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

April, 2022

Abstract

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Acknowledgements

Est suavitate gubergren referrentur an, ex mea dolor eloquentiam, novum ludus suscipit in nec. Ea mea essent prompta constituam, has ut novum prodesset vulputate. Ad noster electram pri, nec sint accusamus dissentias at. Est ad laoreet fierent invidunt, ut per assueverit conclusionemque. An electram efficiendi mea.

`Åsmund Aqissiaq Arild Kløvstad`
Friday 29$^{\text{th}}$ April, 2022

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Related works

In this section we summarize some key papers and their significance to the project.

1.1.1 Patch Theory (Darcs)

Here we discuss several proposed formalisms for a the patch theory employed by Darcs [4]. [8, 16, 6] all attempt to describe Darcs' patch theory. (focus on Lynagh, I think)

Lynagh [8] proposes an "algebra of patches" as a theoretical basis for the Darcs [4] 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:

$$\begin{tikzcd}
\bullet \arrow[r, "B"] & \bullet \\
\bullet \arrow[u, "A"'] \arrow[r, dashed, "B'"'] & \bullet \arrow[u, dashed, "A'"]
\end{tikzcd}$$

Figure 1.1: 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:

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 $AA^{-1} = A^{-1}A = Id$

$$\begin{tikzcd}
\bullet \arrow[r, "B"] \arrow[d, "A"'] & \{b\} \\
\{a\} \arrow[u, dashed, bend right, "A^{-1}"'] \arrow[r, dashed, "B'"'] & \{a,b\} \arrow[u, dashed, "A^{-1\prime}"]
\end{tikzcd}$$

Figure 1.2: Merging A and B by commutation

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 1.1 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 $B'A^{-1\prime}$

3. define merge(A,B) to be the sequence AB'.

This process is shown in Figure 1.2.

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 1.3) to obtain C' with the desired endpoints.

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

$$\begin{tikzcd}
\{a, b\} \arrow[r, "C"] & \{a, b, c\} \\
\{a\} \arrow[u, "B"] \arrow[r, dashed, "C'"'] & \{a, c\} \arrow[u, dashed, "B'"']
\end{tikzcd}$$

Figure 1.3: Commutation for cherry picking

1.1.2 A Categorical Theory of Patches

A Categorical Theory of Patches [10] 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 [5] 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 [13] (if so, figure out the relationship to [10])) (maybe some figures go here)

1.1.3 Homotopical Patch Theory

Homotopical Patch Theory [1] gives a formulation of patch theory in homotopy type theory. A patch theory is represented by a higher inductive type, and its interpretation by a function out of this type.

By representing repository state as points and patches as paths in a higher inductive type, the groupoid structure of the patch theory comes "for free". Paths come with composition, and by the groupoid laws this composition is associative, unital, and respects inverses. Additionally, functions (which are functors) respect this structure so any interpretation must also validate the groupoid laws.

Patch laws are represented by paths between paths (squares? disks? 2D-somethings). For example we may want the application of two independent patches to commute – this is done with a patch law.

While the HIT formulation gives a lot "for free", it also has some drawbacks. In particular, the requirement that all patches have inverses causes some problems. The workaround is to "type" patches with the history they are applicable to. This allows Angiuli et al. to define a merge operation in terms on only the "forward" patches, but leads to a fairly complex theory even for relatively simple settings.

An interesting feature of Angiuli et al.'s patch theories is that the type of repositories must be contractible. Since patches are represented by paths, any point can be retracted along them. As such, all repositories are – in a sense – "the same" and we need better notions of "sub-homotopical" [1] computations to reason about their differences.

1.1.4 Path Spaces of Higher Inductive Types

Path Spaces of Higher Inductive Types in Homotopy Type Theory [7] provides an induction principle for paths in coequalizers. This is extremely useful, since we want to define functions out of spans in HITs. (← rework this sentence)

Summarizing this will be very technical, and may become its own chapter if I successfully formalize the proof in cubical agda. Otherwise it goes here.

Chapter 2

Homotopy Type Theory

It's cool. [18]

The purpose of this section is to give the reader enough prerequisites to follow the ensuing development [pretentious af]. It follows, with numerous omissions, the development in Egbert Rijke's 2019 summer school [15]. [NOT REALLY, ANYMORE] For a more thorough treatment see [14] and for a complete textbook see The Book [18].

2.1 (Dependent) Type Theory

1. (dependent) types in computer science
2. type theories in math/foundations (the formal stuff)
3. Agda syntax?

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, Type Theory (tm) 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 [NOTE: THESE ARE JUDGEMENTAL EQUALITIES, DISTINCT FROM IDENTITY TYPES. MAKE CLEAR AND SETTLE ON SYNTAX]:

$$\Gamma \vdash A \; Type$$

"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 [15].

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 cartesian product $A \times B$ 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

1. types represent propositions (and spaces)

2. implication and simple and/or $(\to, \times, +)$

3. quantifiers and dependent types (fibers) (Σ, Π)

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 $\perp$ which has not terms, and the unit type $\top$ which has one term denoted by **1**. [MAYBE INTRODUCE THE TYPES FIRST]

Under the "types as propositions" interpretation, $\perp$ 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: **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 $\perp$ 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.

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

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)]

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 show that produce a proof (term) of $P(x)$. We use $\Pi_A P$ to represent this quantification. Note that both of these constructions quantify over some base type A, not over "every x in the universe," whatever that means. [REVISIT]

2.3 Types as Spaces

Another important model of types are as spaces. Pioneered by Voevodsky [CITATION? Mention the book?], his model views types as topological spaces and terms as points in those spaces. This view sheds a new light on identity types. A term in $x \equiv_A y$ is a path from x to y and the identity type is path space. Higher identity types also gain new meaning: paths between two paths are precisely the eponymous "homotopies"! Geometrically we visualize them as "filling in" the space between paths.

[WHAT MORE?]

2.4 Identity Types

1. what about things that are equal?
2. J-rule (intuition: reflexive closure? groupoid structure?)
3. paths in space

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 [9], 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 [15]:

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

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 it is enough to know how to use $refl$.

[MAYBE SOMETHING ABOUT THE GROUPOID STRUCTURE AND UIP]

One way to make sense of identity types is through the homotopy theory. With this interpretation 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. [REVISIT]

2.5 Higher Inductive Types

1. inductive types: base case(s) and point generator(s)
2. example(s)
3. HIGHER inductive types: terms and identities
4. ie. points and paths between points (and paths between paths (and paths between paths between paths))
5. example(s)
6. elimination rules? they need to go somewhere, but this might not be it

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:

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

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.

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

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 [15] (4.1)]

2.5.2 Higher Inductive Types

This doesn't really make sense without the interpretation of types as spaces, huh..

1. motivation: "natural" (sic) extension, synthetic topology
2. example: the circle
3. cubical and non-cubical elimination

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 TIME]

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 [17] – 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).

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 ⊣ 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 [I CANNOT DO THIS WITHOUT INTRODUCING MORE CATEORY THEORY. FOOTNOTE? Maclane IV.6: CCC's] 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.

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

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

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))
      :: []))
```

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)
```

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 2.4.

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 [2]. 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 [3]. 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 [11] (and in fact $\mathtt{I}$ may be described as the free de Morgan algebra on a discrete set of variable names $\{i, j, k...\}$ [2]).

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 [19] implements support for cubical type theory in Agda based on the development by Cohen et al. [2]. 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:

$$\mathsf{HPath} : (A : \mathsf{I} \to \mathsf{Type}) \to A\ \mathsf{i0} \to A\ \mathsf{i1} \to \mathsf{Type}$$

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

$$\mathsf{Id} : \{A : \mathsf{Type}\} \to A \to A \to \mathsf{Type}$$
$$\mathsf{Id}\ \{A\}\ x\ y = \mathsf{HPath}\ (\lambda\ \_ \to A)\ x\ y$$

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

$$\mathsf{refl} : \{x :\ A\} \to x \equiv x$$
$$\mathsf{refl}\ \{x = x\} = \lambda\ i \to x$$

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

$$\mathsf{sym} : \{x\ y :\ A\} \to x \equiv y \to y \equiv x$$
$$\mathsf{sym}\ p = \lambda\ i \to p\ (\tilde{}\ i)$$

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?

1. function extensionality
2. univalence (Glue types?)
3. HITs
4. all of the above with canonicity (with two very annoying exceptions)

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 [20].

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

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

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.

$$\begin{aligned}&\mathsf{funExt} : \{A\ B : \mathsf{Type}\}\ \{f\, g : A \to B\}\ (p : (x : A) \to f\, x \equiv g\, x) \to f \equiv g\\ &\mathsf{funExt}\ p\ i\ x = p\ x\ i\end{aligned}$$

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

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

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

$$\begin{aligned}&\mathsf{ua} : \{A\ B : \mathsf{Type}\} \to A \simeq B \to A \equiv B\\ &\mathsf{lineToEquiv} : \{A\ B : \mathsf{Type}\} \to A \equiv B \to A \simeq B\end{aligned}$$

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).

Chapter 3

Version Control Systems

They're not always cool. [12]

Version control systems are ubiquitous in software development, where they help facilitate cooperation and documentation of the development process. Their basic use is to record (*commit*) changes to a codebase (*repository*). Systems may also include ways for the codebase to diverge (*branch*) into different versions, and ways to reunite (*merge*) these versions.

3.1 Background

This section introduces the terminology and overall structure of version control systems for use in the ensuing work.

At the very least we need to define:

- repository
- patch
- merge

Might also be nice:

- branch?
- cherry pick?

3.2 Our idea [BETTER TITLE?]

This section introduces the problems with current approaches and the idea of our proposed solution.

Basically:

1. requirements
   - repo - accurately represents contents
   - patch - applicable in a context, groupoid structure
   - merge - "pushout property"/reconcile, symmetric (for distributed systems), (do we need associativity as well?)
2. hopes/goals
   - repos - modular/composable, somehow polymorphic
   - patches - *semantic* in some sense
   - merge - easily definable [sic.], considers semantics of patches

3.3 Results

This section describes our solution and why it doesn't work.

Basically: equivalences of equivalences is complicated, maybe hpt had a point about reversibility

Chapter 4

Formalization

This chapter describes my formalization(s) of hpt in Cubical agda.

4.1 Agda Code

Discussion of implementation choices and difficulties.

- cubical stuff
- pathover/transport lemmas
- concrete injection/homomorphism in Simple
- set-truncation in With-Laws

4.2 An Elementary Patch Theory

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

4.2.1 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 `suc`.

```
open import Cubical.HITs.S1.Base
  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
```

4.2.2 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 {f1} {f2} {g1} {g2} p = let f1=g2 = cong snd p
                                      g1=f2 = cong fst (sym p) in
  f1 · g1 ≡⟨ cong₂ _·_ f1=g2 g1=f2 ⟩ g2 · f2
          ≡⟨ patch-comm g2 f2 ⟩
  f2 · g2 ∎
```

4.3 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.

4.3.1 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 4.3.2.

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) (permute
```

For the `noop` law we need to show that `swapat` respects it. We proceed in two steps. First `swassId` 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.

```
permuteId : {s : String} → (t : String) → permute (s , s) t ≡ id t
permuteId {s} t with t =? s — t =? s
... — yes t=s — yes _ = sym (ptoc t=s)
... — yes _    — no _  = refl
```

```
... | no t≢s | yes t≡s = ⊥-elim (t≢s t≡s)
... | no _   | no _ = refl

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) (funExt 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)
```

4.3.2 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 →` $\Sigma_{(\mathtt{y}\ :\ \mathtt{R})}$ `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 $\Sigma_{\mathtt{q}\ :\ \mathtt{Patch}}$ `(transport` $^{x \mapsto (x \equiv \mathsf{doc})}$ `p) ≡ 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)
```

$$\equiv\langle \text{cong } (\lambda\ x \to \text{Path } (\Sigma[\ y \in \text{R}\ ]\ y \equiv \text{doc})\ x\ (\text{doc}\ ,\ \text{refl}))\ (\Sigma\text{PathP } (\text{refl}\ ,\ \text{sym } (\text{lUnit } p)))\ \rangle$$
$$\text{Path } (\Sigma[\ y \in \text{R}\ ]\ y \equiv \text{doc})\ (\text{doc}\ ,\ p)\ (\text{doc}\ ,\ \text{refl})$$
$$\equiv\langle \text{sym } \Sigma\text{Path}{\equiv}\text{Path}\Sigma\ \rangle$$
$$(\Sigma[\ q \in \text{Patch}\ ]\ (\text{PathP } (\lambda\ i \to q\ i \equiv \text{doc})\ p\ \text{refl}))$$
$$\equiv\langle \Sigma\text{-cong-snd } (\lambda\ q \to \text{PathP}{\equiv}\text{Path } (\lambda\ i \to q\ i \equiv \text{doc})\ p\ \text{refl})\ \rangle$$
$$(\Sigma[\ q \in \text{Patch}\ ]\ (\text{transport } (\lambda\ i \to q\ i \equiv \text{doc})\ p) \equiv \text{refl})$$

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?]

$$(\Sigma[\ q \in \text{Patch}\ ]\ (\text{transport } (\lambda\ i \to q\ i \equiv \text{doc})\ p) \equiv \text{refl})$$
$$\equiv\langle \Sigma\text{-cong-snd } (\lambda\ q \to \text{cong } (\_{\equiv}\ \text{refl})\ (\text{path-transport-lemma } q\ p))\ \rangle$$
$$(\Sigma[\ q \in \text{Patch}\ ]\ (\text{sym } q \cdot p) \equiv \text{refl})$$

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

$$(\Sigma[\ q \in \text{Patch}\ ]\ (\text{sym } q \cdot p) \equiv \text{refl})$$
$$\equiv\langle \Sigma\text{-cong-snd } (\lambda\ q \to \text{invLUnique } q\ p)\ \rangle$$
$$(\Sigma[\ q \in \text{Patch}\ ]\ p \equiv q)\ \blacksquare$$

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

$$\text{invLUnique} : \{X : \text{Type}\}\ \{x\ y : X\} \to$$
$$(p\ q : x \equiv y) \to (\text{sym } p \cdot q \equiv \text{refl}) \equiv (q \equiv p)$$

This is, in fact, an equivalence since it relies on `compl≡Equiv : ∀ p q r → (q ≡ r) ≃ (p · q ≡ p · r)`, which is made into a path with `ua`.

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

$$\text{optimize} : (p : \text{Patch}) \to \Sigma[\ q \in \text{Patch}\ ]\ p \equiv q$$
$$\text{optimize } p = \text{transport e } (\text{cong opt } p)$$

[1] For the category theorist: this is the functorial action of the contravariant hom-functor [18]

4.4 Computational Results

This section discusses the computational properties of my formalization.

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

Chapter 5

Conclusion

We did some things and they worked out — or maybe they didn't.

Bibliography

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

[2] 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.

[3] 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.

[4] *Darcs*. 2021. URL: http://darcs.net/.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] Egbert Rijke. "Homotopy Type Theory". In: (2012). URL: `https://hottheory.files.wordpress.com/2012/08/hott2.pdf`.

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

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

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

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

[19] 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`.

[20] 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).

Appendix A

This is an appendix, if need be