University of Bergen
Department of informatics

A Very Good Title

Åsmund Aqissiaq Arild Kløvstad
Supervised by Håkon Robbestad Gylterud

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

May, 2022

Abstract

In this thesis we consider theoretical models of version control systems based on Homotopy Type Theory (HoTT). The main contribution is an implementation of Angiuli et al.'s Homotopical Patch Theory [1] in Cubical Agda.

Additionally we survey other approaches including Darcs' "patch algebra" [AND CATEGORICAL?], and (unsuccessfully) propose an approach based on type-theoretic coequalizers.

Finally, the first chapter contains an approachable introduction to HoTT and (Cubical) Agda aimed at an audience of interested computer science students.

Acknowledgements

Åsmund Aqissiaq Arild Kløvstad
Monday 23rd May, 2022

Contents

List of Figures

Chapter 1

Introduction

1. version control
2. (homotopy) type theory
3. formalization + "computational content"?
4. related work?

Chapter 2 contains an introduction to the Homotopy Type Theory (HoTT) setting, a cubical interpretation of this type theory, and an introduction to the syntax and workings of the Agda programming language [21] in general and Cubical Agda [25] in particular.

Chapter 3 gives and exposition of Version Control Systems (VCS) and some approaches to a theory of such systems. We give an account of Darcs' [5] "Algebra of Patches" as described by Lynagh [11], an approach based on category theory by Mimram and Di Giusto [14], and Angiuli et al.'s "Homotopical Patch Theory" [1] which utilizes HoTT. Finally we present another HoTT approach attempted by the author, which has not proven fruitful.

Chapter 4 describes the main contribution of this thesis: an implementation of Homotopical Patch Theory in Cubical Agda. Following Angiuli et al. we implement three patch theories of increasing complexity. In section 4.4 we examine the implementations by testing them on simple examples. We conclude that the theories and models behave as expected, but are severely limited by the current limitations of Cubical Agda – in particular that `transp` and `hcomp` do not compute over indexed families.

Chapter 2

Homotopy Type Theory

In this chapter we introduce the basics of Homotopy Type Theory (HoTT). The purpose is to give the reader the prerequisites to follow the formalization in chapter 4. For an excellent in-depth introduction see Egbert Rijke's 2019 summer school notes [19]. The canonical text book is The Book [24].

We start by giving an intuitive introduction to dependent types and their notation (in terms of inference rules). Then we consider two important interpretations: types as propositions and types as spaces. We then move on to inductive types and higher inductive types (HITS), before an introduction to the syntax of the dependently typed language and proof assistant Agda.

Finally we look at a cubical interpretation of HoTT and Cubical Agda, an implementation of it in Agda.

2.1 (Dependent) Type Theory

Types are a familiar concept to the computer scientist. We are used to working with data, and this data often has a *data type* either explicitly or implicitly. For example, `42` is an `int`, `'c'` is a `char`, and `['a','b','c']` is a list of `char`s (henceforth denoted `[char]`). We call `int`, `char` and `[char]` *types* and `42`, `'c'`, `['a','b','c']` *terms* of those types. While this is a good basis for intuition, the Type Theory we consider is a bit different.

However, let us stick with the programming intuition to introduce a less familiar concept: *dependent* types. First, note that one of the types in the previous paragraph is a bit different from the others: ['a','b','c'] is a list *of* ***chars***. Similarly we could have lists of `int`s, lists of `float`s or even lists of lists! Clearly "lists" comprises many different types, depending on the type of their elements. We could call `list` a family of types *parametrized* by types. Such a family is actually a whole collection of types – one for each other type we can make lists of. Dependent types extend this idea by allowing families to be parametrized by terms. Then we can create new and exciting types like `Vec 3` and `Vec 4` – the types of 3- and 4-dimensional vectors. Again `Vec` is actually a whole collection of types – one for each integer.

We can think of `Vec` as a function that assigns a type to each integer, and may refer to it as "a (type) family over `Int`".

We now leave the familiar world of programming behind and venture in to the spooky (but exciting) world of foundational mathematics.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f(a) : B} \tag{2.1}$$

In this new and wondrous world, a type theory is a system of *inference rules* like 2.1 that can be used to make *derivations*.

This particular inference rule is the elimination rule for function types. It says that if a is a term of type A and f is a function from A to B, then $f(a)$ is a term of type B. Let us take it apart.

The part above the line is a list of hypotheses, and the part below is the conclusion.

Each piece of the rule is called a *judgement*. They consist of a context, some expression and a $\vdash$ separating the two. In this example our judgements are:

$$\Gamma \vdash a : A$$

"In any context Γ, a is a term of type A"

$$\Gamma \vdash f : A \to B$$

"In any context Γ, f is a function from A to B"

$$\Gamma \vdash f(a) : B$$

"In any context Γ, $f(a)$ is a term of type B"

In fact these are all the same kind of judgement: a particular term (resp. $a, f, f(a)$) is of a particular type (resp. $A, A \to B, B$). There are three other kinds of judgements permitted in (Martin-Löf) type theory:

$$\Gamma \vdash A\ Type$$

"A is a type."

$$\Gamma \vdash a \equiv b : A$$

"a and b are judgementally equal terms of type A."

$$\Gamma \vdash A \equiv B\ Type$$

"A and B are judgementally equal types."

The judgement form $\Gamma \vdash A\ Type$ lets us formally define lists and vectors. Lists are easy:

$$\frac{\Gamma \vdash A\ Type}{\Gamma \vdash [A]\ Type}$$

This rule says "if A is a type, then lists of A is a type". Using $\mathbb{N}$ for the type of natural numbers, vectors are very similar:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash Vec(n)\ Type}$$

The preceding introductions of lists and vectors are clearly not complete specifications of the types. They do not tell us how to create new terms, nor how to use those terms in other expressions. In order to give a complete description we will need need more rules. This pattern and terminology will be used to introduce new types, so we elucidate it with a well-known example: the type of (non-dependent) functions.

[NOT SURE ABOUT THIS BIT, MAYBE EXPLAIN WITH Π AND/OR Σ?]

$$\frac{\Gamma \vdash A\ Type \qquad \Gamma \vdash B\ Type}{\Gamma \vdash A \to B\ Type} \tag{2.2}$$

An *introduction rule* (2.2) tells us how to construct the type. In this case, if A and B are types, then functions between them is also a type.

$$\frac{\Gamma, a : A \vdash f(a) : B}{\Gamma \vdash \lambda x.f(x) : A \to B} \tag{2.3}$$

A *formation rule* (2.3) tells us how to construct a *term* of the type. In the case of functions, terms are constructed by lambda abstraction – if for each $a : A$ we have term $b : B$, we can make a function that maps a to b. The result is denoted $f(a)$ to emphasize its dependence on a.

$$\frac{\Gamma \vdash f : A \to B}{\Gamma, a : A \vdash f(a) : B} \tag{2.4}$$

An *elimination rule* (2.4) describes how a term is used. In the case of functions, we may evaluate them with an argument in the domain to obtain a term in the codomain.

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) \equiv f : A \to B} \tag{2.5}$$

$$\frac{\Gamma, a : A \vdash f(a) : B}{\Gamma, a : A \vdash (\lambda y.f(y))(a) \equiv f(a) : B} \tag{2.6}$$

Computation rules postulate when two terms are judgementally equal. In the case of functions we have two: η-reduction (2.5) and β-reduction (2.6). Taken together, they imply that function evaluation and lambda abstraction are mutual inverses [19].

Finally, we consider two important families of dependent types: Σ-types (sometimes called "dependent pairs") and Π-types ("dependent functions"). Intuitively, Σ-types consist of pairs (x, y) where the type of y is allowed to depend on x, and terms of Π-types are functions $\lambda x.y$ where the type of y may depend on x. If the type of y happens to be constant, $\Sigma_A B$ is the product type $A \times B$ and $\Pi_A B$ is the type of non-dependent functions $A \to B$.

$$\frac{\Gamma \vdash A\ Type \qquad \Gamma, x : A \vdash B(x)\ Type}{\Gamma \vdash \Sigma_A B\ Type} \qquad \frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B(x)}{\Gamma \vdash (x, y) : \Sigma_A B}$$

The introduction and formation rules tell use that:

1. if A is a type, and B is a type family over A, then we can make the type $\Sigma_A B$ of dependent pairs
2. if we have a term x of type A and a term y of $B(x)$ we can create a term (x, y) of type $\Sigma_A B$

Additionally we have elimination rules (where π_1, π_2 denote the first and second projection):

$$\frac{\Gamma \vdash x : \Sigma_A B}{\Gamma \vdash \pi_1(x) : A} \qquad \frac{\Gamma \vdash x : \Sigma_A B}{\Gamma \vdash \pi_2(x) : B(\pi_1(x))}$$

[COMPUTATION RULES? THEY'RE PRETTY BORING]

The analogous rules for dependent functions are:

$$\frac{\Gamma \vdash A\ Type \qquad \Gamma, x : A \vdash B(x)\ Type}{\Gamma \vdash \Pi_A B\ Type} \qquad \frac{\Gamma, a : A \vdash b(a) : B(a)}{\Gamma \vdash \lambda x.b(x) : \Pi_A B} \qquad \frac{\Gamma \vdash f : \Pi_A B}{\Gamma, x : A \vdash f(x) : B(x)}$$

2.2 Propositions as Types

In this section we consider an important interpretation of type theory: the Howard-Curry Isomorphism (which isn't an isomorphism, but we're not going into those details).

Under this "isomorphism" types are identified with logical propositions, and terms with proofs of those propositions. This means we can consider a proposition "true" (or at least "proved") if we can construct a term of the corresponding type.

Two very simple types are the empty type $\bot$ which has not terms, and the unit type $\top$ which has one term denoted by $\mathbf{1}$. [MAYBE INTRODUCE THE TYPES FIRST]

Under the "types as propositions" interpretation, $\bot$ represents *false*. The type has no terms so there are no proofs of "false", just like we would expect from a sound system. (Of course this alone does not prove our type theory sound.) Similarly, $\top$ represents *true*. It always has a proof: $\mathbf{1}$.

Let us make some more elaborate propositions. For example given the types (and hence propositions) A and B what would it mean to prove $A \wedge B$? Well if both A and B are true, we should be able to give a proof of A *and* and proof of B. But since proofs are terms of the corresponding type, this is the same as having terms $a : A$ and $b : B$. To keep track of both, lets form the ordered pair (a, b). This is precisely an element of the product type $A \times B$! Hence this product type represents the proposition $A \wedge B$, since its terms correspond exactly to proofs of A and B.

As a sanity check, consider the truth table of $A \wedge B$ (2.1a) alongside the terms of $A \times B$ (2.1b) using $\top$ and $\bot$ to represent true and false. $A \wedge B$ is true when both A and B are true, and similarly $A \times B$ is inhabited exactly when both A and B are inhabited.

As another example, what does it mean to prove an implication $A \to B$? One reasonable answer is that given a proof of A, I can produce a proof of B. In terms of types, that means a way to produce a term of type B given a term of type A, which is exactly a function from A to B! Finally, note that logical "or" is represented by the sum type (disjoint union) $A + B$.

[NOTE: this results in a *constructive* logic (good)]

A	B	$A \wedge B$
false	false	false
false	true	false
true	false	false
true	true	true

(a) logic

A	B	$A \times B$
$\bot$	$\bot$	()
$\bot$	$\top$	()
$\top$	$\bot$	()
$\top$	$\top$	$(\mathbf{1}, \mathbf{1})$

(b) types "()" meaning there are no terms of this type

We have the basic building blocks of propositional logic, but what about first-order logic with $\exists$ and $\forall$? This is where our dependent types come in handy.

First, let us note that a predicate on a variable is a lot like a dependent type. If simple types can be interpreted as propositions, and a predicate on some variable is a proposition that *depends* on a variable, then it stands to reason that a predicate can be represented with a dependent type. As such, we may view a term of the type $B(x)$ as a proof that B holds for the term x. [WHATEVER THAT MEANS, LOL]

Extending this thinking to quantifiers and considering what it means to provide a proof, a proof of $\exists x.P(x)$ should consist of some $x : A$ [CHEATING IN THE A] and a proof that P is true of x. Such a pair is a term of a type we have seen before: the dependent pair $\Sigma_A P$. Note that this term actually contains *more* data than just asserting $\exists x.P(x)$ – it gives us an x.

Similarly, a proof of $\forall x.P(x)$ can be seen as an assertion that whatever $x : A$ you give me, I can produce a proof (term) of $P(x)$. This is exactly a function from $x : A$ to $P(x)$ so we use $\Pi_A P$ to represent this quantification.

Note that both of these constructions quantify over some base type A, so "for all x" necessarily becomes "for all x of type A".

2.3 Identity Types

Given this notion of propositions as types, one of the things we may want to propose (and prove) is the equality of two terms. That is, given two terms of some type, how do we show that they are equal? Note that this is different from the *judgemental* equality discussed in section 2.1. [HOW EXACTLY?]

Since propositions are types and "x is equal to y" is a proposition, there should be a corresponding type. Also, the truth of this proposition depends on x and y (clearly "2 is equal to 2" should be different from "2 is equal to 3") so the type should depend on x and y as well. But how should this type be constructed? What are the terms of such a type?

The solution, proposed by Per Martin-Löf [13], is an inductive family of dependent types called the *identity type*. For each type A and pair of terms $x, y : A$ we construct the identity type $x =_A y$ (the subscript may be dropped when the type of x and y is clear). It has the following formation and introduction rules [19]:

$$\frac{\Gamma \vdash a : A}{\Gamma, x : A \vdash a =_A x \; Type} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash refl_a : a =_A a}$$

and an induction principle given by:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, p : a =_A x \vdash P(x,p) Type}{\Gamma \vdash J_a : P(a, refl_a) \to \Pi_{x:A} \Pi_{p:a=_A x} P(x,p)}$$

This is astonishingly simple! The identity type has one constructor: $refl\_$, and in order to use its terms $p : x =_A y$ it is enough to know the case when $x \equiv y$ and $p \equiv refl$.

Despite the few ingredients, identity types exhibit a great deal of (admittedly expected) structure. For example, the identity type $=_A$ on some type A is an equivalence relation. [TYPE THEORETICALLY? WHAT DO WE CALL THIS?] It is clearly reflexive ($x =_A x$ is inhabited by $refl_x$), but it is also symmetric and transitive. Given proofs $p : x = y$ and $q : y = z$, let us denote the symmetric proof by $p^{-1} : y = x$ and the result of transitivity $p \cdot q : x = z$.

Given a term $a : A$ the J-rule lets us inhabit a type $P(x, p)$ by providing a term of $P(a, refl_a)$. This reduces the task of showing symmetry and transitivity to the cases when the paths are all $refl$. The inverse $refl^{-1}$ is again $refl$ and the composition $refl \cdot refl$ is also $refl$.

UIP and The Groupoid Structure of Types

A question one might ask is "can there be more than one proof of identity?". The affirmative answer is a property known as *Uniqueness of Identity Proofs* (UIP). A type A satisfies UIP if for any $x\ y : A$ and $p\ q : x =_A y$ the type $p =_{x=y} q$ is inhabited[1]. Now the question of uniqueness can be posed as "does UIP hold for every type?" In 1995 Hofman and Streicher [6] showed that the answer is "no" by constructing a model in which it fails to hold.

What, then, are the relationships between these proofs? Hofman and Streicher's model give and answer here too. In it, types are *groupoids* and the identity type $x =_A y$ is modeled by $Hom_A(x, y)$. In addition to the properties we have already seen, they show that composition is associative and respects units and inverses. That is, for proofs $p : x = y$, $q : y = z$, $r : z = w$ the following types are all inhabited:

$$(p \cdot q) \cdot r =_{x=w} p \cdot (q \cdot r)$$

$$p \cdot refl_y =_{x=y} p$$

$$refl_x \cdot p_{x=y} = p$$

$$p \cdot p^{-1} =_{x=x} refl_x$$

$$p^{-1} \cdot p =_{y=y} refl_y$$

This *groupoid structure* of proofs will be very important in chapter 4.

[1] In HoTT we say A *is a set*.

2.4 Spaces as Types

Another (related) way to make sense of identity types is through homotopy theory. With this interpretation pioneered by Voevodsky [26] and the HoTT program [24], a term of $x =_A y$ is like a path in A from x to y. In fact the collection of all such paths is itself a space (and thus a type): the path space. Additionally there may be paths between paths, paths between paths between paths and so on. These higher paths are the eponymous "homotopies" and provide a rich field of study on their own. Geometrically we visualize them as "filling in" the space between paths.

[MORE? FUNDAMENTAL GROUPOIDS?]

2.5 Higher Inductive Types

2.5.1 Inductive Types

One way to construct more elaborate types is by induction. An inductive type is defined by a number of constructors, which can be either constant terms or functions. Let us return to the type of lists. It can be constructed from the empty list and the function `cons` which takes an element and affixes it to the start of a list. Using [] for the empty list and :: for the (infix) cons function we have a pair of introduction rules:

$$\frac{\Gamma \vdash A\ Type}{\Gamma \vdash [] : [A]} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash as : [A]}{\Gamma \vdash (a :: as) : [A]}$$

From these we can construct arbitrarily long lists by starting with the empty list and affixing new terms of A to obtain [], $(a :: [])$, $(a' :: (a :: []))$ etc.

In order to use this type, we also need an elimination rule (or recursion principle in the non-dependent case). The recursion principle tells us how to use terms of the type by defining functions out of it and will be familiar to anyone who has written a recursive function on lists.

$$\frac{\begin{array}{c}\Gamma \vdash b_0 : B \\ \Gamma \vdash b_{cons} : A \times [A] \to B\end{array}}{\Gamma \vdash rec_{[A]}(b_0, b_{cons}) : [A] \to B}$$

In words, this rule states that you can construct a function from $[A]$ to B if you have a term b_0 and a function that takes a pair of an A and a list to produce a B. As one might expect, the resulting function maps [] to b_0 and $(a :: as)$ to $b_{cons}(a, as)$. (By a computation rule that we also need to specify).

[MAYBE GIVE A MORE GENERAL TREATMENT LIKE IN [19] (4.1)]

2.5.2 Higher Inductive Types

When constructing ever more complicated types, it would be nice to have some control over which terms are identified. [Examples? Just quotients, maybe?]

One way to do this is *Higher Inductive Types*. Like inductive types, HITs are constructed from generators, but while the generators of an inductive type may only generate terms, the generators of a HIT may also generate paths.

[ELIMINATION RULES. "VARY CONTINUOUSLY"]

[SYNTHETIC TOPOLOGY?]

The prototypical example of a HIT is the circle S^1, because it is very simple comprising only a single point and one path. Its introduction and formation rules are:

$$\frac{}{\Gamma \vdash S^1\ Type} \qquad \frac{}{\Gamma \vdash base : S^1} \qquad \frac{}{\Gamma \vdash loop : base =_{S^1} base} \tag{2.7}$$

2.6 Agda

In this section we introduce Agda [21] – a dependently typed programming language / proof assistant. The goal is to introduce enough of its syntax and workings to follow the formalization in chapter 4.

The basic syntax of Agda is similar to that of Haskell [CITATION?], but with `:` for typing and significant use of unicode (including → for function types).

As an example of Agda as a dependently typed programming language, let us consider the type of vectors and operations on them. This is a simple dependent type which will give us a good look at Agda's syntax and features.

First, we are going to need the natural numbers (recall that vectors are a family of types indexed by natural numbers). The (Peano) natural numbers are an inductive type, which we introduce with the `data` keyword. It has two constructors: `zero` and `suc`.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

We can now define vectors as a family of types indexed by a type and a natural number. Vectors also have two constructors. The empy vector `[]` has length zero, and a vector of any length can be extended by adding a new element to the start. The implicit argument `{n : ℕ}` should be read as "for all natural numbers n..." (and in fact we could write `∀ {n}` since Agda can easily infer that n must be a natural number).

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Note that the data declaration has `A` before the colon, but ℕ after. This is because `A` stays constant over the two constructors, while the natural number varies.

The cons function (`_::_`) shows two important features of Agda's syntax: infix notation and currying. Infix functions can be used between its arguments, in this case `x :: xs`

would be a vector, and are denoted by underscores. Each underscore in the name represents a position in which we may place the corresponding argument.

Currying (Named after Haskell Curry [CITATION?]) is a way to describe functions with multiple arguments by making use of the product $\dashv$ exponentiation adjunction. This adjunction gives a bijection between $(A \times B) \to C$ and $A \to (B \to C)$ for all objects A, B and C [2] which means we can write the type of a function which takes multiple arguments as a sequence of types with right arrows (associating to the right).

[FOOTNOTE?] Mixfix operators and currying interact wonderfully with partial application. `x ::_` is the function that takes a vector and conses x onto it.

Now we can construct terms of this new type. For example, here is the 3-vector of natural numbers [1,2,3]:

```
one-two-three : Vec ℕ (suc (suc (suc zero)))
one-two-three = suc zero
  :: (suc (suc zero)
    :: (suc (suc (suc zero))
      :: []))
```

Clearly this way to write out natural numbers is pretty verbose. Agda's builtin type of naturals lets us write `3` instead of `suc (suc (suc zero))`.

We can also define convenient functions on vectors, like `map` and concatenation. Here `map` is defined by pattern matching on the vector. It applies a given function f to each element of the vector, potentially changing its underlying type, but not its length. The two types `A` and `B`, as well as the length of the vector, are left implicit and can be inferred from the provided function and vector.

```
map : {A B : Set}{n : ℕ} → (A → B) → Vec A n → Vec B n
map _ [] = []
map f (x :: v) = (f x) :: (map f v)
```

[2] For more on adjunctions in general, and this adjunction in particular see IV.6: Cartesian Closed Categories in [12]

Of course, map would work equally well for the non-dependent type of lists. To make use of this additional power we can define `map-pointwise` which safely applies a different function to each element.

```
map-pointwise : {A B : Set}{n : ℕ} →
                Vec (A → B) n → Vec A n → Vec B n
map-pointwise [] [] = []
map-pointwise (f :: fs) (x :: xs) = f x :: map-pointwise fs xs
```

Concatenation is the binary operation that adjoins one vector to the end of another. This has the effect of adding their lengths, evidenced by the resulting type `Vec A (n + m)`. Note that we only pattern match on the left vector. This is actually important, since `_+_` is defined by pattern matching on its left argument, allowing this definition to type-check. [SHOW +?]

```
_++_ : {A : Set} {n m : ℕ} → Vec A n → Vec A m → Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

In addition to being a dependently typed functional programming language (or perhaps more accurately, *by* being a dependently typed programming language) Agda is a proof assistant. By making use of "propositions as types" as well as Martin-Löf style identity types, proofs and programs are the same thing. Note that the Agda type `_≡_` is *not* the same as the judgemental equality from section 2.1. Rather, it is the identity type described in section 2.3.

The most basic proofs are simply `refl`. We can use refl to prove that one plus one is two, or that zero is the left unit of addition.

```
-- 1 + 1 = 2
_ : (suc zero) + (suc zero) ≡ suc (suc zero)
_ = refl

-- zero is the left unit for addition
+-lunit : ∀ {n} → zero + n ≡ n
+-lunit = refl
```

Of course, not all proofs are so simple. In fact, proving that zero is also the *right* unit takes some work. This is because addition is defined by induction on the left argument, so `+-lunit` is simply the base case.

```
-- zero is the right unit for addition
+-runit : ∀ {n} → n + zero ≡ n
+-runit {zero} = refl
+-runit {suc n} = cong suc +-runit
```

For `+-runit` we need a proof by induction. The base case ($0 + 0 = 0$) is proved by `refl` like before, but the induction step requires slightly more work. Luckily the term we need has type `(suc n + zero) ≡ suc n` and the left-hand side computes to `suc (n + zero)`. Now we have `suc` applied to both sides of an instance of `+-runit` so we can use the induction hypothesis with `cong : (f : X → Y) → x ≡ y → (f x) ≡ (f y)`. (Also note the pattern matching on an implicit argument.)

Another useful tool, mainly to make complicated proofs easier to follow, is `≡-Reasoning`, which introduces `_≡⟨_⟩_` and `_■`. These let the programmer write out the steps of a proof, like the inductive case of the proof below, such that `x ≡⟨ p ⟩ y` means "x is equal to y by p".

```
open ≡-Reasoning
concat-map : {A B : Set} {n m : ℕ} → (f : A → B) (v : Vec A n) (w : Vec A m)
             → map f (v ++ w) ≡ (map f v) ++ (map f w)
concat-map f [] w = refl
concat-map f (x :: v) w = map f ((x :: v) ++ w)
  ≡⟨ refl ⟩ map f (x :: (v ++ w))
  ≡⟨ refl ⟩ f x :: map f (v ++ w)
  ≡⟨ cong (f x ::_) (concat-map f v w) ⟩
    (map f (x :: v) ++ map f w) ■
```

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b\ Type} \qquad \frac{\Gamma, i : \mathtt{I} \vdash x(i) : A}{\Gamma \vdash \lambda i.x(i) : x(\mathtt{0}) =_A x(\mathtt{1})} \qquad \frac{\Gamma \vdash p : a =_A b}{\Gamma, i : \mathtt{I} \vdash p\ i : A}$$

Figure 2.1: Introduction-, formation- and elimination-rules for cubical paths

2.7 Cubical Type Theory

One way to imbue HoTT with computational meaning [INTRODUCE THIS PROBLEM SOMEWHERE] is Cubical type theory [3]. The basic idea is to take the "types as spaces"-interpretation of identity types very literally, as a function from an interval. In particular, it allows for non-axiomatic implementations of univalence and higher inductive types [4]. This section introduces the basic concepts of cubical type theory, Cubical Agda and the Cubical library.

The main ingredient of cubical type theory is the interval type. It represents the closed interval $[0, 1]$ in and we can think of it as a HIT with two points and an equality between them. Denote the interval by $\mathtt{I}$ and its two endpoints by $\mathtt{0}$ and $\mathtt{1}$. An element along the interval is represented by a variable $i : \mathtt{I}$

In addition to its elements, the interval supports three operations. The binary operations $\wedge$ and $\vee$ and the unary operation $\sim$. In the geometric interpretation these represent (respectively) max, min and $1 - \_$. These operations form a de Morgan algebra [15] (and in fact $\mathtt{I}$ may be described as the free de Morgan algebra on a discrete set of variable names $\{i, j, k...\}$ [3]).

We can now define a cubical identity type as functions out of the interval type. Concretely, an identity type $x =_A y$ is the type of functions $p : \mathtt{I} \to A$ such that $p(\mathtt{0}) \equiv x$ and $p(\mathtt{1}) \equiv y$. This corresponds precisely to the notion of a path with endpoints x and y in homotopy theory.

Using lambda-abstraction to define the functions we obtain the inference rules seen in Figure 2.1.

By iterating this construction we obtain higher homotopies. $\mathtt{I} \to A$ represents paths in A, $\mathtt{I} \to \mathtt{I} \to A$ squares, $\mathtt{I} \to \mathtt{I} \to \mathtt{I} \to A$ the eponymous cubes and so on.

2.7.1 Cubical Agda

Cubical Agda [25] implements support for cubical type theory in Agda based on the development by Cohen et al. [3]. Additionally it extends the theory to support records and co-inductive types, a general schema of HITs and univalence through `Glue` types. In this section we look at some examples of Cubical Agda to get familiar with its syntax.

As of Agda version 2.6.0, cubical mode can be activated with:

```
{-# OPTIONS --cubical #-}
```

First, let us consider the cubical path type as introduced in the preceding section. The interval type is denoted by `I`, its two end-points by $i0$ and $i1$ and the operations by $\_\wedge\_$, $\_\vee\_$, $\sim\ \_$. The most basic notion of a path is actually the heterogenous/dependent path type:

```
HPath : (A : I → Type) → A i0 → A i1 → Type
```

The non-dependent identity types as discussed in **??** corresponds to a `HPath` over a constant family:

```
Id : {A : Type} → A → A → Type
Id {A} x y = HPath (λ _ → A) x y
```

As one might expect, `refl` is the constant path

```
refl : {x : A} → x ≡ x
refl {x = x} = λ i → x
```

and symmetry is defined using $\sim\ \_$:

```
sym : {x y : A} → x ≡ y → y ≡ x
sym p = λ i → p (~ i)
```

Higher inductive types are defined by their point and path constructors. As an example, consider the circle S^1 as introduced in section 2.5.

```
data S¹ : Type where
  base : S¹
  loop : base ≡ base
```

Defining functions out of HITs is done by pattern matching. Notice the variable `i:I` which represents "varying along the path". This is the function from the circle to itself which reverses the direction of the loop.

```
reverse : S¹ → S¹
reverse base = base
reverse (loop i) = loop (~ i)
```

[MORE EXAMPLES? ENCODE/DECODE FOR WINDING?]

In addition to the cubical mode, Vezzosi, Mörtberg and Cavallo develop and maintain a cubical standard library [1] containing useful data types, functions and proofs.

2.7.2 Why Cubical Type Theory?

The main benefit of cubical type theories is that they make it possible to prove useful results that are usually only axiomatically defined. Two prominent examples are function extensionality and Voevodsky's univalence axiom [26].

In cubical type theory (and in particular in Cubical Agda) these are not axioms at all, but provable theorems. Function extensionality is especially straightforward: given two (possibly dependent) functions $f, g : A \to B$ and a family of paths $p : \Pi_{(x:A)}\ f(x)\ =_B\ g(x)$, the proof simply swaps the order of operations.

```
funExt : {A B : Type} {f g : A → B} (p : (x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

Univalence is also provable in the sense that a term of the type

[1] A standard library for Cubical Agda: `https://github.com/agda/cubical`

$$\{A\ B : \mathsf{Type}\} \to (A \equiv B) \simeq (A \simeq B)$$

can be constructed. It is often useful to have only one direction of the equivalence. The cubical standard library provides both in:

$$\mathsf{ua} : \{A\ B : \mathsf{Type}\} \to A \simeq B \to A \equiv B$$
$$\mathsf{lineToEquiv} : \{A\ B : \mathsf{Type}\} \to A \equiv B \to A \simeq B$$

Additionally, Cubical Agda's support for HITs and pattern matching on their constructors will be very useful.

The benefit of all this is canonicity. Since `ua` and HITs are non-axiomatic, terms constructed by their use actually compute to a value. This means our formalization actually computes the result of applying patches.

Sadly, however, this is not entirely true. There are two exceptions to canonicity at the time of writing:

1. `transp` over indexed families, and
2. `hcomp` over indexed families.

Regrettably we require both in order to realize repositories as vectors of strings (an indexed family).

[THE STATE OF AFFAIRS? OR LEAVE FOR CONCLUSION — MAYBE BOTH?]

Chapter 3

Version Control Systems

A version control system (VCS) is a software development tool commonly used to keep track of changes to a code base. With large projects, distributed teams and complex dependencies between version this can be a challenge and a plethora of different tools with a variety of theoretical approaches exist.

[NOT HAPPY WITH THIS] By far the most common tool is the distributed git [22] which employs *snapshot*-model in which the repository state is saved and changes are recorded by storing the line-by-line difference with previous snapshots. The distributed Mercurial [23] and centralized Subversion [2] work by similar models. These tools are fast, but often poorly understood and have some un-intuitive behavior [16, 17].

An alternative to snapshot-models is a *patch*-model in which the VCS instead stores the changes and computes the repository state from them. The most prominent example of such a system is Darcs [5] with its "algebra of patches", but we also mention Pijul [18] which takes a more categorical approach.

In this chapter we introduce the basic concepts and terminology, investigate some theoretical approaches to version control – including Homotopical Patch Theory, a formalization of which is covered in chapter 4 – and mention a novel approach based on type-theoretic co-equalizers which the author has not succeeded in implementing.

3.1 Background

This section introduces the terminology and overall structure of version control systems for use in the ensuing work. We focus on a patch-theoretic view in the style of Darcs [5, 11, 20, 8] with groupoid properties as in Homotopical Patch Theory [1].

3.1.1 Basic Ingredients

Version control systems keep track of *data*. In the most common use case this is one or more files containing lines of text, but we do not need to specialize. The basic ingredients of a VCS are *repositories* and *patches*. A repository contains the data we are keeping track of, and a patch records a change made to the repository.

3.1.2 (Groupoid) Properties of Patches

In Patch Theory a repository is a collection of patches, from which the data can be reconstructed. At any given time the specific collection of patches describe a *repository state*.

A patch records a change like "add the line l to the file f". This patch is nonsensical if there is no file f, so each patch has a domain *context* in which it can be applied. The context is a repository state, and a patch can be *applied* to a repository in the appropriate state. Each patch also has codomain context which is the state it leaves the repository in. In this section we denote a patch P with domain x and codomain y by ${}_xP_y$. We may leave out the contexts if they are not important.

For the theory to be useful we should be able to apply more than one patch to a repository. For this purpose there is patch composition. Given two patches ${}_xP_y$ and ${}_yQ_z$ with matching contexts there is a composite patch ${}_xP \cdot Q_z$ which is meant to represent "apply P and then apply Q".

In any given patch theory [1] we have some collection of possible patches and laws they must obey. In the context of this work we assume a few basic laws for all patch theories:

[1] While Patch Theory refers to the study of VCS as repositories characterized by collections of patches, *a* patch theory denotes a specific collection of patches and laws they obey.

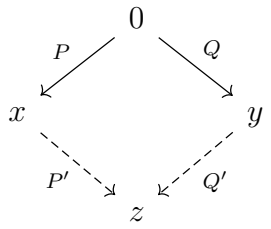

Figure 3.1: Merging (P, Q)

1. For each context x there exists a "do-nothing" patch ${}_xN_x$ such that for any patches ${}_xP_y$:

$$ {}_xN \cdot P_y \;=\; {}_xP_y \;=\; {}_xP \cdot N_y $$

   ie. R is both a left and right unit for composition.

2. Associativity of patch composition. Given patches ${}_xP_y$, ${}_yQ_z$ and ${}_zR_w$:

$$ {}_x(P \cdot Q) \cdot R_w \;=\; {}_xP \cdot (Q \cdot R)_w $$

3. For each patch ${}_xP_y$ there exists an inverse ${}_yP_x^{-1}$ such that:

$$ {}_xP \cdot P_x^{-1} \;=\; {}_xN_x $$

These laws are not arbitrary. In fact they express the assumption that a the repository states and patches between them form a *groupoid*. This will be especially important for subsection 3.2.3 and the following implementation in chapter 4.

3.1.3 Merging

Another common feature of VCSs is *branching*. Branching occurs in distributed systems when two users of the repository have different repository states. If the users wish to reconcile their states, they perform a *merge*.

In our setting merge is a function that takes a span of patches $({}_0P_x, {}_0Q_y)$ (here 0 is some shared base context) and produces a cospan $({}_xP'_z, {}_yQ'_z)$ (Figure 3.1).

For merge to be well behaved we might also want some other properties. Say merge is *symmetric* if

$$merge(P, Q) = (P', Q') \implies merge(Q, P) = (Q', P')$$

and that merge is *reconciling* if

$$merge(P, Q) = (P', Q') \implies P \cdot P' = Q \cdot Q'$$

Note that the groupoid laws imply that there always exists a merge of two patches which is both symmetric and reconciling: take $z = 0$, $P' = P^{-1}$ and $Q' = Q^{-1}$. This is the cospan that simply undoes the changes in both users' repositories and it is not very interesting. We call this the *trivial merge*.

3.2 Prior work / Theoretical Approaches to VCS

In this section we introduce some proposed theoretical models of version control systems. These are the patch theory of Darcs [5, 11], the categorical approach of Mimram and Di Giusto [18, 14] and Angiuli et al.'s "homotopical patch theory" [1] (hpt).

3.2.1 Patch Theory (Darcs)

Here we discuss several proposed formalisms for a the patch theory employed by Darcs [5]. [11, 20, 9] all attempt to describe Darcs' patch theory. (focus on Lynagh, I think)

Lynagh [11] proposes an "algebra of patches" as a theoretical basis for the Darcs [5] version control system.

In this model a repository state is a set of updates (called *patches*, but we want to avoid that ambiguity) and a patch is a change to this set. For example pulling the repository $\{c\}$ into the repository $\{a, b\}$ results in a new repository $\{a, b\} \cup \{c\} = \{a, b, c\}$.

Patches are only applicable to one repository state, and result in a new state. If they are compatible, we may string them together into a *patch sequence*. Denoting the previous example patch by P and the "do-nothing" patch by Id we have $\{a, b\}P\{a, b, c\}Id\{a, b, c\}$ – pulling $\{c\}$ followed by doing nothing. The repository state may be omitted from sequences.

Finally a notion of *commutation* of patches is defined. We say the patch sequence AB commute if there are patches A' and B' such that the following square commutes:

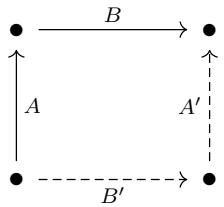

Figure 3.2: Commuting patches

and write $AB \leftrightarrow B'A'$. Note that the initial and final contexts (bottom left and top right, respectively) are the same, but the intermediary contexts need not be.

There are four axioms for patches and commutation:

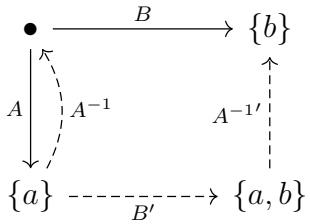

Figure 3.3: Merging A and B by commutation

1. Commutativity (lol) (3.1): $AB \leftrightarrow B'A' \iff B'A' \leftrightarrow AB$
2. Invertibility (3.2): for each A there is an A^{-1} s.t $AA^{-1} = A^{-1}A = Id$
3. Inv-cong (3.3): $AB \leftrightarrow B'A' \iff A^{-1}B' \leftrightarrow BA'^{-1}$. (we can start in the top left corner of Figure 3.2 if we want)
4. Circular (3.5/6): performing all pairwise commutations in a sequence gets us back to the beginning (or, a horrible equation)

These axioms allow us to define some useful operations on repositories. For example, given a span $\{a\} \xleftarrow{A} \bullet \xrightarrow{B} \{b\}$ we may want to incorporate the results of both patches to get $\{a, b\}$. We call this operation "merge" and proceed in three steps:

1. by invertibility, we can find a patch $\{a\}A^{-1}\bullet$
2. now that we have a sequence $A^{-1}B$, we commute it to get the sequence $B'A^{-1'}$
3. define merge(A, B) to be the sequence AB'.

This process is shown in Figure 3.3.

Another useful operation on repositories is "cherry picking". Cherry picking is the act of pulling some, but not all, patches from one repository into another. Consider the patch sequence $\{\}A\{a\}B\{a, b\}C\{a, b, c\}$ and a repository $\{a\}$. We want to incorporate the changes in C, but not the ones in B, but naively combining applying C does not work, since it is only applicable to the context $\{a, b\}$. The solution is to commute $BC \leftrightarrow C'B'$ (Figure 3.4) to obtain C' with the desired endpoints.

Problem: we cannot always commute patches, and Darcs does not have a great solution here.

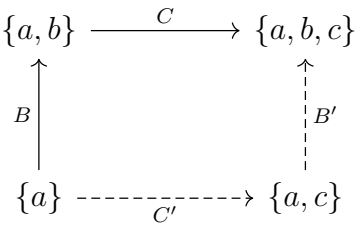

Figure 3.4: Commutation for cherry picking

3.2.2 A Categorical Theory of Patches

[THIS ISN'T REALLY RELEVANT TO ANYTHING ELSE, BUT IT'S A COOL IDEA]

A Categorical Theory of Patches [14] defines a category of files and patches, such that a merge is a pushout. To ensure a merge is always possible they first construct the category $\mathcal{L}$ of files and patches, and then its conservative cocompletion $\mathcal{P}$.

$\mathcal{P}$ contains all finite colimits – and in particular all pushouts – so the merge of a span is always defined. The paper's chief achievement is the explicit construction of this category and these pushouts.

Interesting insights I'm not sure how to incorporate:

- the construction of $\mathcal{P}$ can be understood as the addition of *partially* ordered files to $\mathcal{L}$.
- "flattening" these partial orders leads to cyclic graphs. On editing text [7] objects, but maybe not correctly
- the poset structure of $\mathcal{L}$ and $\mathcal{P}$ is given explicitly by $\mathcal{G}$ and the nerve functor $N_{\_}$ (!!).

(maybe mention Pijul [18] (if so, figure out the relationship to [14])) (maybe some figures go here)

3.2.3 Homotopical Patch Theory

Homotopical Patch Theory [1] (HPT) gives a way to formulate patch theories in homotopy type theory. The formulation makes intrinsic use of higher inductive types (HITs) and a univalent universe to encode relationships between patches and models of a theory.

HPT takes advantage of the inherent groupoid structure of types to encode the groupoid structure of patch theories "for free". A patch theory is given as a higher inductive type `R` with points representing patch contexts and paths representing patches between them. Paths come with units (`refl`), inverses (`sym`) and composition ($\cdot$), and by the groupoid laws this composition is associative, unital, and respects inverses.

Patch laws can be given by paths between paths (squares). For example we may want the application of two independent patches P and Q to commute – this is done with a square whose left- and right-sides are $P \cdot Q$ and $Q \cdot P$.

Models of a patch theory are given by a function `I : R → Type` from the HIT to the universe of types. Each repository state (point) is mapped to a type, and each patch is mapped to a path in the universe. By univalence such paths can be given by equivalences between types, and each patch $P : x \equiv y$ gives rise to a function `interp P : I(x) → I(y)` by `transport`. The functoriality of `I` ensures that such a model validates all the patch laws of `R`.

Angiuli et al. present three example patch theories in order of increasing complexity. Implementations in Cubical Agda are explored in more depth in chapter 4, but we give a brief overview here.

An Elementary Patch Theory

An elementary patch theory describes a VCS with one context `num`, and one patch `add1 : num ≡ num`. Being a HIT[2], the theory automatically includes identity patches, composition and even inverses.

[2]This HIT with one point and one non-identity loop may seem familiar. It is a renaming of the circle S^1!

The intended interpretation of the theory is a repository consisting of a single integer where applying `add1` adds 1 to it. However, there is nothing stopping us from giving a different interpretation. [MAYBE DO THAT? LATER?]

This example illustrates two important points. Firstly, that paths can carry computational content (in this case the successor function and its inverse) revealed by mapping into the universe of types; and secondly that a patch theory may permit multiple models.

A Patch Theory with Laws

In **a patch theory with laws** we see an example of patch laws – squares in the higher inductive type. Again the theory has one point and one constructor path, but the path `s ↔ t @ i` depends on two strings `s,t` and an index `i`.

The theory also includes two patch laws. `indep` asserts that two patches with different indexes commute, while `noop` asserts that a patch where $s = t$ is the same as `refl` - doing nothing.

The intended model for this theory is a fixed-length vector of strings where the patch `s ↔ t @ i` swaps the strings `s` and `t` at index `i`. In order to define this interpretation it is also necessary to give squares showing that it respects the patch laws.

The use of patch laws is illustrated by a **patch optimizer**. This is a function that takes a patch and produces a smaller patch with the same effect. The key idea is to take advantage of `noop` to replace instances of `s ↔ t @ i` with `refl` when `s` and `t` are equal.

Angiuli et al. give two ways to write the optimizer. In the **program then prove** approach they construct a function `opt1 : R → R` and then prove `∀ x → x ≡ opt1 x`. This requires set-truncation of `R`.

The **program and prove** approach avoids truncation by instead constructing a function `opt : (x : R) → Σ(y : R) y ≡ x` which produces both a new point *and* and proof that it is equal to the original. Since the resulting type is contractible, the squares witnessing the patch laws become trivial.

In both cases the actual *patch* optimizer is obtained by applying `opt` along a path.

A Patch Theory with Richer Contexts

The previous two examples show the utility of a patch theory as a HIT, but they do not capture the importance of contexts. In both, there is only one context and every patch is applicable to that context.

A patch theory with richer contexts has contexts `doc h` indexed by a *history* `h`, and the two kinds of patches `add s i` and `rm i` are only applicable to appropriate histories. In particular, a history has a "length" `n` and patches are only applicable when their index `i` is smaller than `n`.

This patch theory also has patch laws describing the relationships between adding and removing lines in different orders. We leave out the details of these laws for now, but note that the histories must also respect patch laws.

The interpretation of `doc h` for a history of length `n` is a single file containing `n` lines of text. What about patches? The `rm` patch should be a path between files of length `n` and files of length `n-1` but these types are not bijective. To solve this we compute the unique result of applying a history and map `doc h` to the singleton type of the result. Now a bijection can be obtained, since all singleton types are in bijection with each other.

Angiuli et al. then illustrates an interesting use of models as functions by defining two different models for this theory. The first model computes the resulting file as a vector of strings, while the second instead produces a log of all the patches that have been applied.

Merging in this richer theory is reduced to a function on histories from the empty file. This ensures that only patches in the "forward direction" are merged, and since histories respect patch laws so does their merge.

Computational Content of Paths

In all three examples, the models of patch theories make crucial use of different paths. In an elementary patch theory `refl`, `add1` and `add1 · add1` all have the same endpoints, but their effects as patches are very different. Therefore making use of this formulation requires

a type theory where paths (and in particular paths between types provided by univalence) carry computational meaning, such as Cubical Agda.

Additionally, the optimizer for a patch theory with laws maps into a contractible type in a non-trivial way. This may seem pointless, since all the elements in such a type are in a sense the same[3], but nevertheless we expect it to compute the correct patch. In practice we require some notion of "sub-homotopical" computation.

For these reasons it is worthwhile to implement the models cubically to gain a better understanding both of homotopical patch theories and cubical equality [25, 1].

[3]In fact any function of the type $(\mathtt{x} : \mathtt{R}) \to \Sigma_{(\mathtt{y}:\mathtt{R})}\ \mathtt{x} \equiv \mathtt{y}$ is homotopic to the "identity function" that maps x to (x , refl). [1]

3.3 Another Type-Theoretic Approach

In this section we explain a different approach to modeling VCSs in homotopy type theory based on a specific class of HITs called coequalizers. The basic idea is again to take advantage of the groupoid structure of types by modeling a VCS as a HIT with point constructors for contexts and path constructors for patches.

By giving the paths explicitly with endpoints, the result is essentially a (type-theoretic) coequalizer: the type of repository contexts quotiented by patches between them. Then, functions out of this type can be defined through a characterization of its patch space given by Kraus and von Raumer [10].

The goal of this approach is to define patch theories that are – in some sense – *semantic*. Operations and laws on patches should be definable in terms of both their endpoints and the *kind* of patch. [HOW DOES THIS CONNECT TO COEQUALIZERS?]

We present an unsuccessful attempt at implementing this approach in Cubical Agda along the results of Kraus and von Raumer, followed by a discussion of the problems encountered.

3.3.1 Repository HIT

The data of a coequalizer is a type A and a doubly indexed family of types $\sim\colon A \to A \to Type$ called a "proof relevant relation" on A. We write $a \sim b$ for the type $\Pi_{(a,b:A)}\ \sim\ a\ b$ of two related terms in A (leaving out explicit introduction of a and b). Then the coequalizer $A/\!\!/_\sim$ is a higher inductive type with points $[a]$ for $a : A$ and paths $glue\ p : [a] \equiv [b]$ for $p : a \sim b$ [10].

In formal terms this is the introduction rule:

$$\frac{\begin{array}{l}\Gamma \vdash A\ Type \\ \Gamma, a\ b : A \vdash a \sim b\ Type\end{array}}{\Gamma \vdash A/\!\!/_\sim\ Type} \tag{3.1}$$

And the two formation rules:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : A/\!\!/_\sim} \qquad \frac{\Gamma \vdash p : a \sim b}{\Gamma \vdash glue\ p : [a] \equiv [b]} \tag{3.2}$$

As a concrete running example we will consider a very simple VCS in which a repository consists of a single value that can be either `Nothing` or `Just x` for some term x, and a patch that sets a `Nothing` to `Just` some value.

```
module repo (A : Type₀) where
  data Maybe : Type₀ where
    Nothing : Maybe
    Just : A → Maybe

  data Simple : Type₀ where
    [_] : Maybe → Simple
    sett : ∀ a → [ Nothing ] ≡ [ Just a ]
```

3.3.2 Merge

Kraus and von Raumer's characterization of the path spaces of such coequalizers take the form of an induction principle for their paths. Given a coequalizer like before, a term $a_0 : A$, and a path-indexed family

$$P : \Pi_{(b:A)}([a_0] \equiv [b] \to Type)$$

the induction principle gives a way to construct a term of the type:

$$\Pi_{(b:A)}\Pi_{(q:[a0]\equiv[b])}P\ b\ q$$

a dependent function out of the (based) path type of $A/\!\!/_\sim$.

For non-HIT types the J-rule is the appropriate such principle, but this is not the case once we allow higher constructors. While the J-rule adequately covers the reflexive identities, it does not cover identities constructed by *glue*. We might try to remedy this by also requiring

a term of $P\ (glue\ s)$ for any $s : a_0 \sim b$, but this is also not sufficient. In particular, such a construction is not closed under symmetry and transitivity of identities. The solution is to instead require an equivalence $P\ q \simeq P(q \cdot (glue\ s))$ for each $q : [a_0] \equiv [b]$ and $s : b \sim c$.

The complete induction rule is given by: [in Agda? (specialized to our repo type)]

$$\frac{\begin{array}{l}\Gamma \vdash a_0 : A \\ \Gamma, b : A \vdash P : [a_0] \equiv [b] \to Type \\ \Gamma \vdash r : P\ refl_{[a_0]} \\ \Gamma, b\ c : A, q : [a_0] \equiv [b], s : b \sim c \vdash P\ q \simeq P\ (q \cdot (glue\ s))\end{array}}{\Gamma \vdash ind\ r\ e : \Pi_{(b:A)}\Pi_{q:[a_0]\equiv[b]} P\ q} \tag{3.3}$$

We will attempt to use this induction rule to define a merge function for our example VCS. For this purpose we introduce the types of spans and cospans indexed by their endpoints:

```
Span : Maybe → Maybe → Type₀
Span x y = Σ[ a ∈ Maybe ] ( [ a ] ≡ [ x ] ) × ([ a ] ≡ [ y ])

CoSpan : Maybe → Maybe → Type₀
CoSpan x y = Σ[ b ∈ Maybe ] ([ x ] ≡ [ b ]) × ([ y ] ≡ [ b ])
```

and, since merging is a binary operation, a binary induction rule

```
bin-path-ind : {ℓ : Level} → (a0 : Maybe) →
  (P : {b c : Maybe} → [ a0 ] ≡ [ b ] → [ a0 ] ≡ [ c ] → Type ℓ) →
  (P refl refl) →
  ({x : A} → (p : [ a0 ] ≡ [ Nothing ]) → P refl p ≃ P refl (p · sett x)) →
  ({x : A} (p : [ a0 ] ≡ [ Nothing ]) →
    ({c : Maybe} (q : [ a0 ] ≡ [ c ]) → P p q)
    ≃ ({c : Maybe} (q : [ a0 ] ≡ [ c ]) → P (p · sett x) q)) →
  ----------------------------------------------------------------
  {b : Maybe} → (p : [ a0 ] ≡ [ b ]) → {c : Maybe} → (q : [ a0 ] ≡ [ c ]) → P p q
bin-path-ind a0 P r e e' = ind (λ p → ({c : Maybe} → (q : [ a0 ] ≡ [ c ]) → P p q))
                               (ind (λ p → P refl p) r e) e'
```

Armed with binary path induction we can define the trivial merge which simply maps every span to the cospan reversing both patches.

```
mergeId : {x y : Maybe} → Span x y → CoSpan x y
mergeId {x = x} {y = y} (base , p , q) =
  bin-path-ind base
    (λ _ _ → CoSpan x y)
    (base , (sym p , sym q))
    (λ _ → idEquiv _)
    (λ _ → idEquiv _)
  p q
```

[NOT SURE WHAT/WHERE/HOW TO WRITE ABOUT THIS. I did a lot of work here with little to show for it]

Attempting to write more useful merges proved both laborious and difficult. In particular, the final term of `bin-path-ind` is an equivalence of (dependent) function types that is difficult to reason about and construct.

3.3.3 Result/Discussion

As presented, this approach leaves a lot to be desired. Apart from the trivial merge, it does not provide an intuitive or useful way to define merges even for very simple examples, and it is not clear [READ: I HAVE NOT PROVEN] that it extends to more complicated theories with multiple kinds of patches and patch laws.

Chapter 4

Formalization

[EMPHASIZE?]

1. computational content of paths [1]
2. computation in Cubical Agda [25, 1]

This section describes the development of a formalization of Homotopical Patch theory [1] in Cubical Agda. First the implementation details of three patch theories, then some computational results.

We show that the elementary patch theory performs as expected, but that "a patch theory with laws" and "a patch theory with richer contexts" require further development of type-checking tools to fully explore.

These limitations are known challenges. Of particular note is the Brunerie number [CITATION] and the "little Brunerie number" [CITATION].

4.1 An Elementary Patch Theory

This section discusses the implementation of a very simple language of patches, described in [**section 4**, 1].

4.1.1 The Circle as a Repository

In the elementary patch theory the repository is a single integer and there is exactly one kind of patch: adding one to the integer. This means the underlying type has one point constructor `num` and one path `add1 : num ≡ num`.

The structure of this type may seem familiar - it is just the circle with its constructors renamed! The cubical library already implements some HITs, including the circle so we will simply rename it and its constructors.

In fact this implementation comes with a proof that the fundamental group of S^1 is the integers, which contains many of the ingredients we will need. Specifically the loop space ΩS^1 is the type of patches, and `helix : S¹ → Type` is precisely the interpretation of points in `R` as types of repositories. Concretely `helix` maps `base` to the integers, and `loop` to `ua` of the equivalence $\mathbb{Z} \simeq \mathbb{Z}$ induced by the successor function.

```
open import Cubical.HITs.S1.Base public
  renaming(
    S¹ to R
  ; base to num
  ; loop to add1
  ; ΩS¹ to Patch
  ; helix to I
  )
```

With this machinery we can easily define an interpretation of patches as bijections on $\mathbb{Z}$ by applying `I` along the patch and weakening the resulting path. For convenience we also define a function to apply a patch to a given integer.

```
interp : Patch → ℤ ≃ ℤ
interp p = pathToEquiv (cong I p)

apply : Patch → ℤ → ℤ
apply p n = equivFun (interp p) n
```

4.1.2 Merge

Knowing that addition on the integers is commutative, merging two patches simply swaps the order.

```
merge : (Patch × Patch) → (Patch × Patch)
merge (p , q) = (q , p)
```

We now prove some properties of merge. Symmetry is essentially trivial, since swapping the order twice gets us back to where we started.

```
symmetric : { f1 f2 g1 g2 : Patch }
          → merge ( f1 , f2 ) ≡ ( g1 , g2 ) → merge ( f2 , f1 ) ≡ ( g2 , g1 )
symmetric p = cong merge p
```

Reconcile turns out to be more involved, but luckily some work is done for us. It boils down to showing that composition of patches commutes, which relies on two facts:

1. `intLoop` is a group homomorphism
2. addition on the integers is commutative

Both of these facts are in the standard library, so the task reduces to stitching them together. First we convert the patches to explicit integers n, m using the fact that `intLoop` is surjective. We then apply the proof of commutativity for integers, and convert back to patches.

It is noteworthy that we were able to define `merge` without reference to explicit numbers, but in order to prove its properties we require a "detour" into the integers.

```
intLoop-sur : (p : Patch) → ∃[ n ] (p ≡ intLoop n)
intLoop-sur p = apply p 0 , sym (decodeEncode num p)

patch-comm : (p q : Patch) → p · q ≡ q · p
patch-comm p q = let (n , p-is-n) = intLoop-sur p
                     (m , q-is-m) = intLoop-sur q in
  p · q ≡⟨ cong₂ _·_ p-is-n q-is-m ⟩
  intLoop n · intLoop m ≡⟨ intLoop-hom n m ⟩
  intLoop (n + m) ≡⟨ cong intLoop (+Comm n m) ⟩
  intLoop (m + n) ≡⟨ sym (intLoop-hom m n) ⟩
  intLoop m · intLoop n ≡⟨ cong₂ _·_ (sym q-is-m) (sym p-is-n) ⟩
  q · p ■
```

With the commutativity of patches established, reconcile follows easily:

```
reconcile : {f1 f2 g1 g2 : Patch}
          → merge (f1 , f2) ≡ (g1 , g2) → f1 · g1 ≡ f2 · g2
reconcile p = let f1=g2 = cong snd p
                  g1=f2 = cong fst (sym p) in
  (cong₂ _·_ f1=g2 g1=f2) · (patch-comm _ _)
```

4.2 A Patch Theory With Laws

In this section we explore a formalization of HPTs section 5: **A Patch Theory with Laws**. This is a more complicated patch theory in which the type `R` has not only repositories and patches, but also patch *laws* represented by squares (paths between paths).

We start by implementing the patch theory, followed by a "patch optimizer" that computes smaller patches with the same effect. This optimizer makes crucial use of the patch law.

4.2.1 The Patch Theory

In this patch theory we consider repositories consisting of a single file with lines of text. There is one type of patch which permutes the line at a given index. Let `Patch` denote the type `doc ≡ doc`.

Additionally we enforce patch *laws* with the `noop` constructor which states that swapping a string for itself is the same as doing nothing.

In the geometric interpretation of HITs this is a space with one point, loops for each choice of `(s1, s2, i)` and a square between each loop where `s1 == s2` and the constant path.

```
data R : Type₀ where
  doc : R
  _↔_AT_ : (s1 s2 : String) (i : Fin size) → (doc ≡ doc)
  noop : (s : String) (i : Fin size) → s ↔ s AT i ≡ refl
```

Angiuli et al's original definition also includes an additional law:

```
indep : (s t u v : String) (i j : Fin size) → (i ≢ j) →
        (s ↔ t AT i) · (u ↔ v AT j)
      ≡ (u ↔ v AT j) · (s ↔ t AT i)
```

This law states that swapping strings commutes as long as the indices are different. We do not include this law as it leads to some problems later. See subsection 4.2.2.

In order to interpret this model in the universe of types we will need three things:

1. a *type* of repository contexts `repoType`,
2. a path `swap` from `repoType` to itself for each choice of strings and index, and
3. a path of paths between `swap s s i` and `refl`

The type of repositories will be realized by vectors of strings of a fixed size.

```
repoType : Type₀
repoType = Vec String size
```

To create a path `swap s1 s2 i : repoType ≡ repoType` we will first construct a bijection, and then use `ua` to make a path in the universe.

Semantically, our patch should swap the line at index `j` if it is equal to either `s1` or `s2` and otherwise leave it alone. This behavior is encoded in `permute` and `permuteAt` applies it to the appropriate index.

```
permute : (String × String) → String → String
permute (s1 , s2) s with s =? s1 — s =? s2
... — yes _ — _     = s2
... — no _ — yes _ = s1
... — no _ — no _ = s

permuteAt : String → String → Fin size → repoType → repoType
permuteAt s t j = _[ j ]%= (permute (s , t))
```

To show that `permuteAt` is a bijection (and hence an equivalence) we need some additional results.

First we show that updating at the same index twice is equal to updating once with the composition of the functions.

```
[]%=twice : ∀ {n} {A : Type₀} (f : A → A) (v : Vec A n) (i : Fin n)
          → (v [ i ]%= f [ i ]%= f) ≡ (v [ i ]%= f ∘ f)
```

Then we show that updating by the identity function does not change the vector.

```
[]%=id : ∀ {n} {v : Vec String n} {j : Fin n} → v [ j ]%= id ≡ v
```

Both are proven by induction on the index.

Finally, permuting twice is equivalent to the identity function. The pointwise result `permuteTwice' : ∀ x → permute (s , t) (permute (s , t) x) ≡ id x` is straightforwardly (but laboriously) proven by case analysis, from which the full result follows by function extensionality.

```
permuteTwice : ∀ {s t} → (permute (s , t) ∘ permute (s , t)) ≡ id
permuteTwice = funExt permuteTwice'
```

With these facts it follows that permuting at an index is its own inverse, and an equivalence `swapat` can be constructed from this isomorphism.

```
permuteAtTwice : ∀ s t j v → permuteAt s t j (permuteAt s t j v) ≡ v
permuteAtTwice s t j v = permuteAt s t j (permuteAt s t j v)
  ≡⟨ []%=twice (permute (s , t)) v j ⟩
    v [ j ]%= permute (s , t) ∘ permute (s , t)
  ≡⟨ cong (v [ j ]%=_) permuteTwice ⟩
    v [ j ]%= id
  ≡⟨ []%=id ⟩ v ■

swapat : (String × String) → Fin size → repoType ≃ repoType
swapat (s , t) j = isoToEquiv
  (iso (permuteAt s t j) (permuteAt s t j) (permuteAtTwice s t j) (permuteAtTwice s t j))
```

For the `noop` law we need to show that `swapat` respects it. We proceed in two steps. First `swapssId` shows that the underlying function of the equivalence `swapat (s , s) j` is the identity function. Then, since two equivalences are equal if their underlying functions are equal we get an identification of `swapat (s , s) j` and the identity equivalence.

```
swapssId : {s : String} {j : Fin size} → equivFun (swapat (s , s) j) ≡ idfun (repoType)
swapssId {s} {j} = funExt pointwise
  where
```

```
pointwise : (r : repoType) → equivFun (swapat (s , s) j) r ≡ idfun repoType r
pointwise r = equivFun (swapat (s , s) j) r
              ≡⟨ cong (λ x → r [ j ]%= id x) permuteId ⟩
                r [ j ]%= id
              ≡⟨ []%=id ⟩
                id r ■

swapatIsId : {s : String} {j : Fin size} → swapat (s , s) j ≡ idEquiv repoType
swapatIsId = equivEq swapssId
```

With these pieces we are ready to interpret the repository HIT. `I` sends `doc` to the type of string vectors, each patch to `ua` of the `swapat` bijection and each `noop` square to `swapatIsId` composed with `uaIdEquiv` which the path identifying `ua (idEquiv _)` and `refl`.

Then we can interpret and apply patches like before.

```
I : R → Type₀
I doc = repoType
I ((s1 ↔ s2 AT j) i) = ua (swapat (s1 , s2) j) i
I (noop s j i i') = (cong ua (swapatIsId {s} {j}) · uaIdEquiv) i i'

interp : Patch → repoType ≃ repoType
interp p = pathToEquiv (cong I p)

apply : Patch → repoType → repoType
apply p = equivFun (interp p)
```

4.2.2 A Patch Optimizer

With the patch theory above it is possible to implement a patch optimizer – a function that takes a patch and produces a new (hopefully smaller) patch with the same effect. The development makes use of the `noop` patch law.

Specifically we implement the *program and prove* approach from section 5.3 of HPT [1]. With this approach we produce a function of type `(p : Patch) → Σ(q : Patch) p ≡ q`. The result is a patch `q`, along with a proof that `q` is equal to the original patch.

We proceed in two steps. First creating a function

```
opt : (x : R) → Σ[ y ∈ R ] y ≡ x
```

that performs the desired optimization on points, and then applying it along a patch with `cong`. The point constructor `doc` gets mapped to itself along with `refl`. This is natural since we want to optimize patches and leave the repositories unchanged. [MORE EXPLANATION/JUSTIFICATION?]

```
opt doc = (doc , refl)
```

The path constructor `s1 ↔ s2 AT j` is where we implement our optimization. If the two strings are different, we do nothing. Note that `x` here captures the interval paramter, and so "varies along the path" as required.

If the strings *are* equal we replace the patch with $\texttt{refl}_{\texttt{doc}}$ by mapping to `doc` regardless of the interval parameter. Now, our result type also requires a proof that `refl` is in fact equal to permuting two equal strings and we have exactly what we need: it's `noop`!

There are two complications:

1. `noop` requires the strings to be the same, not just equal. Luckily we can use the proof that they are equal to get a patch of the correct type
2. the `noop` square goes the wrong way, but this is easily fixed by inverting one interval argument.

```
opt x@((s1 ↔ s2 AT j) i) with s1 =? s2
... — yes s1=s2 = doc
                , λ k → ((cong (_↔ s2 AT j) (ptoc s1=s2) · noop s2 j) (~ k) i)
... — no _ = x , refl
```

For the `noop` constructor we make use of the fact that our codomain is contractible. Since we are mapping into a contractible type (and hence a Set) we know that all paths are equal, and can construct a square with sides matching the paths above.

However, since the sides must be *definitionally* equal in Cubical Agda we employ a trick from the set-truncation HIT elimination rule in the Cubical library. `isOfHLevelDep 2` is the dependent version of `isSet`. We can then provide the sides `cong opt (s ↔ s AT j)` and `refl` (or really `cong opt refl`). Since we are constructing a *dependent* square we also need a family of types `I → I → Type`, but this is exactly what `noop s j` is!

```
opt (noop s j i k) = isOfHLevel→isOfHLevelDep 2
  (isProp→isSet ∘ isContr→isProp ∘ result-contractible)
  _ _ (cong opt (s ↔ s AT j)) refl (noop s j) i k
```

This trick is the reason `indep` was left out. Because we need to apply `opt` to the paths to compute the sides of the square it would not terminate, instead constructing squares back and forth between `(s ↔ t AT i) · (u ↔ v AT j)` and `(u ↔ v AT j) · (s ↔ t AT i)` for eternity.

There is one additional complication: The result of `cong opt p` for some patch `p` is actually of type `Pathover (λ x → Σ(y : R) y ≡ x) p (doc,refl) (doc,refl)`. Luckily this type is equivalent to our desired target type by:

```
e : {p : Patch} →
    (PathP (λ i → Σ[ y ∈ R ] y ≡ p i) (doc , refl) (doc , refl))
    ≡ (Σ[ q ∈ Patch ] p ≡ q)
```

[SHOULD THIS JUST BE PRESENTED AS A PROPOSITION/PROOF?]

By the characterizations of paths over constant families and paths in Σ-types this [WHAT?] is equivalent to $\Sigma_{\texttt{q}\,:\,\texttt{Patch}}\ \texttt{(transport}\ ^{x\mapsto(x\equiv\texttt{doc})}\ \texttt{p)} \equiv \texttt{refl}$.

[USING BOOK SYNTAX I HAVE NOT INTRODUCED..]

```
(PathP (λ i → Σ[ y ∈ R ] y ≡ p i) (doc , refl) (doc , refl))
  ≡⟨ PathP≡Path (λ i → Σ[ y ∈ R ] y ≡ p i) (doc , refl) (doc , refl) ⟩
Path (Σ[ y ∈ R ] y ≡ doc) (transport (λ i → Σ[ y ∈ R ] y ≡ p i) (doc , refl)) (doc , refl)
  ≡⟨ cong (λ x → Path (Σ[ y ∈ R ] y ≡ doc) x (doc , refl)) (ΣPathP (refl , sym (lUnit p))) ⟩
Path (Σ[ y ∈ R ] y ≡ doc) (doc , p) (doc , refl)
  ≡⟨ sym ΣPath≡PathΣ ⟩
```

```
  (Σ[ q ∈ Patch ] (PathP (λ i → q i ≡ doc) p refl))
    ≡⟨ Σ-cong-snd (λ q → PathP≡Path (λ i → q i ≡ doc) p refl) ⟩
  (Σ[ q ∈ Patch ] (transport (λ i → q i ≡ doc) p) ≡ refl)
```

Then we apply lemma 2.11.2 from the book [1] to obtain the Σ-type of patches `q` and proofs that $q^{-1} \cdot p \equiv$ `refl`. [THIS PROOF IS A (SMALL, BUT) GENUINE CONTRIBUTION. NOTE?]

```
  (Σ[ q ∈ Patch ] (transport (λ i → q i ≡ doc) p) ≡ refl)
    ≡⟨ Σ-cong-snd (λ q → cong (_≡ refl) (path-transport-lemma q p)) ⟩
  (Σ[ q ∈ Patch ] (sym q · p) ≡ refl)
```

Finally, we reach the desired type by the groupoid properties of path composition.

```
  (Σ[ q ∈ Patch ] (sym q · p) ≡ refl)
    ≡⟨ Σ-cong-snd (λ q → invLUnique q p) ⟩
  (Σ[ q ∈ Patch ] p ≡ q) ■
```

In particular $p^{-1} \cdot q \equiv$ `refl` is equivalent to $q \equiv p$.

```
invLUnique : {X : Type} {x y : X} →
             (p q : x ≡ y) → (sym p · q ≡ refl) ≡ (q ≡ p)
```

Finally, `optimize` can be implemented as discussed – by applying `opt` and transporting along `e`.

```
optimize : (p : Patch) → Σ[ q ∈ Patch ] p ≡ q
optimize p = transport e (cong opt p)
```

[1] For the category theorist: this is the functorial action of the contravariant hom-functor [24]

4.3 A Patch Theory With Richer Contexts

The previous patch theories have both described repositories with a single context – in which patches are always applicable. In this section we explore a theory with more complex contexts by implementing Angiuli et al.'s **Patch Theory With Richer Contexts**.

4.3.1 The Type of Repositories

The intended model for this theory is one where the patches either insert a string `s` on the `l`th line (`ADD s AT l`), or remove the `l`th line (`RM l`).

Clearly these patches are not always applicable. It does not make sense to remove the 4th line of a file with only 3 lines, nor to insert something on line 14. To incorporate this more complicated patch language, the repository type must also be more complicated. This is accomplished by indexing `R` by a type of patch histories, where `History m n` is the type of sequences of patches which takes an `m`- line file to an `n`-line file.

[I AM LEAVING OUT PATCH LAWS BECAUSE THEY ARE HARD]

```
data History : ℕ → ℕ → Type₀ where
  []       : {m : ℕ} → History m m
  ADD_AT_::_ : {m n : ℕ} (s : String) (l : Fin (suc n)) →
               History m n → History m (suc n)
  RM_::_ : {m n : ℕ} (l : Fin (suc n)) →
           History m (suc n) → History m n


data R : Type₀ where
  doc  : {n : ℕ} → History 0 n → R

  addP : {n : ℕ} (s : String) (l : Fin (suc n))
         (h : History 0 n) → doc h ≡ doc (ADD s AT l :: h)
  rmP : {n : ℕ} (l : Fin (suc n))
        (h : History 0 (suc n)) → doc h ≡ doc (RM l :: h)
```

Another challenge with this richer theory is its interpretation. In the previous theories, the context was modelled by a single type, and patches by an equivalence on this type. In this setting we need a slightly different approach.

While it is natural to model a file of n lines as an `n`-vector of strings, such a solution would lead to problems for patches. For example, `add s l []` would need to be an equivalence of `Vec String 0` and `Vec String 1` which is not possible since these types are not equivalent.

Instead we note that a history actually determines a particular vector (by `replay`), and use the singleton type of this vector. Now, since any function between singletons determines a bijection (`singl-bijject`) and a function on elements determines a function on singletons (`mapSingl`), it suffices to define appropriate functions on vectors (`add` and `rm`) to obtain the needed equivalences.

```
replay : {n : ℕ} → History 0 n → Vec String n
replay [] = []
replay (ADD s AT l :: h) = add s l (replay h)
replay (RM l :: h) = rm l (replay h)

Interpreter : R → Type
Interpreter (doc x) = singl (replay x)
Interpreter (addP s l h i) = ua (singl-bijject {a = replay h} (mapSingl (add s l))) i
Interpreter (rmP l h i) = ua (singl-bijject {a = replay h} (mapSingl (rm l))) i
```

4.3.2 A Merge Function

We now turn our attention to defining a merge function. For this purpose we introduce an alternate interpreter which interprets repositories not as vectors, but just as their history. The purpose is to reduce the problem of defining the merge of patches to the problem of defining a merge of histories.

```
InterpreterH : R → Type
InterpreterH (doc x) = singl x
InterpreterH (addP s l h i) = ua (singl-bijject {a = h} (mapSingl (ADD s AT l ::_))) i
```

```
InterpreterH (rmP l h i) = ua (singl-biject {a = h} (mapSingl (RM l ::_))) i

interpH : ∀ {n m}{h : History 0 n}{h' : History 0 m} → doc h ≡ doc h' → singl h ≃ singl h'
interpH p = (pathToEquiv (cong InterpreterH p))

applyH :{n1 n2 : ℕ} {h1 : History 0 n1} {h2 : History 0 n2} →
        doc h1 ≡ doc h2 → InterpreterH (doc h1) → InterpreterH (doc h2)
applyH p = equivFun (interpH p)
```

Another issue is the following: when is a merge a meaningful operation? To answer that question we introduce the concept of an extension. A history `h2` is an extension of `h1` if `h2` has `h1` as a prefix (there exists a `h3` such that `h1 +++ h3` is equal to `h2`).

```
Extension : {n m : ℕ} → History 0 n → History 0 m → Type
Extension {n} {m} h1 h2 = Σ[ h3 ∈ History n m ] (h1 +++ h3) ≡ h2
```

Here `+++` denotes straight-forward concatenation of histories.

It is straight-forward to turn a history into a path in `R`, and likewise to turn an extension into a path. Note that `extToPath` actually ignores the extension itself, instead computing the patch going "via" the empty file.

```
toPath : {n : ℕ} (h : History 0 n) → doc [] ≡ doc h
toPath [] = refl
toPath (ADD s AT l :: h) = (toPath h) · addP s l h
toPath (RM l :: h) = (toPath h) · rmP l h

extToPath : {n m : ℕ} {h : History 0 n} {h' : History 0 m} →
            Extension h h' → doc h ≡ doc h'
extToPath {h = h} {h' = h'} _ = sym (toPath h) · toPath h'
```

This successfully reduces merging to a function on histories. Let us assume such a function:

```
mergeH : {n m : ℕ} →
         (h1 : History 0 n) (h2 : History 0 m) →
         Σ[ n' ∈ ℕ ] (Σ[ h' ∈ History 0 n' ] (Extension h1 h' × Extension h2 h'))
```

We can then obtain histories through `InterpreterH`, apply the history merger, and turn the resulting extensions back into paths.

```
merge : {n1 n2 : ℕ}{h1 : History 0 n1}{h2 : History 0 n2}
      → (doc [] ≡ doc h1) → (doc [] ≡ doc h2)
      → Σ[ n' ∈ ℕ ] (Σ[ h' ∈ History 0 n' ] (doc h1 ≡ doc h') × (doc h2 ≡ doc h'))
merge p1 p2 = let (p1H , p1P) = applyH p1 ([] , refl)
                  (p2H , p2P) = applyH p2 ([] , refl)
                  (_ , (h' , ((ext1 , ext1-proof) , (ext2 , ext2-proof)))) = mergeH p1H p2H
                  e1 = ext1 , cong (_+++ ext1) p1P · ext1-proof
                  e2 = ext2 , cong (_+++ ext2) p2P · ext2-proof
  in (_ , (h' , extToPath e1 , extToPath e2))
```

4.4 Computational Results

Having implemented several patch theories we can have a look at what they actually do. In this section we do exactly that, considering some concrete examples of repositories, patches and merges for the three theories.

[NOTES]

- rewrite?
- trans/hcomp problems
- mention Brunerie number (and the smaller Brunerie nr.)

4.4.1 Elementary Patch Computations

First, consider the elementary patch theory implemented in section 4.1. Recall that this theory has one type of repositories – the integers – and one patch – `add1`.

By the usual path operations we obtain some more patches: the "do nothing"-patch `noop`, the inverse `sub1` and compositions like `add2`.

```
noop sub1 add2 : Patch
noop = refl
sub1 = sym add1
add2 = add1 · add1
```

All of these suggestively named patches behave as one might expect:

```
_ : apply noop 1 ≡ 1
_ = refl

_ : apply add1 1 ≡ 2
_ = refl

_ : apply sub1 1 ≡ 0
```

```
_ = refl

_ : apply add2 1 ≡ 3
_ = refl

_ : apply (add1 · sub1) 1 ≡ 1
_ = refl
```

We can generalize further and create patches to add or subtract any integer, and these also compute as expected.

```
-- maybe hide this definition
addN : ℤ → Patch
addN (pos zero) = noop
addN (pos (suc n)) = add1 · addN (pos n)
addN (negsuc zero) = sub1
addN (negsuc (suc n)) = sub1 · addN (negsuc n)

_ : apply (addN 22) 20 ≡ 42
_ = refl

_ : apply (addN (- 22)) 42 ≡ 20
_ = refl
```

Clearly, this patch theory is a fully functioning calculator (for integer addition and subtraction), but the detour through algebraic topology takes a computational toll. The following proof typechecks, but it takes about 2 minutes.

```
_ : apply (addN 1000) 0 ≡ 1000
_ = refl
```

Finally, we look at `merge`. The function `merger` neatly computes the result of merging patches p and q from the original repository n as shown in Figure 4.1.

```
merger : ℤ → Patch → Patch → ℤ × ℤ
merger n p q = let x = apply p n
```

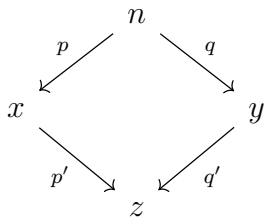

Figure 4.1: `merger`

```
                y = apply q n
                (p' , q') = merge (p , q)
  in (apply p' x , apply q' y)
```

Applying `merger` to a few test cases, it too behaves as expected. The resulting two integers are always equal, which is exactly what we want merge to do. Of course this is a consequence of the general case proven by `reconcile` in section 4.1, but it is good to see it in practice.

```
_ : merger 0 noop sub1 ≡ (-1 , -1)
_ = refl

_ : merger 0 (addN 5) (addN (-3)) ≡ (2 , 2)
_ = refl
```

4.4.2 Patch Computations with Laws

Next we examine the patch theory with laws and its patch optimizer from section 4.2. In this theory the repository is a fixed-length vector of strings, and the patches permute the string at a given index.

For concrete examples, consider the starting repository and patches:

```
repo : repoType
repo = "hello" :: "world" :: []

nop swap swap' comp : Patch
```

```
nop = "nop" ↔ "nop" AT (# 0)
swap = "hello" ↔ "greetings" AT (# 0)
swap' = "world" ↔ "earthlings" AT (# 1)
comp = swap · swap'
```

When applying these patches, we encounter the current limits of Cubical Agda. In particular, `Vec String size` is a dependent family so `transp` and `hcomp` do not compute on it. In the simple case of applying just one patch the issue is resolved by `transportRefl : (x : A) → transport refl x ≡ x`, giving the expected result.

```
_ : apply nop repo ≡ repo
_ = transportRefl repo

_ : apply swap repo ≡ "greetings" :: "world" :: []
_ = transportRefl _
```

In the case of composition it gets more difficult. The following cannot be proven by `transportRefl`, since the computation gets stuck on the composition.

```
_ : apply comp repo ≡ "greetings" :: "earthlings" :: []
_ = {!!}
```

Of course it is possible to compute the result by hand. Here we have some more information about the patches being composed, and are able to eliminate the composition before applying the patch.

```
_ : apply (nop · swap) repo ≡ "greetings" :: "world" :: []
_ = apply (nop · swap) repo
  ≡⟨ cong (λ p → apply (p · swap) repo) (R.noop "nop" (# 0)) ⟩
    apply (refl · swap) repo
  ≡⟨ cong (λ p → apply p repo) (sym (lUnit swap)) ⟩
    apply swap repo
  ≡⟨ transportRefl _ ⟩ ("greetings" :: "world" :: []) ■
```

Applying the patches in order also produces the expected result. [THIS SHOULD BE EQUIVALENT TO COMPOSITION, BUT IT'S REALLY HAIRY]

```
_ : apply swap (apply swap' repo) ≡ "greetings" :: "earthlings" :: []
_ = cong (apply swap) (transportRefl ("hello" :: "earthlings" :: [])) · transportRefl _
```

In addition to the patches themselves this theory includes an optimizer making use of the patch laws. In our implementation these optimized patches come equipped with a proof that they are equal to the original patch, so testing the results should not reveal anything new – nevertheless it is interesting to note just how slow these computations are.

All three exhaust the heap, taking on the order of 10s of minutes to do so, `swapOpt` is particularly notable since `optimize` does not actually do anything. The strings in `swap` are not equal, and so the patch should be kept as it is.

```
nopOpt swapOpt compOpt : Patch
nopOpt = fst (optimize nop)
swapOpt = fst (optimize swap)
compOpt = fst (optimize comp)

-- _ : apply swapOpt repo ≡ "greetings" :: "world" :: []
-- _ = transportRefl "greetings" :: "world" :: []

-- _ : apply nopOpt repo ≡ repo
-- _ = transportRefl repo

-- _ : apply compOpt repo ≡ "greetings" :: "earthlings" :: []
-- _ = transportRefl _
```

4.4.3 Richer Contexts

Finally, we consider the patch theory with richer contexts from section 4.3. For this theory we have implemented two interpretations, we will look at them in turn followed by merging. For the purpose of testing, define a few simple patches:

```
addPatch : doc [] ≡ doc (ADD "hello" AT zero :: [])
addPatch = addP "hello" zero []

rmPatch : doc (ADD "hello" AT zero :: []) ≡ doc (RM zero :: (ADD "hello" AT zero :: []))
rmPatch = rmP zero (ADD "hello" AT zero :: [])
```

Vector Interpretation

The first interpretation sends each `doc h` to a singleton type of the vector determined by replay. For the simplest patches this works as expected with `transportRefl`. (Here `S` denotes the inclusion into the singleton type.)

```
_ : apply addPatch (S []) ≡ S ("hello" :: [])
_ = transportRefl _

_ : apply rmPatch (S ("hello" :: [])) ≡ S []
_ = transportRefl _

_ : apply rmPatch (apply addPatch (S [])) ≡ S []
_ = cong (apply rmPatch) (transportRefl ("hello" :: [] , refl))
    · (transportRefl ([] , refl))
```

Again, direct composition of patches runs in to the current limits of Cubical Agda. Because `hcomp` does not reduce in singletons (which is a Σ-type), we get stuck trying to compute the (enormous) composition term.

```
-- _ : apply (addPatch · rmPatch) (S []) ≡ S []
-- _ = apply (addPatch · rmPatch) (S (replay []))
--   ≡⟨ transportRefl _ ⟩ _
--   ≡⟨ {!!} ⟩ S (replay (RM zero :: (ADD "hello" AT zero :: []))) ■
```

History Interpretation

The second interpretation eludes replaying the patches, instead sending `doc h` to the singleton history `h`. In a familiar turn of events the simple patches give expected results, but composition poses a problem. (Note that `[]` in these examples is the empty *history* rather than the empty vector.)

```
_ : applyH addPatch (S []) ≡ S (ADD "hello" AT zero :: [])
_ = transportRefl _

_ : applyH rmPatch (S (ADD "hello" AT zero :: []))
  ≡ S (RM zero :: (ADD "hello" AT zero :: []))
_ = transportRefl _
```

Merge

In section 4.3 we reduced the task of merging patches to merging histories. As a concrete example, consider a merger of histories which keeps one history if the other is empty, but simply undos the changes in both branches if there is a possibility of conflict.

```
undo-merge : {n m : ℕ} →
             (h1 : History 0 n) (h2 : History 0 m) →
             Σ[ n' ∈ ℕ ] (Σ[ h' ∈ History 0 n' ] (Extension h1 h' × Extension h2 h'))
undo-merge {_} {m} [] h2 = m , h2 , (h2 , +++-left-id h2) , ([] , refl)
undo-merge {n} {_} h1 [] = n , h1 , ([] , refl) , (h1 , +++-left-id h1)
undo-merge {_} {_} h1 h2 = 0 , [] , (undo h1 , undo-inverse h1) , (undo h2 , undo-inverse h2)
open merging {undo-merge}
```

We further define some simple patches

```
p1 : doc [] ≡ doc (ADD "hello" AT zero :: [])
p1 = addP "hello" (zero) []

p2 : doc (ADD "hello" AT zero :: []) ≡ doc (ADD "world" AT suc zero :: (ADD "hello" AT zero ::
p2 = addP "world" (suc zero) (ADD "hello" AT zero :: [])
```

and observe that the merged histories (or at least their lengths) give the expected results by `transportRefl`. Merging `p1` with `refl` keeps `p1`:

```
_ : fst (merge refl p1) ≡ 1
_ = fst (undo-merge (fst (applyH refl ([] , refl))) ((fst (applyH p1 ([] , refl)))))
  ≡⟨ cong {y = []} (λ x → fst (undo-merge x (fst (applyH p1 ([] , refl))))) (transportRefl _) ⟩
    fst (undo-merge [] ((fst (applyH p1 ([] , refl)))))
  ≡⟨ cong {y = ADD "hello" AT zero :: []} (λ x → fst (undo-merge [] x)) (transportRefl _) ⟩
    fst (undo-merge [] (ADD "hello" AT zero :: [])) ■
```

Merging two non-empty patches results int he empty patch:

```
_ : fst (merge p1 p1) ≡ 0
_ = fst (undo-merge (fst (applyH p1 ([] , refl))) ((fst (applyH p1 ([] , refl)))))
```

```
  ≡⟨ cong {y = (ADD "hello" AT zero :: [])}
          (λ x → fst (undo-merge x (fst (applyH p1 ([] , refl))))) (transportRefl _) ⟩
    fst (undo-merge (ADD "hello" AT zero :: []) (fst (applyH p1 ([] , refl))))
  ≡⟨ cong {y = (ADD "hello" AT zero :: [])}
          (λ x → fst (undo-merge (ADD "hello" AT zero :: []) x)) (transportRefl _) ⟩
    fst (undo-merge (ADD "hello" AT zero :: []) (ADD "hello" AT zero :: [])) ■
```

Finally we note that composition does not pose a problem here, since `undo-merge` extracts the history and does not need to compute the actual composition of patches.

```
_ : fst (merge (p1 · p2) refl) ≡ 2
_ = fst (undo-merge (fst (applyH (p1 · p2) ([] , refl))) ((fst (applyH refl ([] , refl)))))
  ≡⟨ cong {y = []} (λ x → fst (undo-merge (fst (applyH (p1 · p2) ([] , refl))) x)) (transportRefl _) ⟩
    fst (undo-merge (fst (applyH (p1 · p2) ([] , refl))) [])
  ≡⟨ cong {y = ADD "world" AT (suc zero) :: transport refl (ADD "hello" AT zero :: transport refl
          (λ x → fst (undo-merge x [])) (transportRefl _) ⟩
    fst (undo-merge (ADD "world" AT (suc zero) :: transport refl (ADD "hello" AT zero :: transport
  ≡⟨ cong {y = ADD "hello" AT zero :: []}
          (λ x → fst (undo-merge (ADD "world" AT suc zero :: x) [])) (transportRefl _) ⟩
    fst (undo-merge (ADD "world" AT (suc zero) :: (ADD "hello" AT zero :: [])) []) ■
open richer
```

Chapter 5

Conclusion

In this thesis we have constructed and outlined an implementation of homotopical patch theory in Cubical Agda. The implementation makes use of higher inductive types and univalence, and since the cubical model imbues univalence with computational meaning we are able to show that models of the theory behave as expected – at least for simple examples.

A full exploration of the behavior is (at the time of writing) limited by two factors: the efficiency of typechecking for complicated terms, and the fact that Cubical Agda does not fully reduce `transp` and `hcomp` for indexed families of types.

The former means that it is computationally expensive (and time consuming) to verify the behavior of the implementation, while the latter makes it impossible to compute results for the more complicated models. In particular for compositions of patches (using `hcomp`) when modeling the repository as a vector of strings (and indexed family of types).

Additionally, the first chapter contains an exposition of HoTT and (cubical) Agda and the second a survey of some approaches to a theory of VCS.

5.1 Future Work

There are two main avenues for future work. Firstly on the formalization of HPT and secondly on other type-theoretic approaches like the one discussed in section 3.3.

The HPT formalization also permits two directions of further inquiry. One is to implement more of the original paper. In particular the `indep` law for the patch theory with laws and all patch laws for the theory in section 6 are missing. Their inclusion would require a different way to map higher-dimensional paths into the universe which is guaranteed to terminate. The other is to work towards more computational results. Specifically we are limited by indexed families, and further work in Cubical Agda's normalization would lead to more results "for free". It would also be interesting to look at the computation of `opt`, as it requires some notions of sub-singletons [1].

Other type theoretic approaches could also be investigated. While HPT provides an elegant encapsulation of groupoid properties and separation of theories from models, it does not provide tools to reason about the semantics of patches and operations on them. In particular, a formal theory could incorporate merging and its properties. Another direction for expansion would be a more modular theory in which different patch theories (e.g for text files and integers) could be combined in a principled way to form something like a directory.

Bibliography

[1] Carlo Angiuli et al. "Homotopical patch theory". In: *Journal of Functional Programming* 26 (2016). ISSN: 14697653. DOI: 10.1017/S0956796816000198.

[2] Apache. *Apache Subversion*. 2022. URL: https://subversion.apache.org/.

[3] Cyril Cohen et al. "Cubical type theory: a constructive interpretation of the univalence axiom". In: *arXiv preprint arXiv:1611.02108* (2016). URL: https://arxiv.org/pdf/1611.02108.pdf.

[4] Thierry Coquand, Simon Huber, and Anders Mörtberg. "On higher inductive types in cubical type theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 255–264. URL: https://arxiv.org/pdf/1802.01170.pdf.

[5] *darcs*. 2022. URL: http://darcs.net/.

[6] Martin Hofmann and Thomas Streicher. "The groupoid interpretation of type theory". In: *Twenty-five years of constructive type theory (Venice, 1995)*. Vol. 36. Oxford Logic Guides. New York: Oxford Univ. Press, 1998, pp. 83–111.

[7] Robin Houston. *Revisiting "On Editing Text"*. 2014. URL: https://bosker.wordpress.com/2014/06/19/revisiting-on-editing-text/.

[8] Judah Jacobson. "A formalization of darcs patch theory using inverse semigroups". In: (2009). URL: ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf.

[9] Dagit Jason. *Type-correct changes — a safe approach to version control implementation*. 2009. URL: https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/47429f27z.

[10] Nicolai Kraus and Jakob von Raumer. "Path Spaces of Higher Inductive Types in Homotopy Type Theory". In: (2019). arXiv: 1901.06022 [math.LO].

[11] Ian Lynagh. "An algebra of patches". In: (2006). URL: http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf.

[12] Saunders Mac Lane. *Categories for the working mathematician. 1998*. Vol. 5. 1998.

[13] Per Martin-Löf. "An intuitionistic theory of types: Predicative part". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 1975, pp. 73–118.

[14] Samuel Mimram and Cinzia Di Giusto. "A Categorical Theory of Patches". In: *CoRR* abs/1311.3903 (2013). arXiv: 1311.3903. URL: http://arxiv.org/abs/1311.3903.

[15] Anders Mörtberg and Loïc Pujet. "Cubical synthetic homotopy theory". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 158–171.

[16] Russell O'Connor. *Git is Inconsistent*. http://r6.ca/blog/20110416T204742Z.html. Apr. 2011.

[17] Zooko O'Whielacronx. *badmerge – abstract version*. Jan. 2009. URL: https://tahoe-lafs.org/~zooko/badmerge/simple.html.

[18] *Pijul*. 2021. URL: https://pijul.org/.

[19] Egbert Rijke. *Introduction To Homotopy Type Theory*. Lecture Notes: https://hott.github.io/HoTT-2019/images/hott-intro-rijke.pdf. 2019.

[20] Ganesh Sittampalam. "Some properties of Darcs patch theory". In: (2005). URL: http://urchin.earth.li/darcs/ganesh/darcs-patch-theory/theory/formal.pdf.

[21] Agda development team. *Agda*. 2021. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[22] git development team. *git*. 2022. URL: https://git-scm.com/.

[23] Mercurial development team. *Mercurial*. 2022. URL: https://www.mercurial-scm.org/.

[24] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[25] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A dependently typed programming language with univalence and higher inductive types". In: *Journal of Functional Programming* 31 (2021). URL: https://staff.math.su.se/anders.mortberg/papers/cubicalagda2.pdf.

[26] Vladimir Voevodsky. “The equivalence axiom and univalent models of type theory.(Talk at CMU on February 4, 2010)”. In: *arXiv preprint arXiv:1402.5556* (2014).