University of Bergen
Department of informatics

A Cubical Implementation of Homotopical Patch Theory

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[1].

Additionally the first chapter contains an approachable introduction to HoTT and Cubical Agda aimed at an audience of interested computer science students covering dependent Martin-Löf-style type theory, propositions as types, univalent foundations, higher inductive types and and CCHM cubical type theory.

Finally, we discuss some other approaches to a theory of version control systems in Darcs' "algebra of patches" and an unsuccessful attempt to model repositories in type theory as coequalizers.

[1] Available at `https://github.com/Aqissiaq/hpt-experiments`.

Acknowledgements

Åsmund Aqissiaq Arild Kløvstad
Monday 30th May, 2022

Contents

1 Introduction and Background **1**

1.1 Homotopy Type Theory 2
1.2 (Dependent) Type Theory 3
1.3 Propositions as Types 9
1.4 Identity Types 11
1.5 Spaces as Types 14
1.6 Higher Inductive Types 16
1.6.1 Inductive Types 16
1.6.2 Higher Inductive Types 17
1.7 Agda 20
1.8 Cubical Type Theory 24
1.8.1 Cubical Agda 25
1.8.2 Function Extensionality and Univalence 28
1.8.3 Canonicity 29

2 Version Control Systems **31**

2.1 Background 32
2.1.1 Basic Ingredients 32
2.1.2 (Groupoid) Properties of Patches 32
2.1.3 Merging 33
2.2 Theoretical Approaches to VCS 35
2.2.1 Darcs 35
2.2.2 Homotopical Patch Theory 38

3 Formalization **43**

3.1 An Elementary Patch Theory 44

3.1.1 The Circle as a Repository . . . 44
3.1.2 Merge . . . 45
3.2 A Patch Theory With Laws . . . 47
3.2.1 The Patch Theory . . . 47
3.2.2 A Patch Optimizer . . . 50
3.3 A Patch Theory With Richer Contexts . . . 54
3.3.1 The Type of Repositories . . . 54
3.3.2 A Merge Function . . . 55
3.4 Computational Results . . . 58
3.4.1 Elementary Patch Computations . . . 58
3.4.2 Patch Computations with Laws . . . 60
3.4.3 Patch Computations with Richer Contexts . . . 62

4 Conclusion **67**
4.1 Discussion . . . 68
4.2 Future Work . . . 69

Bibliography **72**

A Another Type-Theoretic Approach **75**
A.1 Repository HIT . . . 75
A.2 Merge . . . 77
A.3 Result/Discussion . . . 78

Chapter 1

Introduction and Background

This thesis is about Homotopy Type Theory (HoTT), its interpretation and use in the Cubical Agda proof assistant, and an application to version control systems (VCSs). It is accordingly organized into three parts.

The remainder of this chapter contains an introduction to the Homotopy Type Theory setting, a cubical interpretation of this type theory, and an introduction to the syntax and workings of the Agda programming language [29] in general and Cubical Agda [33] in particular.

Chapter 2 gives and exposition of Version Control Systems (VCS) and some approaches to a theory of such systems. We give an account of Darcs' [6] "Algebra of Patches" as described by Lynagh [17] and Angiuli et al.'s "Homotopical Patch Theory" [1] which utilizes HoTT.

Chapter 3 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 3.4 we examine the implementations by testing them on simple examples. We conclude that the theories and models behave as expected, but are restricted by the current limitations of Cubical Agda – in particular that `transp` and `hcomp` do not compute over inductive families.

Finally, chapter 4 discusses the results of the formalization and possible directions for future work on homotopical path theory in particular and HoTT models of version control in general.

1.1 Homotopy Type Theory

The purpose of this introduction is to give the reader the prerequisites to follow the formalization in chapter 3. For an excellent in-depth introduction see Egbert Rijke's 2019 summer school notes [26]. The canonical text is The Book [32].

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.

1.2 (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 different from the others: ['a','b','c'] is a list *of* ***`char`****s*. 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 int 3` – lists of exactly 3 integers – and `Vec [char] 4` – lists of exactly 4 lists of characters. Again `Vec` is actually a whole collection of types – one for each choice of type and integer.

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

We now leave the familiar world of programming behind and venture in to the exciting world of foundational mathematics. In this new and wondrous world, a type theory is a system of *inference rules* like 1.1 that can be used to make *derivations*.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f(a) : B} \tag{1.1}$$

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:

$$\Gamma \vdash A \ \text{Type}$$

"A is a type."

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

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

$$\Gamma \vdash A \equiv B \ \text{Type}$$

"A and B are judgementally equal types."

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

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

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:

$$\frac{\Gamma \vdash A \ \text{Type} \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \texttt{Vec}\ A\ (n)\ \text{Type}}$$

Like lists, vectors are parametrized by another type, but unlike lists they also depend on a natural number – their length.

The preceding introductions of lists and vectors are clearly not complete specifications of the types. They do not tell us what the terms of look like, 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.

$$\frac{\Gamma \vdash A \ \text{Type} \qquad \Gamma \vdash B \ \text{Type}}{\Gamma \vdash A \to B \ \text{Type}} \tag{1.2}$$

An *introduction rule* (1.2) tells us how to construct the type. In this case, if A and B are types, then functions between them is also a type.

$$\frac{\Gamma, a : A \vdash f(a) : B}{\Gamma \vdash \lambda x.f(x) : A \to B} \tag{1.3}$$

A *formation rule* (1.3) 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.

$$\frac{\Gamma \vdash f : A \to B}{\Gamma, a : A \vdash f(a) : B} \tag{1.4}$$

An *elimination rule* (1.4) 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.

$$\frac{\Gamma, a : A \vdash f(a) : B}{\Gamma, a : A \vdash (\lambda y.f(y))(a) \equiv f(a) : B} \tag{1.5}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) \equiv f : A \to B} \tag{1.6}$$

Computation rules postulate when two terms are judgementally equal. In the case of functions we have two: β-reduction (1.5) and η-reduction (1.6). Taken together (and eliding any complications of variable substitution), they show that function evaluation and lambda abstraction are mutual inverses [26].

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 $A \to B$. First we look at the type of dependent pairs.

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

The introduction and formation rules tell us 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 $\Sigma_A B$

In a dependent setting we call the most general elimination rule an "induction principle". Such a principle describes how to construct a term in a family over the type we are interested in. For dependent pairs, it looks like this:

$$\frac{\begin{array}{l}\Gamma, x : \Sigma_A B \vdash P \ \text{Type} \\ \Gamma, a : A, b : B \ a \vdash p_{a,b} : P(a,b)\end{array}}{\Gamma, x : \Sigma_A B \vdash \text{ind} \ \ p_{a,b} \ x : P \ x}$$

In words: if, given an $a : A$ and $b : B(a)$ we have a term of type $P \ (a, b)$, then given an x in $\Sigma_A B$ we can construct a term of type $P \ x$. Note that the term $p_{a,b}$ depends on the given a and b, and ind $\ p_{a,b} \ x$ depends on both $p_{a,b}$ and x.

The computational rule associated with the above induction principle postulate that the result applying the induction rule to a pair (a, b) is the dependent term $p_{a,b}$.

$$\frac{\begin{array}{l}\Gamma, x : \Sigma_A B \vdash P \ \text{Type} \\ \Gamma, a : A, b : B \ a \vdash p : P(a,b)\end{array}}{\Gamma, a : A, b : B \ a \vdash \text{ind} \ \ p \ (a, b) \equiv p}$$

The analogous rules for dependent functions are:

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

$$\frac{\Gamma \vdash f : \Pi_A B}{\Gamma, x : A \vdash f(x) : B(x)}$$

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

Normalization

One important property of this kind of type theory is *normalization*. Because the derivations are made up from introduction rules, and elimination is justified by computation rules all terms reduce to a "normal form". One important consequence is that type-checking (the task of deciding whether a term is of a certain type) is decidable. Another consequence is *canonicity*: that any closed term of type $\mathbb{N}$ normalizes to a numeral [32].

These properties have important consequences for the interpretations presented in section 1.3 and section 1.5, and explain the usefulness of type theory in proof assistants.

1.3 Propositions as Types

In this section we consider an important interpretation of type theory: the Curry-Howard Correspondence.

Under this correspondence 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 no terms, and the unit type $\top$ which has one term denoted by $\mathbf{1}$. 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 the 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 type $A \times B$ with A and B ranging over $\top$ and $\bot$. Its logical counterpart $A \wedge B$ is true precisely when both A and B are true, and indeed if $A = B = \top$ we can construct the term $(\mathbf{1}, \mathbf{1}) : A \times B$. Conversely, if A (resp. B) is $\bot$ we cannot construct a pair since there are no terms to put on the left (resp. right) hand side of the pair.

As another example, what does it mean to prove an implication $A \to B$? One reasonable answer is that given a proof of A, we 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, we mention that logical "or" is represented by the sum type (disjoint union) $A + B$ and negation ($\neg$) by a function into $\bot$.

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 set is a proposition that *depends* on an element of it, then it stands to reason that a predicate can be represented by a type that depends on a term. As such, we view a term of the type $B(x)$ as a proof that the proposition represented by B holds for the term x.

Extending this thinking to quantifiers, a proof of $\exists x.P(x)$ should consist of some $x : A$ 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 whenever a $x : A$ is given, we 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 universal 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". This is usually left implicit in FOL.

The Curry-Howard correspondence also elucidates why dependent elimination rules are called "induction principles." Viewing a family of types as a predicate, an induction rule for some type tells us precisely how to prove that the predicate is true for all terms of that type.

Constructivity

Notably, the propositions-as-types logic is *constructive*. This is a consequence of the type theory's normalization property and means that any proof necessitates constructing a proof term. For example, a proof of existence must explicitly construct the thing that exists as discussed above.

Conversely there are some classical propositions that are not inhabited when considered as types. Most prominently the law of excluded middle $(A + \neg A)$ cannot be inhabited for every A. After all, we would have to know whether A is inhabited or not before even deciding whether to form an A or a $\neg A$.

1.4 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 *propositional* equality is different from the *judgemental* equality discussed in section 1.2. While judgemental equality is part of the inference rules of the type theory, the propositional equality of two terms is a type that can be inhabited and whose terms can be used.

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 [20], 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 [26]:

$$\frac{\Gamma \vdash a : A}{\Gamma, x : A \vdash a =_A x \ \text{Type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{refl}_a : a =_A a}$$

and an elimination rule (called the induction principle or simply the J-rule) given by:

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma, x : A, p : a =_A x \vdash P\ x\ p\,\text{Type} \\ \Gamma, a : A \vdash J_a : P\ a\ \text{refl}_a\end{array}}{\Gamma, x : A, p : a =_A x \vdash P\ x\ p}$$

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

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. 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 $P(a, \text{refl}_a)$. 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 $\text{refl} \cdot \text{refl}$ is also refl.

Armed with the J-rule many useful functions on identity types can be constructed. We mention two closely related functions:

The first is cong f (called ap_f in [**lemma 2.2.1**, 32]) which for some function $f : A \to B$ maps identity proofs $a = a'$ in A to identity proofs $f\ a = f\ a'$ in B. By the J-rule we may assume that the path in question is refl_x and define cong f $\text{refl}_x := refl_{f\ x}$.

The second is a dependent version of cong called transport. Given a family P over A and an identification $p : a =_A a'$, $\text{transport}^P\ p$ gives a function $P\ a \to P\ a'$. In the case where p is $refl_x$ it is the identity function $P\ x \to P\ x$.

UIP and The Groupoid Structure of Types

A question one might ask is "can there be more than one proof of identity?". The negative 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 necessarily hold for every type?" In 1995 Hofman and Streicher [9] showed that the answer is "no" by constructing a model in which it fails to hold.

So an identity is not merely a type which either does or does not have an inhabitant, but does it have more structure? Hofman and Streicher's model give an 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

[1] In HoTT we say A *is a set* or has H-level 0.

units and inverses. That is, for proofs $p : x = y$, $q : y = z$, $r : z = w$ the following types are all inhabited:

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

$$p \cdot \mathrm{refl}_y =_{x=y} p$$

$$\mathrm{refl}_x \cdot p_{x=y} = p$$

$$p \cdot p^{-1} =_{x=x} \mathrm{refl}_x$$

$$p^{-1} \cdot p =_{y=y} \mathrm{refl}_y$$

This *groupoid structure* of identity proofs will be very important in chapter 3.

1.5 Spaces as Types

Another (related) way to make sense of identity types is through homotopy theory. With this interpretation, pioneered by Voevodsky [34] and the HoTT program [32], a term of $x =_A y$ is 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.

Voevodsky's Model

We have already seen a model that lets us view types as groupoids. When also considering higher paths, types have the structure of "weak higher-dimensional" groupoids. A classical example of weak higher-dimensional groupoids can be found in homotopy theory, where Kan complexes are a specific kind of "well behaved" space. This was observed by both Voevodsky and Streicher around 2006 [28] and Voevodsky used the insight to construct a model for Martin-Löf type theory.

Briefly, the model takes a combinatorial view of spaces by mapping them to *simplicial sets* – a sort of arbitrary-dimensional triangulation. To accommodate dependent types, we only consider Kan complexes – intuitively the simplicial sets where every triangle can be filled in.

The space containing all the basic spaces also forms a Kan complex called a *Universe* and often denoted by a $\mathcal{U}$. Importantly the universe does not contain itself, but may be contained in another universe "one level up". If necessary, a cumulative hierarchy of universes containing all the lower ones is possible, but Voevodsky settles for two. This is a very brief overview, a full exposition by Kapulkin and Lumsdaine is found in [11].

Univalence and Canonicity

Voevodsky also shows that his model satisfies the *univalence property*. Loosely speaking this property says that the identity type for the universe is equivalent to the type of equivalences. In other words, equivalent types can be identified.

This property aligns well with mathematical practice where structures are often considered up to some suitable notion of equivalence. With well defined types, univalence means groups can be identified when they are isomorphic, and natural numbers can be identified only when they are equal. Consequently HoTT provides a setting for mathematics with desirable properties; the "Univalent Foundations" program as introduced in The Book [32] aims to do mathematics in this setting.

Since the univalence property holds in Voevodsky's model, we might justifiably want to include it in our type theory and The Book does exactly that by including the following axiom (where idToEquiv is a function of type $(A =_{\mathcal{U}} B) \to (A \simeq B)$):

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash \text{univalence } A\ B : \text{isEquiv}(\text{idToEquiv } A\ B)}$$

The inverse of idToEquiv is ua, with the computation rule that transport along $\text{ua}(f)$ is (propositionally) equal to applying f.

However this poses a problem: when extending the type theory with an axiom, the canonicity property is lost – terms no longer necessarily reduce to a normal form and we lose the nice constructive properties of the resulting logic. The constructive/computational meaning of the univalence axiom in homotopy type theory is an open area of research, and we will return to it in section 1.8.

1.6 Higher Inductive Types

Now that we have several useful interpretations of types, we need some ways to construct more complex types to study. In this section we look at one such way: Higher Inductive Types (HITs). HITs are motivated by the interpretation of types as spaces. In this section we first introduce regular inductive types, then their "higher" counter-part and finally an application of HITs serving as a concrete example.

1.6.1 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:

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

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. For this reason we may refer to the introduction rules as "constructors" of $[A]$ and say that lists of A are "generated" by its two constructors.

In order to use this type, we also need an induction principle. The induction principle tells us how to use terms of the type by defining functions into a family over it. If the family is constant (i.e. the function is non-dependent) we call such a rule a "recursion principle".

$$\frac{\begin{array}{l}\Gamma, l : [A] \vdash P\ l \ \text{Type} \\ \Gamma \vdash p_{[\,]} : P\ [\,] \\ \Gamma \vdash p_{::} : \Pi_{a:A}\Pi_{as:[A]} P\ (a :: as)\end{array}}{\Gamma \vdash ind_{[A]}\ p_{[\,]}\ p_{::} : \Pi_{[A]} P}$$

This rule states that a dependent function out of $[A]$ can be constructed by knowing what it does on the empty list, and – given an $a : A$ and a list $as : [A]$ – what it does on $(a :: as)$. It is not a coincidence that we require two terms – they correspond precisely to the two constructors.

The resulting function maps the empty list to $p_{[\,]}$ and any arbitrary list $(a :: as)$ to $p_{::}$ by the computation rules:

$$\frac{\Gamma \vdash p_{[\,]} : P\ [\,] \qquad \Gamma \vdash p_{::} : \Pi_{a:A}\Pi_{as:[A]}P\ (a :: as)}{\Gamma \vdash ind_{[A]}\ p_{[\,]}\ p_{::}\ [\,] \equiv p_{[\,]}} \qquad \frac{\Gamma \vdash p_{[\,]} : P\ [\,] \qquad \Gamma \vdash p_{::} : \Pi_{a:A}\Pi_{as:[A]}P\ (a :: as)}{\Gamma, a : A, as : [A] \vdash ind_{[A]}\ p_{[\,]}\ p_{::}\ (a :: as) \equiv p_{::}\ a\ as}$$

1.6.2 Higher Inductive Types

When constructing ever more complicated types, it is useful to have some control over which terms are identified. For example we might want to construct a quotient $A/_\sim$ where related terms are identified.

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 identity proofs. We will often use the language of Homotopy Theory and refer to the terms and identity proofs in a HIT as "points" and "paths" respectively.

There is no reason to stop there! We could apply the same idea to give generators for paths between paths, paths between paths between paths and so on. In section 3.2 we will make use of such "higher paths", but for now let us focus on a simple example.

The Circle

Utilizing the language of spaces we introduce a very simple HIT: the circle S^1. This is a HIT with one point constructor (base) and one path constructor (loop). Its introduction and formation rules are:

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

The circle also comes with a recursion principle [2]. Like the induction principle for lists, a function out of the circle requires one term for each constructor. A point p for the base, and a path from p to itself for the loop.

$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash l : b =_B b}{\Gamma \vdash rec_{S^1}\ b\ l : S^1 \to B}$$

The computation rules state that the resulting function maps base to b and applying it around loop gives l. Note that the computation rule for loop does not give a judgemental equality, but rather postulates a term called loopComp witnessing the result.

$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash l : b =_B b}{\Gamma \vdash rec_{S^1}\ b\ l\,\text{base} \equiv b} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash l : b =_B b}{\Gamma \vdash \text{loopComp} : \text{cong}(rec_{S^1}\ b\ l)\,\text{loop} = l}$$

In homotopy type theory as presented in The Book, HITs are included by adding their formation, introduction, elimination and computation rule(s) as axioms. Like the univalence axiom, this means their inclusion interferes with the computational properties of the theory and once again we lose canonicity. We will look at one possible remedy in section 1.8.

Synthetic Homotopy Theory

An important use of HITs is synthetic homotopy theory [22, 14]. By constructing representation of spaces as HITs it is possible to formalize results of homotopy theory very directly.

Here we recount a proof that the fundamental group of the circle is $\mathbb{Z}$ originally given by Licata and Shulman [16]. This showcases the use of circle recursion and also proves useful in section 3.1.

The fundamental group of a space X (with base point x_0) $\pi_1(X, x_0)$ is the group whose elements are loops at x_0 and whose operation is path concatenation. In type theoretic terms we define the loop space (at x_0) ΩX as the identity type $x_0 =_X x_0$. Path concatenation is simply the transitive property of identity proofs.

The proof that $\pi_1(S^1, \text{base}) \simeq \mathbb{Z}$ has four parts:

[2] The fully general induction principle requires a notion of dependent paths "over" loop. See section 6.2 in The Book [32]

1. a function winding : $\Omega S^1 \to \mathbb{Z}$
2. a function intLoop : $\mathbb{Z} \to \Omega S^1$
3. a proof that winding and intLoop are mutual inverses
4. a proof that either winding or intLoop is a group homomorphism (it suffices to show one direction)

We limit ourselves to (1) to showcase circle recursion, but the full details are spelled out in section 8.1 of The Book [32] an repeated by Mörtberg and Pujet [22].

Since we do not have a recursion principle for ΩS^1 we first construct the covering space of the circle called helix. This is a family over S^1 which can be viewed as a function from S^1 to the universe, so we use circle recursion to define:

$$\text{helix} := rec_{S^1}\ \mathbb{Z}\ (\text{ua}\,\text{succEquiv})$$

Where succEquiv is the equivalence $\mathbb{Z} \simeq \mathbb{Z}$ induced by the successor function.

The winding number is computed by transporting in the helix. Given some $x : S^1$ and path $p : \text{base} = x$, define:

$$\text{encode}(p) := \text{transport}^{\text{helix}}\ p\ 0$$

and finally

$$\text{winding} := \text{encode}\,\text{base}$$

Transporting along loop in helix is equivalent to transporting along cong helix loop in the identity family, which by loopComp is identified with transport^{id} (ua succEquiv). Computation rules for ua give the final result: the successor function.

The other direction could be defined by simple recursion on the integers, but this makes it difficult to prove (3). Instead, Licata and Shulman define a decode-function analogous to helix such that intLoop is decode of base.

This technique – making use of the recursion principle to define a function on all of S^1 before specializing to base – is known as an "encode-decode" style of proof.

1.7 Agda

In this section we introduce Agda [29] – a dependently typed programming language and proof assistant. The goal is to introduce enough of its syntax and workings to follow the formalization in chapter 3.

The basic syntax of Agda will be familiar to users of Haskell [19], but with `:` for typing and significant use of unicode (including → for function types). In general, terms will appear as `term : Type` followed by `term = ...` where the first line provides the type and the second defines the specific term.

As an example, we consider the type of vectors and operations on them. This is a dependent type that provides a good look at the syntax and features of Agda as a programming language.

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)` is a vector – and are denoted by underscores. Each underscore in the name represents a position in which we may place the corresponding argument.

Currying means that a function like `_::_` that takes two arguments of types `A` and `Vec A n` and produces a `Vec A (suc n)` can be written as `_::_ : A → Vec A n → Vec A (suc n)` (with → associating to the right). [3].

This style allows for partial application of functions where `_::_ x` results in a unary function `Vec A n → Vec A (suc n)`. 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)
```

[3] This is possible because of the product ⊣ exponentiation adjunction in cartesian closed categories which gives a bijection between $(A \times B) \to C$ and $A \to (B \to C)$ for all objects A, B and C See IV.6: Cartesian Closed Categories in [18]

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

```
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.

```
_++_ : {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 1.2. Rather, it is the identity type described in section 1.4.

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.

```
+-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) ■
```

1.8 Cubical Type Theory

The computational properties of type theory as a foundation is what makes a proof assistant like Agda possible. However, we have seen that adding axioms to the theory breaks these properties. That poses a problem when there are axioms we would like to make use of – in particular the univalence axiom and higher inductive types.

One way to imbue HoTT with computational meaning is Cubical type theory [4]. The basic idea is to take the "spaces as types"-interpretation of identity types very literally, as a function from the interval. With this interpretation, univalence and HITs do not need to be added axiomatically – they become provable theorems [5]. This section introduces the basic concepts of cubical type theory, Cubical Agda and the Cubical library. In particular we discuss the theory emplyed by Cubical Agda: the CCHM (Cohen, Coquand, Huber and Mörtberg) cubical type theory [4].

The main ingredient of cubical type theory is the interval type. It represents the closed interval $[0, 1]$ in and we can roughly think of it as a HIT with two points and a path 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 [22] (and in fact $\mathtt{I}$ may be described as the free de Morgan algebra on a discrete set of variable names $\{i, j, k...\}$ [4]).

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 from $\mathtt{I}$ we obtain the inference rules seen in Figure 1.1.

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

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

Figure 1.1: Introduction-, formation- and elimination-rules for cubical paths

call the A's which permit such a mapping of cubes "cubical sets" and use them to model types in our theory.

Composition of paths is slightly involved. The most natural notion of composition is actually ternary because it corresponds to "putting a lid" on an open box. Given paths $p : x = y$, $q : y = z$ and $r : z = w$ the ternary composition is the dotted line in Figure 1.2.

This operation is called `hcomp` (homogenous composition). In order for hcomp to be well defined, it must be possible to put a lid on every such open box. This is precisely the "Kan condition" on cubical sets, so types are modelled by *Kan* cubical sets. (Recall that Voevodsky's model used Kan simplices).

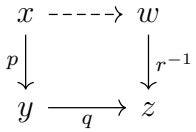

Figure 1.2: Composition of p, q, r

Note that the right wall is inverted to be parallel with the left. To obtain binary composition $p \cdot q$ we fill in the box where the right-hand wall is refl_z (it does not actually matter which wall we choose).

1.8.1 Cubical Agda

Cubical Agda [33] implements support for cubical type theory in Agda based on the development by Cohen et al. [4]. 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 section 1.4 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 introduced in section 1.6.

```
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) = sym loop i
```

This is very much like rec_{S^1}. In order to define a function we require a point (`base`) and loop (`sym loop`) at that point. Since paths in Cubical agda are functions from the interval, the loop also includes an argument `i` which we supply to `sym loop`, representing travelling along the path.

Similarly, let us define the helix and winding number from section 1.6.

```
helix : S¹ → Type
helix base = ℤ
helix (loop i) = ua succEquiv i


encode : (x : S¹) → base ≡ x → helix x
encode _ p = subst helix p (pos 0)

winding : base ≡ base → ℤ
winding = encode base
```

Since everything computes, we do not need to evoke any computation rules to show that this function computes the winding number. Each case is witnessed directly by refl.

```
_ : winding loop ≡ 1
_ = refl

_ : winding (loop · loop) ≡ 2
_ = refl

_ : winding (sym loop) ≡ (- 1)
_ = refl
```

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

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

1.8.2 Function Extensionality and Univalence

In addition to the higher inductive types, a 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 [34].

In CCHM type theory (and therefore in Cubical Agda) these are not axioms at all, but provable theorems. Function extensionality is especially straightforward: given two 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.

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

It is also possible to prove function extensionality from univalence [8, 3], but the above is much more direct.

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.

Recall that types are modelled by Kan cubical sets which permit filling in the final side of any open box. Is the universe also such a type? The answer informs what paths in the universe looks like, so this is where univalence shows up. In Cubical Agda it shows up in the form of a new type former called `Glue`.

Conceptually, `Glue` provides a way to construct lids of open boxes in the universe given a family of types and equivalences over `I`. We may think of it as a generalization of composition which allows a family of equivalences, rather than a family of paths [13].

In order to define `ua` of some equivalence e we let the left wall be e, the bottom refl and the right the identity equivalence. Then Glue closes the box providing the desired path (Figure 1.3)

The result is one way of the equivalence above. The other direction is called `lineToEquiv`. It is easily obtained by transporting along the path.

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

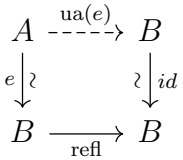

Figure 1.3: $ua(e)$ in terms of Glue

1.8.3 Canonicity

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

However, that is not entirely true. There are two exceptions to canonicity at the time of writing:

1. `transp` (the primitive used to implement `transport`) over inductive families, and
2. `hcomp` (the primitive used to implement path composition) over inductive families.

Inductive families refer to inductive types that are also indexed by some indexing type. For example, `Vec A` is an inductive family over `A` indexed by integers, and as such expressions like `transport (λ i → Vec A (p i)) [ ]` do not reduce [33]. Canonicity-preserving support for inductive families and its inclusion in Cubical Agda is an ongoing project [4]. We will return to the issue in chapter 4, as it relates to the formalizations and results in chapter 3.

[4] https://github.com/agda/agda/issues/3733

Chapter 2

Version Control Systems

A version control system (VCS) is a software development tool used to keep track of data and changes to it – often the data is the code for a software project. 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.

By far the most widely used tool is the distributed git [30] which employs a *snapshot*-model. In this approach the system stores the state of the repository, and computes patches in terms of line-by-line differences between snapshots.

The distributed Mercurial [31] and centralized Subversion [2] work by similar models. These tools are fast, but often poorly understood and have some un-intuitive behavior [23, 24].

An alternative to snapshot-models is a *patch*-model. VCS with this model instead store the set of patches explicitly, and then computes the repository state from them. The most prominent example of such a system is Darcs [6] with its "algebra of patches", but we also mention Pijul [25] which takes a more categorical approach.

In this chapter we introduce the basic concepts and terminology, survey the patch theory of Darcs as presented by Lynagh [17] and finally explore "homotopical patch theory" [1] – an approach to VCS using homotopy type theory and univalence.

2.1 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 [6, 17, 27, 10] with groupoid properties as in Homotopical Patch Theory [1].

2.1.1 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.

2.1.2 (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.

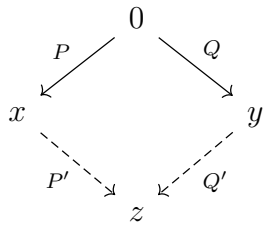

Figure 2.1: Merging (P, Q)

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 2.2.2 and the following implementation in chapter 3.

2.1.3 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 2.1).

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*.

2.2 Theoretical Approaches to VCS

In this section we introduce two proposed theoretical models of version control systems. These are the patch theory of Darcs [6, 17] and Angiuli et al.'s "homotopical patch theory" [1] (HPT).

The former is presented as an example of theoretical approaches to version control system frameworks, while the latter forms the basis of the formalizations making up the rest of the thesis.

2.2.1 Darcs

Lynagh [17] proposes an "algebra of patches" as a theoretical basis for the Darcs [6] 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 adding the update c to 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 $\{a, b\}P\{a, b, c\}Id\{a, b, c\}$ – adding 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:

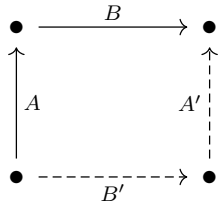

Figure 2.2: Commuting patches

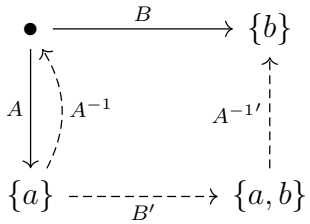

Figure 2.3: Merging A and B by commutation

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(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 2.2 if we want)
4. Circular (3.5): 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 2.3.

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

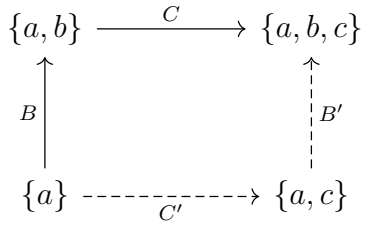

Figure 2.4: Commutation for cherry picking

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 2.4) to obtain C' with the desired endpoints.

One immediate problem is that patches cannot always be commuted. Lynagh's solution is a new type of patch called a *conflictor* which represents a conflict between two patches, and whose effect is to "undo" the changes that cause a conflict.

However, conflictors do not entirely solve the problem. In particular they are insufficient to fill certain gaps, because while conflicting is a symmetric relation, dependencies (often the cause of a conflict) are not [7].

2.2.2 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 3, 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 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. For example we may interpret `num` as the type of booleans and `add1` as negation to obtain a theory of a binary repository and an on/off button.

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 records the number of lines in the file it describes and patches are only applicable when their indexing is within the number of lines

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

All three examples of patch theories make crucial use of HITs and univalence. As we have seen, that means they will not compute in type theories which treat either axiomatically. Specifically, the result of applying a patch to a repository will require some additional reasoning to obtain.

Additionally, the optimizer for a patch theory with laws maps into a contractible type in a potentially 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 [1].

Is it possible to formulate HPT in a setting with both HITs and univalence in such a way that application and optimization of patches reduces completely? In the following chapter we explore that question in one candidate setting: Cubical Agda.

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

Chapter 3

Formalization

This section describes the development of a formalization of Homotopical Patch Theory [1] in Cubical Agda. The development follows Angiuli et al. in defining three patch theories of increasing complexity.

"An elementary patch theory" uses the fundamental group of S^1 to implement a patch theory of integers and the successor function, "A patch theory with laws" extends to fixed-size vectors of strings and includes a patch law which is used to implement a patch optimizer, and "a patch theory with richer contexts" allows for more complex patch contexts with a repository indexed by its history.

The aim of this chapter is to explain implementation choices alongside the key code and definitions. The full implementation is available on github [1].

Lastly, we look at some concrete computations using the formalization and 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 Cubical Agda to fully explore.

[1] `https://github.com/Aqissiaq/hpt-experiments`

3.1 An Elementary Patch Theory

This section discusses the implementation of *an elementary patch theory* as described by section 4 of hpt [1].

3.1.1 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
```

3.1.2 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 _ _)
```

3.2 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 a patch *law* represented by a square (a path between path).

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.

3.2.1 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 a patch *law* 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 cylinder between each loop where `s1 == s2` and the constant path.

```
data R : Type 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 would lead to problems later. See subsection 3.2.2.

In order to interpret this model in the universe of types (called `Type` in Cubical Agda) 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 : Type
repoType = Vec String size
```

To create a path `swap s1 s2 i : repoType ≡ repoType` we will first construct an isomorphism 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 an isomorphism (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` equivalence and each `noop` square to `swapatIsId` composed with `uaIdEquiv` which is the path identifying `ua (idEquiv _)` and `refl`.

Then we can interpret and apply patches like before.

```
I : R → Type
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)
```

3.2.2 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 and (potentially) 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.

```
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 parameter, so that `(x , refl)` may be a point along the path.

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. Firstly `noop` requires a swap-patch with identical strings. Luckily we can use the proof that they are equal to get a patch of the correct type. Secondly the `noop` square goes the wrong way – from the patch to `refl` – 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
```

Contractibility of the result type is immediate from the characterization of paths in Σ-types and the inverse of the provided path.

```
result-contractible : {X : Type} → (x : X) → isContr (Σ[ y ∈ X ] y ≡ x)
result-contractible x = (x , refl) , (λ (_ , p) → ΣPathP (sym p , λ i j → p (~ i ∨ j)))
```

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)
```

We present the steps of the proof here. By the characterizations of paths over constant families and paths in Σ-types the Pathover is equivalent to $\Sigma_{\texttt{q : Patch}}$ `(transport` $^{x \mapsto (x \equiv \texttt{doc})}$ `p) ≡ refl`.

```
(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[2] to obtain the Σ-type of patches `q` and proofs that $q^{-1} \cdot p \equiv$ `refl`. The proof of `path-transport-lemma` is by path induction. It was written for this purpose and has been contributed to the Cubical library.

```
  (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)
```

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, `invLUnique` identifies $p^{-1} \cdot q \equiv$ `refl` with $q \equiv p$. The proof is by path induction and application of groupoid laws.

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)
```

[2]For the category theorist: this is the functorial action of the contravariant hom-functor [32]

3.3 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*.

3.3.1 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 in the same context. 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.

```
data History : ℕ → ℕ → Type 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
```

It would also be possible to include higher constructers to specify patch laws, but we refrain in order to simplify other developments.

```
data R : Type 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
```

3.3.2 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 simple to turn a history into a path in `R` by using the constructors of `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))
```

3.4 Computational Results

Having implemented several patch theories in a cubical setting, we may inspect their computational properties to see what application of patches does. In this section we do exactly that, considering some concrete examples of repositories, patches and merges for the three theories.

3.4.1 Elementary Patch Computations

First, consider the elementary patch theory implemented in section 3.1. 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.

```
_ : 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 takes on the order of minutes.

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

Finally, we look at `merge`. The function `merger` neatly computes the two patches p' and q' resulting from merging patches p and q from the original repository n and returns a pair of integers obtained by applying them.

```
merger : ℤ → Patch → Patch → ℤ × ℤ
merger n p q = let x = apply p n
                   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 3.1.

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

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

3.4.2 Patch Computations with Laws

Next we examine the patch theory with laws and its patch optimizer from section 3.2. 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 an inductive 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 one after the other also produces the expected result.

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

The Patch Optimizer

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.

Consider the following applications of optimized patches:

```
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 _
```

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.

3.4.3 Patch Computations with Richer Contexts

Finally, we consider the patch theory with richer contexts from section 3.3. For this theory we have implemented two interpretations, we will look at them in turn before considering 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` is 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 an 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 3.3 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 refl [])) [])
  ≡⟨ 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 :: [])) []) ■
```

Chapter 4

Conclusion

In this thesis we have constructed and outlined an implementation of homotopical patch theory in Cubical Agda[1]. 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 inductive 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).

This chapter discusses the implementation and results of chapter 3 and outlines directions for future work on homotopical patch theory.

[1] Available here: `https://github.com/Aqissiaq/hpt-experiments`

4.1 Discussion

Cubical Agda provides a good setting for formalizations relying on univalence, function extensionality and higher inductive types. Implementation of the ideas of HPT was direct, avoiding complicated elimination rules associated with HITs and univalence [2].

However, a few complications were encountered. In particular the independence patch law in "a patch theory with laws" is a glaring omission. While it is easy to state, it does not play well with our technique – adapted from the Cubical library's set truncation – for mapping out of higher path constructors. This is not a fundamental limitation, and may have solutions, but was a hurdle.

The other omission is the patch laws of "a patch theory with richer contexts". In this case the issues arose already trying to define `Interpreter`. The cases for patch laws should encode the fact that `replay` respects these patch laws, but since repositories are already indexed by a HIT of histories, the resulting type becomes very complicated and I was unable to state it correctly.

We might also expect that *any* square will do, since all four corners are singletons and all singletons are equivalent. However, this does not appear to pass Cubical Agda's typechecking because the sides must *definitionally* agree with the mapping of points and patches.

The computational results are promising, but also reveal the current limits of HoTT. There are two distinct issues: firstly the issue of `transp` and `hcomp` over inductive families which simply does not reduce, and secondly an issue of performance.

The performance limitations are also visible in two distinct places. First, recall the elementary computations in subsection 3.4.1. A detailed performance analysis is outside the scope of this thesis, but it is noteworthy that computing 1000 successor functions takes significant time.

Second, the results of `optimize` appear to be very slow. In fact, the author has been unable to compute the result of applying any optimized patch due to memory constraints. As

[2]Compare the length of just the contractibility proof in `https://github.com/dlicata335/hott-agda/blob/master/programming/PatchWithHistories3.agda` to our one-liner.

noted in section 3.4 the results themselves are not in question, since `optimize` also produces a proof that the resulting patch is equal to the original, but the resource use is interesting nevertheless.

Possible causes include the trick used to define `opt` of the `noop` constructor and the transport along the long sequence of paths constituting `e`.

4.2 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 Appendix A.

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.

Additionally, Angiuli et al. implement an alternative version of the patch theory with laws in which `R` is set-truncated by adding another constructor. This is used as an alternative to the contractibility of the target type when defining `opt` and may offer a solution to the termination problems of the independence patch law.

The other is to work towards more computational results. Specifically we are limited by inductive families, and further work in Cubical Agda's normalization would lead to more results "for free" [3].

It would also be interesting to look at the computation of `opt`, as it requires some notions of sub-singletons [1] and is very time consuming at the time of writing.

Additionally, it might be possible to obtain more results by writing a more direct implementation. In May of 2022 Mörtberg and Ljungström succeeded in computing the Brunerie

[3]This is an active area of work for Cubical Agda, see for example `https://github.com/agda/agda/issues/3733#issuecomment-903581647`, `https://github.com/agda/cubical/pull/309` and section 3.2.4 of [33]

number in Cubical Agda in a matter of seconds [cite axel's blog post when it shows up?] - where previous attempts had resulted in programs that ran for over 90 hours without a result [21]. The new proof achieves this by providing a more direct construction [citation needed] and avoiding computationally expensive equivalences (in particular $S^3 \simeq S^1 * S^1$). Their development suggests that more results may be achieved by identifying the computationally expensive aspects of the current construction, and avoiding these with more direct alternatives.

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. An attempt at an alternative approach is included in Appendix A.

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] Andrej Bauer and Peter LeFanu Lumsdaine. *A Coq proof that Univalence Axioms implies Functional Extensionality*. 2011. URL: http://nlab-pages.s3.us-east-2.amazonaws.com/nlab/files/BauerLumsdaineUnivalence.pdf.

[4] 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.

[5] 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.

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

[7] *Darcs Theory: Conflictors*. 2022. URL: http://darcs.net/Theory/Conflictors.

[8] Nicola Gambino, Chris Kapulkin, and Peter LeFanu Lumsdaine. "The univalence axiom and functional extensionality". In: (2016). URL: https://www.uwo.ca/math/faculty/kapulkin/notes/ua_implies_fe.pdf.

[9] 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.

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

[11] Chris Kapulkin and Peter LeFanu Lumsdaine. "The Simplicial Model of Univalent Foundations (after Voevodsky)". In: (2012). DOI: 10.48550/ARXIV.1211.2851. URL: https://arxiv.org/abs/1211.2851.

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

[13] Amélia Liao. *1Lab.Univalence*. 2021. URL: https://1lab.dev/1Lab.Univalence.html.

[14] Daniel R Licata and Guillaume Brunerie. "A cubical approach to synthetic homotopy theory". In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE. 2015, pp. 92–103.

[15] Daniel R Licata and Robert Harper. "2-dimensional directed type theory". In: *Electronic Notes in Theoretical Computer Science* 276 (2011), pp. 263–289.

[16] Daniel R. Licata and Michael Shulman. "Calculating the Fundamental Group of the Circle in Homotopy Type Theory". In: *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. USA: IEEE Computer Society, 2013, pp. 223–232. ISBN: 9780769550206. DOI: 10.1109/LICS.2013.28. URL: https://doi.org/10.1109/LICS.2013.28.

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

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

[19] Simon Marlow et al. "Haskell 2010 language report". In: (2010).

[20] 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.

[21] Anders Mörtberg. *Yet Another Cartesian Cubical Type Theory yacctt*. 2018. URL: https://www.youtube.com/watch?v=oc_ChPjL-Rk&t=350s&ab_channel=HausdorffCenterforMathematics.

[22] 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.

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

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

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

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

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

[28] T. Streicher. "A model of type theory in simplicial sets: A brief introduction to Voevodsky's homotopy type theory". In: *Journal of Applied Logic* 12.1 (2014). Logic Categories Semantics, pp. 45–49. ISSN: 1570-8683. DOI: https://doi.org/10.1016/j.jal.2013.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1570868313000347.

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

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

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

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

[33] 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.

[34] Vladimir Voevodsky. "The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010)". In: (2014). DOI: 10.48550/ARXIV.1402.5556. URL: https://arxiv.org/abs/1402.5556.

Appendix A

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 [12].

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. Constructing such theories in terms of higher inductive types leads 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.

A.1 Repository HIT

Aiming to construct a HIT in which point constructors are repository contexts and path constructor represent patches, we arrive at an inclusion of some base type into the HIT and one or more named paths with endpoints specified in the base type.

The result is much like the coequalizers discussed by Kraus and von Raumer.

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 $p : a \sim b$ [12].

In formal terms this is the introduction rule:

$$\frac{\begin{array}{l}\Gamma \vdash A\ Type \\ \Gamma, a\ b : A \vdash a \sim b\ Type\end{array}}{\Gamma \vdash A/\!\!/_\sim\ Type} \tag{A.1}$$

And the two formation rules:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : A/\!\!/_\sim} \qquad \frac{\Gamma \vdash p : a \sim b}{\Gamma \vdash glue\ p : [a] \equiv [b]} \tag{A.2}$$

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 ]
```

A.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)]

$$\frac{\begin{array}{l}\Gamma \vdash a_0 : A \\ \Gamma, b : A \vdash P : [a_0] \equiv [b] \to Type \\ \Gamma \vdash r : P\ refl_{[a_0]} \\ \Gamma, b\ c : A, q : [a_0] \equiv [b], s : b \sim c \vdash P\ q \simeq P\ (q \cdot (glue\ s))\end{array}}{\Gamma \vdash ind\ r\ e : \Pi_{(b:A)}\Pi_{q:[a_0]\equiv[b]}P\ q} \tag{A.3}$$

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
```

A.3 Result/Discussion

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.

One possible intuition, in the case of merge, is that it represents an equivalence between "ways to merge an arbitrary q with p" and "ways to merge an arbitrary q with $p \cdot sett x$".

However, this has not proven fruitful and despite efforts to construct more fitting equivalences the result has been the same "reverse everything"-merge.

In their discussion of homotopical patch theory [1] Angiuli et al. mention that the requirement that all patches must have strict inverses presents difficulties. Their solution is the history-inedxed repositories of "a patch theory with richer contexts," which introduces a lot of complexity. Our problem may be related, since the requirement that the final induction term be an equivalence arises from the need for paths to have inverses.

Possible solutions include a directional indexing like HPT's histories and further investigation of directed type theory [15] which treats paths non-symmetrically.

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 immediately clear that it extends to more complicated theories with multiple kinds of patches and patch laws.