!]$ (object part of $[\![-]\!]$)

a binary operation $[\![comp]\!]$ on $[\![C]\!]$
an element $[\![e]\!] \in [\![C]\!]$ (morphism part of $[\![-]\!]$)
a unary operation $[\![{}^{-1}]\!]$ on $[\![C]\!]$

such that $[\![comp]\!](g, [\![e]\!]) = g$ for all $g \in [\![C]\!]$, ... (that $[\![-]\!]$ respects equality of morphisms)

If we unpack this definition, the object part of the functor gives us a carrier set; the morphism part gives us the group operations; and those operations must satisfy the group laws because the morphisms in $\mathbb{G}$ did. For the purposes of this talk, we say the image is a *concrete implementation* in the sense that it is just sets and functions, so we can run it.

Functorial Semantics

A product-preserving functor $[\![-]\!] : \mathbb{G} \to \mathbf{Set}$ is

a set $[\![C]\!]$ (object part of $[\![-]\!]$)

a binary operation $[\![comp]\!]$ on $[\![C]\!]$
an element $[\![e]\!] \in [\![C]\!]$ (morphism part of $[\![-]\!]$)
a unary operation $[\![^{-1}]\!]$ on $[\![C]\!]$

such that $[\![comp]\!](g, [\![e]\!]) = g$ for all $g \in [\![C]\!], \ldots$ (that $[\![-]\!]$ respects equality of morphisms)

If we unpack this definition, the object part of the functor gives us a carrier set; the morphism part gives us the group operations; and those operations must satisfy the group laws because the morphisms in $\mathbb{G}$ did. For the purposes of this talk, we say the image is a *concrete implementation* in the sense that it is just sets and functions, so we can run it.

Functorial Semantics

A product-preserving functor $[\![-]\!] : \mathbb{G} \to \mathbf{Set}$ is

a set $[\![C]\!]$ (object part of $[\![-]\!]$)

a binary operation $[\![comp]\!]$ on $[\![C]\!]$
an element $[\![e]\!] \in [\![C]\!]$ (morphism part of $[\![-]\!]$)
a unary operation $[\![^{-1}]\!]$ on $[\![C]\!]$

such that $[\![comp]\!](g, [\![e]\!]) = g$ for all $g \in [\![C]\!], \ldots$ (that $[\![-]\!]$ respects equality of morphisms)

If we unpack this definition, the object part of the functor gives us a carrier set; the morphism part gives us the group operations; and those operations must satisfy the group laws because the morphisms in $\mathbb{G}$ did. For the purposes of this talk, we say the image is a *concrete implementation* in the sense that it is just sets and functions, so we can run it.

Functorial Semantics

A product-preserving functor $[\![-]\!] : \mathbb{G} \to$ **Set** is

a set $[\![C]\!]$ (object part of $[\![-]\!]$)

a binary operation $[\![comp]\!]$ on $[\![C]\!]$
an element $[\![e]\!] \in [\![C]\!]$ (morphism part of $[\![-]\!]$)
a unary operation $[\![^{-1}]\!]$ on $[\![C]\!]$

such that $[\![comp]\!](g, [\![e]\!]) = g$ for all $g \in [\![C]\!], \ldots$ (that $[\![-]\!]$ respects equality of morphisms)

If we unpack this definition, the object part of the functor gives us a carrier set; the morphism part gives us the group operations; and those operations must satisfy the group laws because the morphisms in $\mathbb{G}$ did. For the purposes of this talk, we say the image is a *concrete implementation* in the sense that it is just sets and functions, so we can run it.

Functorial Semantics

A product-preserving functor $[\![-]\!] : \mathbb{G} \to \textbf{Set}$ is

a set $[\![C]\!]$ (object part of $[\![-]\!]$)

a binary operation $[\![comp]\!]$ on $[\![C]\!]$
an element $[\![e]\!] \in [\![C]\!]$ (morphism part of $[\![-]\!]$)
a unary operation $[\![^{-1}]\!]$ on $[\![C]\!]$

such that $[\![comp]\!](g, [\![e]\!]) = g$ (that $[\![-]\!]$ respects
for all $g \in [\![C]\!]$, … equality of morphisms)

If we unpack this definition, the object part of the functor gives us a carrier set; the morphism part gives us the group operations; and those operations must satisfy the group laws because the morphisms in $\mathbb{G}$ did. For the purposes of this talk, we say the image is a *concrete implementation* in the sense that it is just sets and functions, so we can run it.

Homotopy Type Theory

Homotopy Type Theory

HoTT is a constructive, proof-relevant theory of equality inside dependent type theory.

Equality proofs $\mathtt{p} : \mathtt{a} =_{\mathtt{X}} \mathtt{b}$ are identifications of $\mathtt{a}$ with $\mathtt{b}$.

Everything I'll do from here on is in HoTT, an extension of dependent type theory (like Agda or Coq).

Homotopy Type Theory

$$
\begin{tikzcd}
\mathtt{a} \arrow[loop left, dashed, "\mathtt{refl}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathtt{a} \arrow[rr, bend left, "\mathtt{p}"] & & \mathtt{b} \arrow[ll, bend left, dashed, "\mathtt{!p}"]
\end{tikzcd}
$$

$$
\begin{tikzcd}
& \mathtt{b} \arrow[dr, bend left, "\mathtt{q}"] & \\
\mathtt{a} \arrow[ur, bend left, "\mathtt{p}"] \arrow[rr, bend right, dashed, "\mathtt{q} \circ \mathtt{p}"'] & & \mathtt{c}
\end{tikzcd}
$$

These identifications are reflexive, symmetric, and transitive, because equality is.

Homotopy Type Theory

We can have identifications of identifications.

$$\begin{tikzcd} \mathtt{a} \arrow[r, bend left, "\mathtt{p}"] \arrow[r, bend right, "\mathtt{q}"'] & \mathtt{b} \end{tikzcd}$$

$$\mathtt{p,q} : \mathtt{a} =_{\mathtt{X}} \mathtt{b}$$
$$\mathtt{a,b} : \mathtt{X}$$

Because the identifications are proof-relevant—they come with evidence—those identifications can themselves be identified. This leads to an infinite-dimensional tower of equalities. In “ordinary” types, these identifications represent exact equality, and are always reflexivity if they exist. In that case, where the equality types themselves are uninteresting, we call the type a *set*.

Homotopy Type Theory

We can have identifications of identifications.

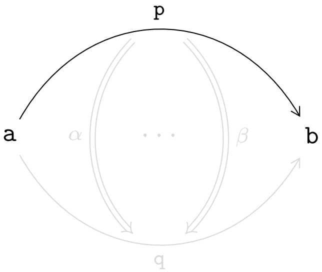
$$\begin{tikzcd}[column sep=huge] a \arrow[rr, bend left=70, "p"{name=P}] \arrow[rr, bend right=70, "q"'{name=Q}] & \cdots & b \arrow[from=P, to=Q, Rightarrow, bend right=40, "\alpha"'] \arrow[from=P, to=Q, Rightarrow, bend left=40, "\beta"] \end{tikzcd}$$

$$\begin{array}{lcl} & \vdots & \\ \alpha, \beta & : & p =_{(a =_X b)} q \\ p, q & : & a =_X b \\ a, b & : & X \end{array}$$

Because the identifications are proof-relevant—they come with evidence—those identifications can themselves be identified. This leads to an infinite-dimensional tower of equalities. In “ordinary” types, these identifications represent exact equality, and are always reflexivity if they exist. In that case, where the equality types themselves are uninteresting, we call the type a *set*.

Homotopy Type Theory

We can have identifications of identifications.



$$\vdots$$

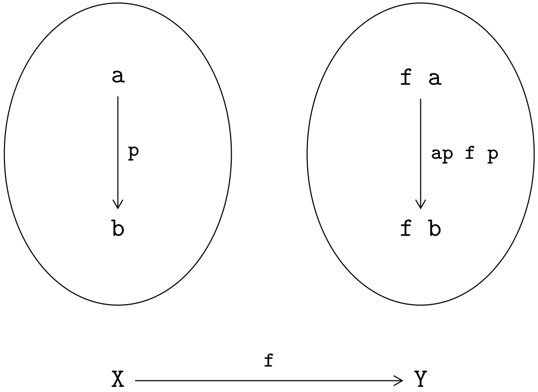
$$\alpha, \beta \;:\; \mathtt{p} =_{(\mathtt{a} =_{\mathtt{X}} \mathtt{b})} \mathtt{q}$$

$$\left.\begin{array}{l} \mathtt{p}, \mathtt{q} \;:\; \mathtt{a} =_{\mathtt{X}} \mathtt{b} \\ \mathtt{a}, \mathtt{b} \;:\; \mathtt{X} \end{array}\right\} \mathtt{X} \text{ is a set}$$

Because the identifications are proof-relevant—they come with evidence—those identifications can themselves be identified. This leads to an infinite-dimensional tower of equalities. In "ordinary" types, these identifications represent exact equality, and are always reflexivity if they exist. In that case, where the equality types themselves are uninteresting, we call the type a *set*.

Homotopy Type Theory

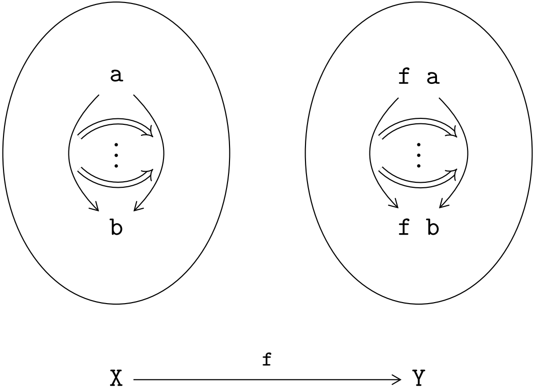
Functions preserve this structure.



As expected, functions send equal elements to equal results. In fact, functions preserve *all* this structure, at all levels.

Homotopy Type Theory

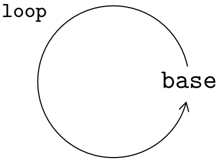
Functions preserve this structure.



As expected, functions send equal elements to equal results. In fact, functions preserve *all* this structure, at all levels.

Homotopy Type Theory

Higher Inductive Types introduce non-sets: arbitrary spaces.




```
space Circle : Type where
  base : Circle
  loop : base =Circle base
```

This is the first way we'll introduce interesting identifications into type theory; the other will come up in a bit. Note that `loop` just *generates* identifications; we also get `loop ∘ loop`, `!loop`, etc.

Patch Theory

Repositories and Changes

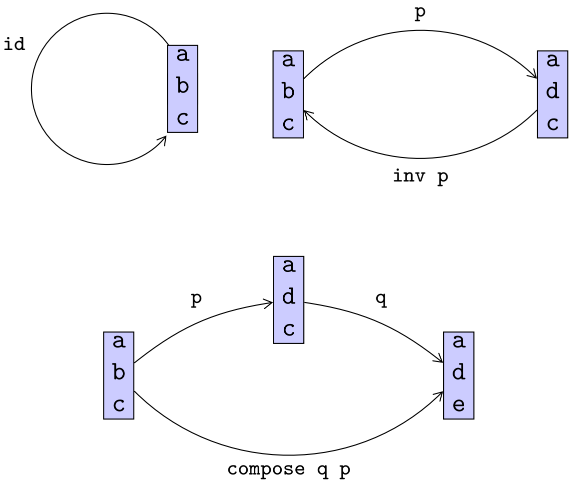
`Vec n String` (a repository)

`ADD s@l`
`RM l` (and changes to it)
`...`

such that
`RM l ○ ADD s@l = id...` (satisfying patch laws)

We want to study *repositories and patches*. For example, this is a concrete implementation of a (one-file) repository and changes one might apply to it.

Repositories and Changes



We want to study the general phenomenon of repositories and changes (similar to how group theory was invented to generalize symmetry groups). What sorts of things should be true of all patch theories? There are identity patches at every repository, and patches are invertible and composable.

Repositories and Changes

Some laws hold in all patch theories. (`compose id p = p`)

Patches aren't always applicable. (`RM 5` in a 3-line file)

Patch Theory

`num` (an abstract repository)

`add1`
`id`, `compose`, `inv...` (with abstract patches)

such that
`compose id p = p ...` (satisfying patch laws)

Here's an abstract theory of a repository. The idea is that the repository contains a single number, and the only patches add to (or subtract from, thanks to inverses) that number.

Patch Theory

Abstract patches as a HIT:

```
space Patch : Type where
  add1 : Patch
  id : Patch
  compose : Patch → Patch → Patch
  inv : Patch → Patch

  unitl : compose id p =Patch p
              ⋮
```

We can model these patches as a HIT. The patches are `add1`, identity, and compositions and inverses of these; and we identify certain compositions by the groupoid laws (for example, identity is a left unit for composition).

Patch Theory

Interpret these patches functorially:

```
interp : Patch → (Int → Int)

interp add1 = λn.n+1
interp id = λn.n
interp (compose p2 p1) = λn.interp p2 (interp p1 n)
```

$\texttt{ap interp unitl : interp (compose id p)} =_{\texttt{Int}\to\texttt{Int}} \texttt{interp p}$

$$\vdots$$

If we interpret `num` as the type `Int`, then we interpret patches as concretely effecting changes on `Int`s, in a functorial way.

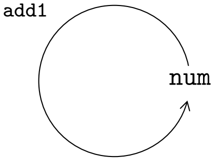
Patches as Identifications

In HoTT, equality is groupoidal and respected functorially!

Key idea: a patch taking `a` to `b` is an identification of `a` and `b`.

But the existence of identities, compositions, and inverses, and preservation thereof by functions, is already guaranteed in HoTT for identifications! We can take advantage of this by modeling patches as identifications.

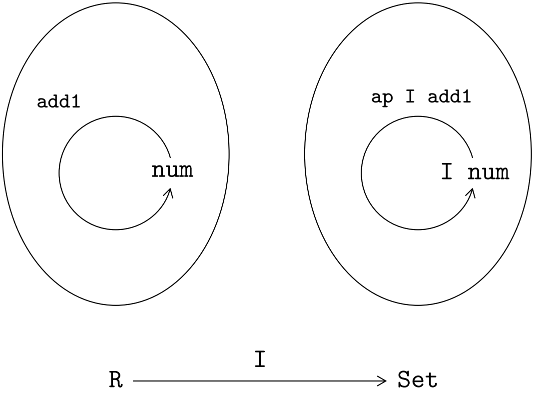
Patches as Identifications




```
space R : Type where
  num : R
  add1 : num =R num
```

Thus, we say the type of patches is `num = num`, which gives us the groupoid operations and laws, and functoriality, for free! In this case, the patch theory `R` looks just like the circle. (Recall that the `add1` constructor generates additional identifications.)

Interpreting Patch Theory



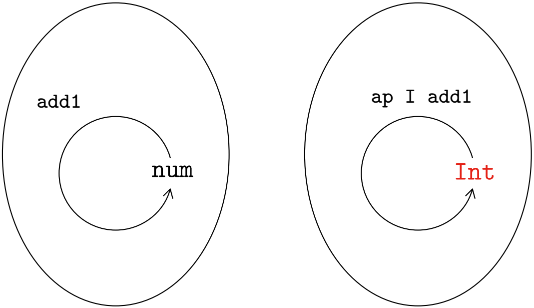
$\texttt{I : R} \to \texttt{Set}$

$\texttt{I num}$
$\quad \texttt{: Set}$

$\texttt{ap I add1}$
$\quad \texttt{: I num} =_{\texttt{Set}} \texttt{I num}$

To use that built-in functoriality, if we interpret patches as identifications, then `add1` is an identification between `Int` and itself. How might we get one of those?

Interpreting Patch Theory



$$\mathtt{R} \xrightarrow{\mathtt{I}} \mathtt{Set}$$

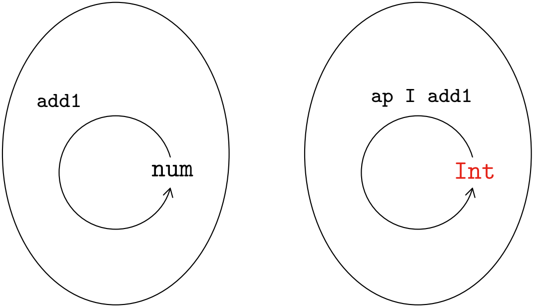
$\mathtt{I : R \to Set}$

$\mathtt{I\ num = Int}$
$\mathtt{\ : Set}$

$\mathtt{ap\ I\ add1}$
$\mathtt{\ : I\ num =_{Set} I\ num}$

To use that built-in functoriality, if we interpret patches as identifications, then `add1` is an identification between `Int` and itself. How might we get one of those?

Interpreting Patch Theory



$$\mathtt{R} \xrightarrow{\mathtt{I}} \mathtt{Set}$$

`I : R → Set`

`I num = Int`
` : Set`

`ap I add1`
` : Int =`$_{\mathtt{Set}}$` Int`

To use that built-in functoriality, if we interpret patches as identifications, then `add1` is an identification between `Int` and itself. How might we get one of those?

Univalence Axiom

Bijections between sets `X` and `Y` yield identifications $\texttt{X} =_{\texttt{Set}} \texttt{Y}$.

$$\texttt{ua : Bijection X Y} \to \texttt{X} =_{\texttt{Set}} \texttt{Y}$$

The second way we add new identifications into type theory is by the univalence axiom. Remember, equality is proof-relevant. We're not saying isomorphic types are the *same*; we're saying that we identify them via their isomorphism.

Univalence Axiom

Bijections between sets $\mathtt{X}$ and $\mathtt{Y}$ yield identifications $\mathtt{X} =_{\mathtt{Set}} \mathtt{Y}$.

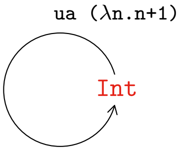
$$\mathtt{ua} : \mathtt{Bijection}\ \mathtt{X}\ \mathtt{Y} \to \mathtt{X} =_{\mathtt{Set}} \mathtt{Y}$$

In particular,

$$\mathtt{ua}\ (\lambda\mathtt{n.n+1}) : \mathtt{Int} =_{\mathtt{Set}} \mathtt{Int}$$

The second way we add new identifications into type theory is by the univalence axiom. Remember, equality is proof-relevant. We're not saying isomorphic types are the *same*; we're saying that we identify them via their isomorphism.

Interpreting Patch Theory



`I num = Int` (I on elements)

`ap I add1 = ua (λn.n+1)` (I resp. equality)

`ap (ap I) unitl :`
`  ap I (refl ∘ p)` $=_{\texttt{Int}=_{\texttt{Set}}\texttt{Int}}$ `ap I p` (I resp. equality of equalities)

Then the objects/elements part of I determines the way we interpret the abstract repository; I's respect for equality determines the way we interpret patches; and I's respect for equalities of equalities ensures that the interpretation of patches satisfies the patch laws (here, just the groupoid laws).

Interpreting Patch Theory

$$\begin{tikzcd} \mathtt{Int} \arrow[loop, "\mathtt{ua}\ (\lambda \mathtt{n.n+1})"] \end{tikzcd}$$

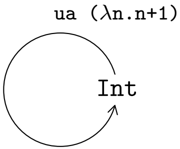
$\mathtt{I\ num = Int}$ (I on elements)

$\mathtt{ap\ I\ add1 = ua\ (}\lambda\mathtt{n.n+1)}$ (I resp. equality)

$\mathtt{ap\ (ap\ I)\ unitl}$:
$\quad \mathtt{ap\ I\ (refl} \circ \mathtt{p)} =_{\mathtt{Int} =_{\mathtt{Set}} \mathtt{Int}} \mathtt{ap\ I\ p}$ (I resp. equality of equalities)

Then the objects/elements part of I determines the way we interpret the abstract repository; I's respect for equality determines the way we interpret patches; and I's respect for equalities of equalities ensures that the interpretation of patches satisfies the patch laws (here, just the groupoid laws).

Interpreting Patch Theory




```
I num = Int
```
(I on elements)

```
ap I add1 = ua (λn.n+1)
```
(I resp. equality)

ap (ap I) unitl :
  ap I (refl $\circ$ p) $=_{\mathtt{Int} =_{\mathtt{Set}} \mathtt{Int}}$ ap I p

(I resp. equality of equalities)

Then the objects/elements part of I determines the way we interpret the abstract repository; I's respect for equality determines the way we interpret patches; and I's respect for equalities of equalities ensures that the interpretation of patches satisfies the patch laws (here, just the groupoid laws).

Now It Gets Tricky…

A Different Repository

`Nat` (a repository)

generated by λ`n.n+1` (and changes to it)

such that… (satisfying patch laws)

Let's change the previous example a bit—how do we abstractly model the situation where what if the repository is a natural number, with a patch to increment it?

A Different Patch Theory

`num` (an abstract repository)

`add1`
`id`, `compose`, `inv`... (with abstract patches)

such that... (satisfying patch laws)

The obvious solution is to do the same thing as before, but the problem is that this will give us inverse patches like `!add1`...

A Different Patch Theory

But inverses don't exist in general:

$$
\begin{tikzcd}
?? & \mathtt{0} \arrow[l, mapsto, bend left, "\lambda\mathtt{n.n-1}"] \arrow[r, mapsto, bend left, "\lambda\mathtt{n.n+1}"] & \mathtt{1} \arrow[l, mapsto, bend left, "\lambda\mathtt{n.n-1}"] \arrow[r, mapsto, bend left, "\lambda\mathtt{n.n+1}"] & \mathtt{2} \arrow[l, mapsto, bend left, "\lambda\mathtt{n.n-1}"] & \cdots
\end{tikzcd}
$$

. . . and this doesn't actually work on all `Nats`! (By the way, this is one reason we would like HoTT without inverses, which we call *directed type theory*.)

A Different Patch Theory

Index the contexts to characterize patch applicability:

$$\begin{tikzcd}
\texttt{doc 0} \arrow[r, bend left, "\texttt{add1 0}"] & \texttt{doc 1} \arrow[r, bend left, "\texttt{add1 1}"] & \texttt{doc 2} \cdots
\end{tikzcd}$$

```
space I* : Type where
  doc : Nat → I*
  add1 : (n : Nat) → doc n =I* doc n+1
```

We can't help but have inverses, so the solution is to make sure that the inverses only exist in situations where they are actually possible. Indexing the contexts makes this possible by essentially giving "types" to the patches.

A Different Patch Theory

Index the contexts to characterize patch applicability:

$$
\begin{tikzcd}[column sep=huge]
\texttt{doc 0} \arrow[r, bend left, "\texttt{add1 0}"] & \texttt{doc 1} \arrow[l, bend left, "\texttt{!(add1 1)}"] \arrow[r, bend left, "\texttt{add1 1}"] & \texttt{doc 2} \arrow[l, bend left, "\texttt{!(add1 0)}"] \quad \cdots
\end{tikzcd}
$$

$$
\begin{array}{l}
\texttt{space I* : Type where} \\
\quad \texttt{doc : Nat} \to \texttt{I*} \\
\quad \texttt{add1 : (n : Nat)} \to \texttt{doc n} =_{\texttt{I*}} \texttt{doc n+1}
\end{array}
$$

We can't help but have inverses, so the solution is to make sure that the inverses only exist in situations where they are actually possible. Indexing the contexts makes this possible by essentially giving "types" to the patches.

Interpretation

How do we interpret this? Obvious idea:

$$\begin{tikzcd}
\texttt{I*} \arrow[d, "\texttt{I}"'] & \texttt{doc 0} \arrow[r, "\texttt{add1 0}"] & \texttt{doc 1} \arrow[r, "\texttt{add1 1}"] & \texttt{doc 2} & \cdots \\
\texttt{Set} & \texttt{Nat} \arrow[r, "\texttt{ua }(\lambda\texttt{n.n+1})"] & \texttt{Nat} \arrow[r, "\texttt{ua }(\lambda\texttt{n.n+1})"] & \texttt{Nat} & \cdots
\end{tikzcd}$$

But λ`n.n+1` isn't a `Bijection Nat Nat`.

Now let's build the interpretation. The obvious thing to do is to send each `doc n` to `Nat`, but this doesn't work because λ`n.n+1` isn't a bijection between `Nat` and itself! (Indeed, it isn't invertible, which was the problem in the first place.)

Interpretation

Fix: interpret `doc n` as the singleton type of `n`.

$$
\begin{tikzcd}
\texttt{I*} \arrow[d, "\texttt{I}"'] & \texttt{doc 0} \arrow[r, "\texttt{add1 0}"] & \texttt{doc 1} \arrow[r, "\texttt{add1 1}"] & \texttt{doc 2} & \cdots \\
\texttt{Set} & \texttt{S(0)} \arrow[r, "\texttt{ua ...}"] & \texttt{S(1)} \arrow[r, "\texttt{ua ...}"] & \texttt{S(2)} & \cdots
\end{tikzcd}
$$

where `S(n) = Σm:Nat.m=n`

`S(n)` is essentially the type of numbers equal to `n`. (Technically, it is any number, with a proof it is equal to `n`.) λ`n.n+1` is a bijection when restricted to a singleton, as is any map.

What Else?

What Else?

- More on interpreting non-invertible patches.
- Fancier patch theories, with fancier patch laws.
- Defining patch optimizers.
- Defining merging.

Expanded version of paper with more exposition:
`http://tinyurl.com/icfp-htpt`

We recommend that you read the expanded version of the paper, available on the authors' websites (and at this link), which has an addendum with some additional exposition.

Computation vs. Homotopy

There's a tension between:

equating terms by identifications	$\texttt{doc 0} =_{\texttt{I*}} \texttt{doc 1}$
distinguishing them by computations	$\texttt{doc 0} \mapsto \texttt{S(0)}$ $\texttt{doc 1} \mapsto \texttt{S(1)}$

The last point I'd like to bring up is that these additional identifications seem counter to the idea of computation, in the sense that we still wish to tell apart the different repositories.

Computation vs. Homotopy

Analogy: function extensionality already equates bubble sort and quicksort.

They are the same function but different programs.

Computation is finer-grained than equality.

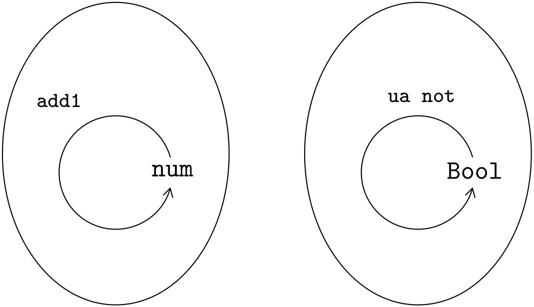
When we have function extensionality, we equate, for example, bubble sort and quicksort as functions, but they compute very differently on the same list. As *logicians* we want to equate the functions, but as *computer scientists* we want to distinguish the programs. Indeed, there's already a trend (OTT, internalizing parametricity, etc.) of extending the syntax of type theory with additional semantic equations.

Thanks!

Any questions?

Interpreting Patch Theory

There are other ways to interpret `R`.




```
I num = Bool
ap I add1 = ua not
```

$$\mathtt{R} \xrightarrow{\mathtt{I}} \mathtt{Set}$$

Interpreting Patch Theory

The `Bool` interpretation satisfies additional laws.
(`ap I (add1` $\circ$ `add1) =` λ`n.n`)

`Int` is the complete interpretation, because

The fundamental group of the circle is `Int`.

In a sense, this means that the `Int` interpretation doesn't validate any extra laws: it's the free model of the theory.