Bachelor's Thesis Computing Science

Radboud University Nijmegen

Implementing Patch Theories in Homotopy Type Theory

Author:
Dick Blankvoort
s1056960

First supervisor/assessor:
Niels van der Weide

Second assessor:
Freek Wiedijk

July 19, 2023

Abstract

Homotopy type theory is a field lying at the intersection of functional programming and various mathematical fields including algebraic topology, homological algebra and higher category theory. In this paper, we implement a primitive version of Darcs patch theory in homotopy type theory, as laid out theoretically in the paper *Homotopical Patch Theory* by Angiuli, Morehouse, Licata and Harper. For this we make use of the proof assistant Coq, and more specifically its homotopy type theory library Coq-HoTT. To achieve our goals, we first reimplement the integers and then use this new definition to successively implement all the theoretical concepts put forth in the paper.

Contents

Chapter 1

Introduction

1.1 Homotopy Type Theory

1.1.1 What is Homotopy Type Theory?

Homotopy type theory (HoTT for short) is a field lying at the intersection of functional programming and various mathematical fields including algebraic topology, homological algebra, and higher category theory. It aims to provide a novel set of foundational principles which can be used in both mathematics and computing. It is a relatively recent field, dating back to a paper in 1994 by Hofmann and Streicher [1]. Afterwards, the topic was greatly expanded upon and refined in subsequent papers by Voevodsky [2] and Awodey and Warren [3]. Since its inception, it has grown to encompass a wide variety of specializations, has been implemented in both Coq and Agda, and has even developed a subdiscipline of its own [4], [5].

1.1.2 Differences With Other Foundations

Homotopy type theory stems from the field of type theory, which is an alternative to set theory. Instead of considering every object either as a set or an assertion about a set, type theory distinguishes between types (objects containing elements) and elements. Along with sidestepping Russell's paradox [6], this system introduces much greater depth and nuance in the conceptual structure of equalities (otherwise denoted 'paths').

The way homotopy type theory expands on type theory is by introducing the univalence axiom, which asserts that an equality between types is equivalent to an equivalence. As such all isomorphic types are automatically considered equal to each other, meaning that, for instance, there exists only one singleton type `Unit` [7] as opposed to many singleton types or sets. Homotopy type theory also provides a unique geometric view of types, providing its link to homotopy theory and by extension to algebraic topology. Under homotopy type theory types are commonly interpreted as spaces, with the elements of each type serving as points within the space and equalities between elements as paths or identifications within the space. Under this interpretation, it is possible to model geometrical objects such as circles and toruses as types.

1.1.3 Key Example: The Circle

In homotopy type theory we have a core construct of a higher inductive type. This construct is quite similar to an ordinary inductively defined type, and similarly is built up from a series of constructors. It fundamentally consists of one or more point constructors and one or more path constructors. Point constructors are used to generate the points, which is to say the elements, of the higher inductive type. The path constructors, then, are used to specify paths between these elements and indicate when points are deigned to be equal. If we take the trivial case when the point constructor provides a single point `x` and the path constructor is the identity path stating that `x = x`, the higher inductive type remarkably takes on the form of the homotopical notion of a circle [8]. See below.

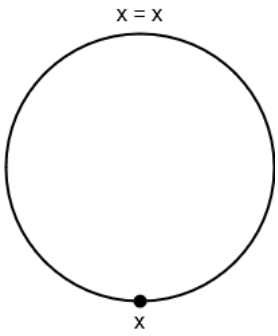

Figure 1.1: the circle as a higher inductive type.

As it turns out, the fundamental group of this type (its group of homotopy equivalence classes) is exactly the type $\mathbb{Z}$ of the integers. This has led to the development of many techniques applicable to the integers but designed for and named after the circle [9]. Similar techniques have also been used to prove additional relations, such as that the product of two circles in the theory is geometrically equivalent to the torus [4], [10].

One ordinarily interacts with inductive types through its induction principle, which reduces it to its constructors. Similarly, one interacts with a higher inductive type through its recursion principle and its induction principle. The recursion principle is responsible for assigning mappings from a higher inductive type to some other type, which we will call `A`. It does this by:

1. Assigning mappings from every point constructor `x` to an element `a` of type `A`.
2. Assigning mappings from every path constructor to paths `a`, `b`, ..., `z`.

The induction principle, meanwhile, takes a mapping `P` from the point constructors to the universe type and:

1. Assigns mappings from a point constructor `x` to an element `a` of type `P x`.
2. Maps every path constructor to some operation on the aforementioned members.

It must be noted that the recursion principle is in fact a weaker form of the induction principle, being the case where `P` is a constant type family.

1.2 Darcs Patch Theory

Darcs (an acronym for Darcs Advanced Version Control System) is a version control system which abstracts many of the traditional operations one would expect in version control systems back to the two elementary operations of pushing and pulling [11]. Separate branches do not explicitly exist, allowing for the merge and the branch operation to be expressed simply as a node being brought into the same state as another node.

Another hallmark feature of Darcs patch theory is its view of repositories as sets of patches [12]. When combined with the aforementioned abstracting away of commands this allows the core functionality of the theory to be expressed in particularly few lines of code, which is useful both for creating a solid foundation on which more complex operations can be built and for implementing versions of Darcs in other programming languages. In this paper, we make use of the ease of portability associated with this system to partially implement it in homotopy type theory.

1.3 Problem Formulation

In this paper, we seek to implement the first of the homotopical patch theories set out in the paper *Homotopical Patch Theory* by Angiuli, Morehouse, Licata and Harper [13]. We do this within the framework of Coq-HoTT, which is an implementation of homotopy type theory in the proof assistant Coq [14]–[17].

The paper *Homotopical patch theory* formulates a theoretical way to implement Darcs patch theory in homotopy type theory. It does this through three high-level formulations, each of which adds more features to the theory. The first patch theory the authors propose treats nodes as integers and only allows for a patch to add or subtract from a node by a certain integer amount. Their second patch theory expands on this by instead treating nodes as collections of strings, and patches as a series of replace-at operations which replace a certain line with another. Their third patch theory builds on this again by considering proper patch histories and adding additional constraints.

As this thesis only implements the first of these patch theories, we only discuss this one in detail. The first patch theory is implemented as a higher inductive type, with nodes being treated as elements and patches as paths between nodes. Patches can be translated to other types by means of interpreters, and the paper proposes two of these. The first interpreter sees patches as bijections between integers, with the left- and right-hand sides of the bijection being interpreted as the contents of the node before and after the patch respectively. The second interpreter sees patches as bijections between booleans. Under this interpreter, nodes are seen to contain boolean values and a patch either leaves a node unchanged (if it is interpreted as a bijection mapping `true` to `true` and `false` to `false`), or inverts it (if it is interpreted as a bijection mapping `true` to `false` and `false` to `true`).

Besides proposing the structure of the patch theory and the two interpreters, the paper also proposes a merge operation for patches. This operation is envisioned as taking a pair of patches and returning the patches necessary to bring them into the same state. The opera-

tion satisfies two laws known as `reconcile` and `symmetric`. The `reconcile` law requires that the returned patches indeed bring the patches given as arguments into the same state. The `symmetric` law, meanwhile, requires that the merge operation is commutative. The paper then proceeds to lay out a rough approach for how to implement the merge operation, which we will mostly be following in our paper.

1.3.1 Related Work

Swierstra and Löh [18] give a more general overview about how one would go about implementing a version control system in a system of formal logic. Mimram and Di Giusto [19] provide a different high-level description of a patch theory within the related mathematical field of category theory. Reynolds [20] provides an alternative proposal for implementing a patch theory using Hoare logic. Lastly, Jason [21] provides an implementation of a simple version control system in the programming language Haskell.

1.4 Overview of the Paper

In Chapter 2, we lay out some of the preliminary knowledge needed to understand both homotopy type theory in general as well as its implementation in Coq known as Coq-HoTT.

In Chapter 3, we reimplement the integers as to make them simpler to work with in our implementation of the patch theory. In Chapter 4, we use the encode-decode principle to prove that these integers form a set, allowing us to use many properties common to all sets.

In Chapter 5, we implement most of the patch theory as laid out in *Homotopical patch theory*. We also create the interpreters proposed for this patch theory in order to enable us to demonstrate consistency of the model with the theory.

In Chapter 6, we complete the implementation of our patch theory by defining the merge principle. We then demonstrate consistency of this principle by providing proof of two key properties. Lastly, in Chapter 7 we conclude our paper and discuss possible future work.

For the code of our implementation, see here.

Chapter 2

Preliminaries

2.1 Homotopy Type Theory

2.1.1 Paths and Equivalences

First of all, it is important to distinguish *paths* from *equivalences*. A path is a direct, a priori, equality. It indicates that two elements are not just indistinguishable from each other but the exact same thing. In this paper, we indicate paths using the notation `x = y`. An equivalence, on the other hand, indicates that two types are *functionally equal*. That is, they contain both the same elements and have the same higher dimensional path structure (something we will come back to in Section 2.1.5 and Section 2.1.6). In this paper, we indicate an equivalence using the notation `x ≃ y`.

2.1.2 Introduction, Elimination and Computation rules

Paths in homotopy type theory are considered types, and along with types come the concepts of an introduction, elimination, and computation rule. The introduction rule for paths states that a path can be constructed between types only if all its elements match [6]. Conversely, the elimination rule states that if there exists a path between two elements, then all its components must match.

The computation rules for paths allow one to identify or 'extract' either endpoint of a path. These rules make use of an application operation `ap`, which applies a function to both endpoints, and a projection operation `pr`, which extracts a certain element from a type. Together the introduction, elimination, and computation rules form the basic bedrock on which all composite operations can be built.

2.1.3 Basic Notation

One of the most elementary types in homotopy type theory is the type `Unit` containing only a single element [7]. When working with Coq-HoTT, we commonly denote this element with `tt`. The unit type, like paths, comes accompanied with an introduction, elimination and computation rule. The introduction rule for the unit type states that it comes equipped with a single element `tt`. The elimination rule states that for any mapping `P : Unit → Type`, we have

an induction function `unit_ind : P tt → (forall x : Unit, P(x))` which allows for the mapping to only be defined on `tt`. The computation rule proceeds to state that `forall (p : P tt)`, `unit_ind p tt ≃ p`. In natural language: the induction principle does not change where `tt` maps to.

Another elementary type in homotopy type theory is the type `Empty` containing no elements. The introduction rule for the type `Empty` states that it does not come equipped with any elements. The elimination rule states that for any mapping `P' : Empty → Type` we have an induction function which makes it vacuously hold. Applying this induction function is commonly referred to as empty-induction. The computation rule for `Empty` does not exist, given that the type does not contain any elements.

When we formulate statements as types in homotopy type theory, we have the property that a statement holds if and only if we can construct one of its elements. Given that it is impossible to identify an element in the empty type and always possible to identify one in the unit type, we arrive at the logical conclusion that when we interpret the two types as statements, the empty type translates to "false" while the unit type translates to "true".

Functions in homotopy type theory are defined in a way much analogous to functional programming. That is, there exists an identity function `idmap` which takes an argument and leaves it unchanged and fundamentally only single-argument functions exist. Functions with more than one arguments are subsequently defined in a composite way using a currying operation `o`. There also exists a notion of an anonymous / lambda function, for which we use the notation `fun <args> ⇒ <return>`.

The two most important computation rules for functions are those of beta- and eta-conversion. The beta-conversion rule states that the following equivalence holds:

`(fun x ⇒ f(x))(a) ≃ f(a)`

while the eta-conversion rule states that the following equivalence holds:

`f ≃ (fun x ⇒ f(x))`

2.1.4 Path Induction

The path induction principle of homotopy type theory states each path can be constructed from the identity path 1. More concretely, is says that given:

1. A type family `C`, taking two elements `x` and `y` of some type `A` and outputting some type family `B(p)` indexed by a path `p : x = y`.
2. A function `c(x) := C x x 1`.

We have a function `f(x, y, p) := C(x, y, p)` such that `f(x, x, 1) = c(x)` [6]. Because of this inductive definition of paths, it is always possible to use induction to reduce paths back to the identity path as long as one end is open. This operation is called path induction, and it serves as one of homotopy type theory's core tenets. Some other important operations for paths are as follows:

- ^: forall (A : Type) (x y : A), x = y → y = x, which gives the inverse of a given path.
- ap: forall (A B : Type) (f : A → B) (x y : A), x = y → f x = f y, which can be used to make functions applicable to paths.
- transport: forall (A : Type) (P : A → Type) (x y : A), x = y → (P x → P y), which transforms a path a = b into a function mapping a to b.

Together these operations give way to a wide variety of properties, the following of which have particular relevance to our paper:

- For all paths p : x = y and q : y = z, we have that (p @ q)^ = q^ @ p^ [inv_pp].
- Given a type A, a function P : A → Type, two points x y : A, a path p : x = y, and the image u of x in P, we have that transport P p u = transport_idmap (ap P p) u [transport_idmap_ap].
- Given a type A, a function P : A → Type, three points x y z : A, paths p : x = y and q : y = z, and the image u of x in P, we have that transport P (p @ q) u = transport P q (transport P p u) [transport_pp].
- Given a type A, two functions B C : A → Type, two points x1 x2 : A, a path p : x1 = x2, a function f : B x1 → C x1, and the image y of x2 in B, we have that transport (fun x : A ⇒ B x → C x) p f y = transport C p (f transport B p^ y) [transport_arrow].
- Given a type A, three points x y1 y2 : A, and paths p : y1 = y2 and q : x = y1, we have that transport (fun y : A ⇒ x = y) p q = q @ p [transport_paths_r].
- Given a function f : A → B, two points x y : A, and a path p : x = y, we have that (ap f p)^ = ap f p^ [inverse_ap].
- Given a function f : A → B, two points x y : A, and a path p : x = y, we have that ap f p^ = (ap f p)^ [ap_V].
- Given a function f : A → B, three points x y z : A, and paths p : x = y and q : y = z, we have that ap f (p @ q) = ap f p @ ap f q [ap_pp].
- Given a type A, two points x y : A, and a path p : x = y, we have that 1 @ p = p [concat_1p].
- Given a type A, two points x y : A, and a path p : x = y, we have that p @ 1 = p [concat_p1].
- Given a type A, two points x y : A, and a path p : x = y, we have that p @ p^ = 1 [concat_pV].
- Given a type A, two points x y : A, and a path p : x = y, we have that p^ @ p = 1 [concat_Vp].
- Given a type A, points x y z : A, and paths p : x = y and q : y = z, we have that (p @ q) @ q^ = p [concat_pp_V].

- Given a type `A`, points `x y z : A`, and paths `p : x = y` and `q : y = z`, we have that `p @ (p^ @ q) = q` [`concat_p_Vp`].
- Given a type `A`, points `x y z : A`, and paths `p : x = y` and `q : y = z`, we have that `p^ @ (p @ q) = q` [`concat_V_pp`].
- Given a type `A`, points `x y z t : A`, and paths `p : x = y`, `q : y = z` and `r : z = t`, we have that `(p @ q) @ r = p @ (q @ r)` [`concat_pp_p`].
- Given a type `A`, points `x y z t : A`, and paths `p : x = y`, `q : y = z` and `r : z = t`, we have that `p @ (q @ r) = (p @ q) @ r` [`concat_p_pp`].

2.1.5 hLevels

We already briefly mentioned that types have a higher dimensional path structure, however this is not always the case. For contractible types [22], we have the special property that there exists a path between every element. This implies that the type only contains one element, meaning every contractible type is exactly the type `Unit`. Propositions under homotopy type theory meanwhile have the special property that the type of paths between them is contractible. In other words: every means of proving that two propositions are equal is equivalent to each other. For sets we do not have this property, however we do have the property that the type of paths between them is a type of propositions. We thus arrive at a three-tiered system, with contractible objects at the bottom and set-like objects at the top, and with us moving down one level when we consider a path between objects.

Homotopy type theory further generalizes this system to the concept of truncations. Under this system, contractible objects are seen as (-2)-truncated objects, mere propositions as (-1)-truncated objects and sets as 0-truncated objects. A 1-truncated object would logically be an object for which a proof of equality forms a set, a 2-truncated object an object for which a proof of equality forms a 1-truncated object, etcetera. In Chapter 4, we use this novel construct to show that our new definition for the integers forms a set.

A good advantage of this system is that it gives us a certain 'ceiling' to the path space of types. Since there is exactly one element in a contractible object, we have a lemma `path_ishprop` that all paths within a mere proposition exist. After all, these paths all correspond to that single element. This further propagates to higher truncation levels, protecting us from the otherwise common issue of having to compute the behavior of the path space up the truncation levels *ad infinitum*. In general, hLevels are a useful tool in homotopy type theory for keeping the necessary workload manageable.

2.1.6 Equivalences

Requirements for an equivalence

In homotopy type theory, two types `A` and `B` are called *equivalent* to each other when there exists an equivalence between them. This is the case when:

1. There exists a bijection (as defined by set theory) between `A` and `B`.
2. The path spaces of `A` and `B` are the same.

In Coq-HoTT, we denote such an equivalence using the type `Equiv A B`. To construct an element of such a type we provide an element of the type `Build_Equiv`, which requires the following:

1. A function `f` from `A → B`
2. A function `g` from `B → A`.
3. A proof `eisretr` that `f o g` is equivalent to the identity map.
4. A proof `eissect` that `g o f` is equivalent to the identity map.
5. A proof `eisadj` that for every element `x : A`, `eisretr (f x) = ap f (eissect x)`.

Here, the first four properties prove that there exists a bijection between `A` and `B` constructed through `f` and `g`, as defined in the context of set theory. The fifth property then proves that the path spaces of `A` and `B` are equivalent. It must be noted that after we have provided the first function `f`, we end up with a type `IsEquiv f` which we can construct using the remaining conditions. This secondary type is occasionally used instead of `Equiv` to define properties for equivalences.

It may be noted that hLevels can be used to circumvent proving that the path spaces of two types are equivalent. If we know that the types are both mere propositions, for example, then we automatically know that their path spaces are the same. As such, if we deem it unnecessary to prove that the path spaces are equal we can instead employ the function `equiv_isequiv` to remove the condition `eisadj` from the proof of `IsEquiv f`.

Similar to functions, equivalences also have a composition and an inverse operation. In this paper and in Coq-HoTT we denote these by `oE` and `^−1` respectively. There are also some other properties relating to equivalences which are provided by Coq-HoTT in the form of lemmas. Of these, the following have particular relevance to our paper:

- Given types `A` and `B` and a function `f : A → B` there exist paths between all proofs of `IsEquiv f`. In other words: the type of equivalences between two types `A` and `B` is a mere proposition / (-1)-truncated object [`path_equiv`].
- Given types `A` and `B`, if there exists an equivalence between `A` and `B` then there also exists one between `B` and `A` [`equiv_inverse`].
- Given types `A` and `B`, if there exists a path between `A` and `B` then there also exists an equivalence between them [`equiv_path`].
- Given types `A` and `B` and a path `p` between them, `equiv_path B A (p^) = (equiv_path A B)^−1` [`equiv_path_V`].

- Given types `A` and `B`, a function `f : A → B`, a proof `H` that `A` and `B` form an equivalence through `f`, two points `x` and `y` in `A`, and a proof that `f x = f y`, we have that `x = y` [`equiv_inj`].

- If there exists an equivalence between two types `A` and `B`, the types have the same truncation level [`istrunc_equiv_istrunc`].

2.1.7 Univalence

The univalence axiom is one of the central tenets of homotopy type theory. Put simply, it states that an equivalence between types is equivalent to a path. The axiom has greatly expanded the scope of the field and provided some of the most key separations from set theory. One consequence of the axiom, for example, is that it implies that there must exist some type which is not a 0-truncated object, namely the type `Type` containing all types. One analogue to the univalence axiom is the function extensionality axiom, which states that a homotopy between function is equivalent to a path. This axiom also sees some minor applications in our model, see Section 2.2.2 and Section 5.2.1.

In Coq-HoTT, the univalence axiom is implemented on one end by `equiv_path : A = B → A ≃ B`, which is distinct from the lemma `path_equiv` we encountered above, and on the other end by a lemma called `path_universe_uncurried : A ≃ B → A = B`. For this last lemma we have a variety of useful properties, of which the following are of relevance to our paper:

- We have that `path_universe_uncurried f^−1 = (path_universe_uncurried f)^` [`path_universe_V_uncurried`].

- We have that `path_universe_uncurried (equiv_compose g f) = path_universe_uncurried f @ path_universe_uncurried g` [`path_universe_compose_uncurried`].

- We have that `path_universe 1%equiv = 1` [`path_universe_1`].

- We have that `transport idmap (path_universe_uncurried f) z = f z` [`transport_path_universe_uncurried`].

- We have that `path_universe_uncurried (equiv_path A B p) = p` [`path_universe_uncurried_equiv_path`].

- We have that `transport idmap (path_universe_uncurried f)^ z = f^−1 z` [`transport_path_universe_V_uncurried`].

- If two equivalences `f` and `g` are definitionally equal, then there exists a path between `path_universe_uncurried f` and `path_universe_uncurried g` [`path2_universe`].

2.2 Working with Coq-HoTT

2.2.1 Structures

In Coq-HoTT, there are a variety of structures used to create types. In this section, we will go through them one by one.

Firstly, we have `Definition`s, `Proposition`s, and `Lemma`s. These keywords are all treated analogously, and can be used to define most types without a recursive aspect (including assertions). Usually, the keyword `Definition` is used to define proper types, `Proposition` is used to define assertions and `Lemma` is used to define prerequisites required to prove a particular `Definition`. There is also the `Axiom` structure, which serves a similar role to a `Lemma` however does not necessitate a proof. Usually, one seeks to transform all axioms into lemmas before concluding a proof.

Next, there is the `Inductive` notation. This can be used to define an inductive structure for a given type by providing both the base element(s) of that type and the functions acting on those elements. Each member of the inductive type is indicated on a separate line with a vertical bar (|) before it. For instance, the inductive type of the positive integers consisting of a number 1 and a successor function is defined as follows:

```
Inductive Pos : Type :=
| one : Pos
| succ : Pos → Pos.
```

Match-notation is used to provide case analysis in a more readable form than the ordinary induction tactic. It starts with the statement `match` <args> `with` and ends with the keyword `end`. In between, the various cases are worked out in a similar notation to Inductives however with the specific case indicated before the symbol ⇒ . If a case distinction is recursive (like is commonly the case when basing a definition on Inductives), the surrounding type receives a special label known as a `Fixpoint`. For instance, if we want to construct positive number addition on the `Inductive` type `Pos` using match-notation, we would arrive at the following definition:

```
Fixpoint pos_add (x y : Pos) : Pos :=
match x, y with
| one, one ⇒ succ one
| succ p, one ⇒ succ (succ p)
| one, succ q ⇒ succ (succ q)
| succ p, succ q ⇒ succ (succ (pos_add p q))
end.
```

A `Global Instance` is a special keyword which can be used to provide properties about the hLevels of other types (more on that next section). Using such a keyword registers the proof as a typeclass, allowing tactics such as `apply` to automatically find it. Furthermore, type constraints can be defined for lemmas which allow them to only be used when certain typeclasses are in the context. In terms of syntax the structure operates much analogously to a regular `Definition`, and as such the `Global Instance` which proves that, for instance, the positive numbers for a set is constructed with the following signature:

```
Global Instance hset_pos : IsHSet Pos.
```

2.2.2 Tactics

Coq-HoTT offers a wide variety of tactics to modify a proof state. In this subsection, we will go over those tactics which are of most relevance in the paper.

Firstly, there are `intro` and `intros`. These tactics take a goal which starts with either an implication or a forall, and modify the proof state to move either the condition of the implication or the variable in the forall to the assumptions. The name of this assumption is determined automatically if no argument is specified, or set to the argument if is. The tactic `intro` performs a single introduction while `intros` performs it recursively (with additional arguments specifying additional assumption names). The tactic `revert` is the opposite of these tactics, taking the name of an assumption as an argument and modifying the proof state to bring that assumption back into the goal. To illustrate this more clearly, take the following proof state:

```
A : Type
______________________________________(1/1)
forall x y : A, x = y → y = x
```

If we apply the tactic `intros`, the two variables `x` and `y` are moved into the assumptions as well as the condition `x = y`. This then gives us the following state:

```
A : Type
x, y : A
X : x = y
______________________________________(1/1)
y = x
```

If it now turns out that we need to keep the implication to be able to properly close the proof, we can either go back and modify our tactic to `intros x y`, or bring the variable back using `revert X`. Either way we obtain the following:

```
A : Type
x, y : A
______________________________________(1/1)
x = y → y = x
```

Next, there is `rewrite`. This tactic takes the name of an assumption as an argument and tries to use that assumption to modify the goal. For example, if we take the proof state we obtained after applying `intros` and apply the tactic `rewrite X`, we obtain the following state:

```
A : Type
x, y : A
X : x = y
______________________________________(1/1)
y = y
```

Next up is `reflexivity`. In the event that the goal of a proof state simplifies to a reflexive equality, `reflexivity` closes that goal as completed. For instance we could apply `reflexivity` in the above proof state to complete our proof.

The tactic `pose` takes the name of an existing lemma and adds its contents to the assumptions. Alternatively, it also allows one to provide such contents directly as the argument.

The tactic `enough` functions similarly to `pose`, however instead adds an element of the argument it is given to the assumptions. It then proceeds to create a second branch where the existence of such an element can be proven as a goal.

The tactic `apply` takes the name of an existing lemma and tries to apply it to the current goal. Using the above example again, assume that the type `A` was instead the type `Unit`. This would give us the following proof state:

```
x, y : Unit
X : x = y
______________________________________(1/1)
y = x
```

The lemma `path_unit` states that any two elements in `Unit` are equal, and so in this case we can close off the proof without even needing our condition `X` by making use of the tactic `apply path_unit`.

In the event that the goal is exactly equal to some variant of a lemma we know, the tactic `exact` immediately closes the proof state. In practice this lemma is able to do the same things as `apply`, and which one is more convenient depends on the specific circumstance. A variant of `exact` is `refine`, which allows for holes to be left which are subsequently transformed into separate branches. This process also involves beta-reduction to simplify the branches and goals. However, since this last step can occasionally decrease rather than increase comprehensibility a variant tactic `simple refine` is offered which does not perform it.

The tactics `cbn` and `simpl` simplify a goal by applying a variety of reduction rules. The key distinction between the two is that while `cbn` unfolds and simplifies objects defined through match-notation, `simpl` does not.

The tactic `induction` takes an element whose type is defined through induction or case distinction and either starts a proof by induction or splits the cases of the argument up. It does this by creating a separate branch for every base and inductive case. In the case of objects defined through induction, it also adds an inductive hypothesis to the assumptions where applicable. The tactic `destruct` operates much like `induction`, however is solely concerned with splitting up goals and assumptions defined through case distinction.

The tactic `split` splits up a proof state where the goal is somehow a conjunction of two subgoals. This could be through an actual conjunction, however it is also capable of splitting up goals which are 2-tuples or product types (although these are essentially the same in Coq-HoTT).

The tactic `funext` operates much like `intro`, however introduces a variable on both sides of an equality instead. This tactic is not possible on every equality however, and in particular might require first applying the lemma `path_equiv`.

2.2.3 Higher Inductive Types

In Coq-HoTT, higher inductive types are defined in a separate module using the syntax `Module Export` <name>. ... `End` <name>.. The point constructor is defined in a special `Private Inductive` structure as to obfuscate its number of elements outside of the module, necessitating the use of the recursion and induction principles to interact with it. The path constructors are defined separately using `Axiom`s. The recursion and induction principles, meanwhile, are defined in two parts. They use a `Definition` for the point constructor mapping and an `Axiom` for the path constructor mapping.

2.2.4 Environment

For the implementation created in our paper we used a variety of constructs provided by the Coq-HoTT library. More specifically, we set up out environment as follows:

```
From HoTT Require Import Basics Types.

Context `{Univalence}.
```

In terms of lemmas we only needed to import the very basics. As such the lemmas provided by `Basics` and `Types` were deemed enough. Aside from that we also added the univalence axiom to our context as a typeclass (achieved using the `{} notation), allowing us to use `path_universe_uncurried` and its related lemmas. Together this gave us all the tools necessary to tackle the problem at hand.

Chapter 3

Integers

In the previous chapters, we discussed the basics of homotopy type theory and patch theory, and made a start with the implementation by defining the environment used for working with homotopical patch theory. In this chapter, we continue to develop the prerequisites by defining the positive numbers and integers as types.

Coq-HoTT's integers are defined in terms of bitstrings. While this is usually a very elegant way to define such numbers (requiring relatively few recursive calls and allowing for more efficient multiplication than a definition using a successor function), this makes them rather cumbersome to work with when performing induction. As such, before we are able to provide a clean implementation of the patch theory we first need to reimplement the positive numbers using a successor function. This then allows us to redefine the integers using this new type of positive numbers. In total we need the following:

1. An inductive definition of the positive numbers as a type `Pos` (Section 3.1).
2. An addition function `pos_add` for positive numbers (Section 3.1).
3. A proof `pos_add_comm` that positive number addition commutes (Section 3.1).
4. A proof `hset_pos` that the positive numbers form a set (covered in chapter 4, Section 4.1.1).
5. An definition of the integers as a type `Int` (Section 3.2.1).
6. A successor function `int_succ` and predecessor function `int_pred` for the integers (Section 3.2.1).
7. A proof `equiv_int_succ` that the successor and predecessor functions for the integers form an equivalence (Section 3.2.2).
8. An addition function `int_add` for the integers (Section 3.2.3).
9. A proof `int_add_comm` that integer addition commutes (Section 3.2.4).
10. A proof `hset_int` that the integers form a set (covered in chapter 4, Section 4.2.1).

3.1 Positive Numbers

The type of positive numbers can be defined as the inductive type containing the number 1 and a successor function. In Coq-HoTT syntax, this gives the following type:

Type 3.1.1

```
Inductive Pos : Type :=
| one : Pos
| succ : Pos → Pos.
```

Positive number addition can then be defined relatively straightforwardly through case distinction as follows:

Type 3.1.2

```
Fixpoint pos_add (x y : Pos) : Pos :=
match x, y with
| one, one ⇒ succ one
| succ p, one ⇒ succ (succ p)
| one, succ q ⇒ succ (succ q)
| succ p, succ q ⇒ succ (succ (pos_add p q))
end.
```

Indeed, it holds that:

- $1 + 1 = 2$
- $(n + 1) + 1 = n + 2$
- $1 + (m + 1) = m + 2$
- $(n + 1) + (m + 1) = (n + m) + 2$

We choose a four-case definition here as it results in the two arguments being treated equivalently. If instead we had only defined two cases, one for each of the possible values of `x`, the operations `pos_add x y` and `pos_add y x` would be evaluated quite differently and we would need to apply induction much more often in order to show that they are equivalent.

The proposition `pos_add_comm` that positive number addition operation commutes has the following signature:

Type 3.1.3

```
Proposition pos_add_comm : forall (x y : Pos), pos_add x y = pos_add y x.
```

We prove this using induction. Applying the tactic `induction x, y` gives us the following four branches:

```
______________________________________(1/4)
pos_add one one = pos_add one one
______________________________________(2/4)
pos_add one (succ y) = pos_add (succ y) one
______________________________________(3/4)
pos_add (succ x) one = pos_add one (succ x)
______________________________________(4/4)
pos_add (succ x) (succ y) = pos_add (succ y) (succ x)
```

The reason we use `induction x, y` here rather than applying `induction x` and `induction y` separately is that it results in a more powerful inductive hypothesis. This will be necessary in almost all our inductive proofs to prove the final branch.

Branches 1-3 immediately hold through reflexivity, since the definitions on the right- and left-hand sides unfold to the same terms. For the fourth branch, unfolding the definitions using the tactic `cbn` gives the following:

```
x : Pos
IHx : forall y : Pos, pos_add x y = pos_add y x
y : Pos
______________________________________(1/1)
succ (succ (pos_add x y)) = succ (succ (pos_add y x))
```

Here the goal is simply the inductive hypothesis instantiated to `y` and with the successor operation applied twice to both the left- and right-hand sides. As such, the branch can be resolved by using the tactic `exact (ap (succ o succ) (IHx y))`. With this, we have completed our definition of the necessary basic types associated with the positive numbers.

3.2 Integers

3.2.1 Basics

For our purposes, integers can best be defined as the disjoint union of the number zero with the types of positive numbers with positive and negative sign. This definition allows for easy application of induction while keeping the formulation of the addition function relatively straightforward. In Coq-HoTT syntax, this results in the following:

Type 3.2.1

```
Inductive Int : Type :=
| zero : Int
| pos : Pos → Int
| neg : Pos → Int.
```

As mentioned previously (Item 6), we require a successor function as well as a predecessor function for this type. Our successor function can be defined using case distinction as follows:

Type 3.2.2

```
Definition int_succ (i : Int) : Int :=
match i with
| zero ⇒ pos one
| pos p ⇒ pos (succ p)
| neg p ⇒ match p with
            | one ⇒ zero
            | succ q ⇒ neg q
            end
end.
```

Indeed, we have that:

- succ$(0) = 1$
- For all integers $n \geq 1$, succ$(n) = n + 1$
- succ$(-1) = 0$
- For all integers $n \geq 2$, succ$(-n) = -(n - 1)$

Where succ is the successor function. In a similar way, our predecessor function can be defined as follows:

Type 3.2.3

```
Definition int_pred (i : Int) : Int :=
match i with
| zero ⇒ neg one
| pos p ⇒ match p with
            | one ⇒ zero
            | succ q ⇒ pos q
            end
| neg p ⇒ neg (succ p)
end.
```

Indeed, we have that:

- `pred`$(0) = -1$
- `pred`$(1) = 0$
- For all integers $n \geq 2$, `pred`$(n) = n - 1$
- For all integers $n \geq 1$, `pred`$(-n) = -(n + 1)$

Where `pred` is the predecessor function.

3.2.2 Proof of equivalence

In order to prove that these two functions form an equivalence, we need to prove that they cancel when composed either of two ways (see Section 2.1.6). As such, we need to prove

Type 3.2.4

```
Lemma int_succ_pred : int_succ o int_pred == idmap.
```

and

Type 3.2.5

```
Lemma int_pred_succ : int_pred o int_succ == idmap.
```

The proofs of these two statements proceed identically, and so we only consider the proof of the first lemma. We start by moving the argument `x` to the assumptions using the tactic `intro x`. After this, we apply induction on `x` to obtain the following three branches:

```
______________________________________(1/3)
int_succ (int_pred zero) = zero
______________________________________(2/3)
int_succ (int_pred (pos p)) = pos p
______________________________________(3/3)
int_succ (int_pred (neg p)) = neg p
```

For the first branch the definitions unfold to make both sides of the equality the same, and so we can immediately close the branch using `reflexivity`. The second and third branches are proven in quite similar ways (the only difference being whether `pos` or `neg` is used), and so we only consider the proof of the second branch. The way the definitions of this branch unfold depends on whether the positive number `p` is in the base case or in the inductive case. As such, we apply induction again on `p` to obtain the following two branches:

```
______________________________________(1/2)
int_succ (int_pred (pos one)) = pos one
______________________________________(2/2)
int_succ (int_pred (pos (succ p))) = pos (succ p)
```

For both of these branches their definitions unfold to make both sides of the equality the same, and so we can immediately close them off using `reflexivity`.

Now it must be shown that these functions form an equivalence. To do this, we must provide a construction of the following statement:

Type 3.2.6

```
Definition equiv_int_succ : Equiv Int Int.
```

In Coq-HoTT, an element of type `Equiv` can be constructed using the type `Build_Equiv` (see Section 2.1.6). We therefore start the proof using `simple refine (Build_Equiv _ _ _ _)`. This requires us to fill in two things:

1. A function used to construct the relevant equivalence.
2. A proof that this function forms an equivalence.

After slotting in the function `int_succ` in the first branch, we can use the tactic `simple refine (isequiv_adjointify _ _ _ _)` in the second branch. This allows us to construct an element of the required type by providing:

1. An inverse function to `int_succ`.
2. Proofs that `int_succ` cancels when composed with this inverse function either of two ways.

The key advantage here of using `isequiv_adjointify` is that we do not have to provide a proof that the two proofs of cancellation in Item 2 are analogous, which can be quite non-trivial.

In the branches created by the `simple refine` we slot in `int_pred`, `int_succ_pred` and `int_pred_succ` respectively. This completes the proof.

3.2.3 Integer addition

Next up is to define integer addition. We provide a definition by case distinction, giving us the following structure:

Type 3.2.7

```
Definition int_add (x y : Int) : Int :=
match x, y with
| zero, y ⇒ ?1?
| x, zero ⇒ ?2?
| pos p, pos q ⇒ ?3?
| neg p, neg q ⇒ ?4?
| pos p, neg q ⇒ ?5?
| neg p, pos q ⇒ ?6?
end.
```

Here, terms of the form ?_? indicate holes to be filled in. Once again, we define all cases to ensure symmetry in how the function is evaluated.

The addition of some number with zero is again that number. As such the first two cases return `y` and `x` respectively. For two integers with the same sign, their addition is that of their absolute values (embedded `Pos` value) modified with their sign. As such the third and fourth cases return `pos` (`pos_add p q`) and `neg` (`pos_add p q`) respectively. For two numbers with different sign we run into some problems. In these cases, we need to perform a subtraction operation of some sort on the numbers, which we have not yet defined. In order to get around this we define a new function `int_pos_sub` which takes two positive numbers `x` and `y` and returns $x - y$ as an integer. In the fifth and sixth cases we can then call `int_pos_sub p q` and `int_pos_sub q p` respectively, giving us the following definition:

Type 3.2.8

```
Definition int_add (x y : Int) : Int :=
match x, y with
| zero, y ⇒ y
| x, zero ⇒ x
| pos p, pos q ⇒ pos pos_add p q
| neg p, neg q ⇒ neg pos_add p q
| pos p, neg q ⇒ int_pos_sub p q
```

```
| neg p, pos q ⇒ int_pos_sub q p
end.
```

We define our type `int_pos_sub` as follows:

Type 3.2.9

```
Fixpoint int_pos_sub (x y : Pos) : Int :=
match x, y with
| one, one ⇒ zero
| succ p, one ⇒ pos p
| one, succ q ⇒ neg q
| succ p, succ q ⇒ int_pos_sub p q
end.
```

Indeed, we have the following computation rules for addition:

- $1 - 1 = 0$
- $(p + 1) - 1 = p$
- $1 - (q + 1) = -q$
- $(p + 1) - (q - 1) = p - q$

3.2.4 Commutativity of integer addition

Lastly for this chapter, we need to prove that integer addition is commutative. To do this, we provide a proof of the following proposition:

Type 3.2.10

```
Proposition int_add_comm : forall (x y : Int), int_add x y = int_add y x.
```

Applying induction on `x` and `y` through `induction x as [ |p1|p1], y as [ |p2|p2]` gives us the following nine branches:

```
_____________________________________(1/9)
int_add zero zero = int_add zero zero
_____________________________________(2/9)
int_add zero (pos p2) = int_add (pos p2) zero
_____________________________________(3/9)
int_add zero (neg p2) = int_add (neg p2) zero
```

```
______________________________________(4/9)
int_add (pos p1) zero = int_add zero (pos p1)
______________________________________(5/9)
int_add (pos p1) (pos p2) = int_add (pos p2) (pos p1)
______________________________________(6/9)
int_add (pos p1) (neg p2) = int_add (neg p2) (pos p1)
______________________________________(7/9)
int_add (neg p1) zero = int_add zero (neg p1)
______________________________________(8/9)
int_add (neg p1) (pos p2) = int_add (pos p2) (neg p1)
______________________________________(9/9)
int_add (neg p1) (neg p2) = int_add (neg p2) (neg p1)
```

In branches 1-4 and 6-8, the definitions unfold in such a way that we can immediately apply `reflexivity`. This leaves us with the goals in branches 5 and 9.

Unfolding the definitions of these branches using `cbn` gives us two very similar proof states. Branch 5 gives

```
p1, p2 : Pos
______________________________________(1/1)
pos (pos_add p1 p2) = pos (pos_add p2 p1)
```

while branch 9 gives

```
p1, p2 : Pos
______________________________________(1/1)
neg (pos_add p1 p2) = neg (pos_add p2 p1)
```

For both of these states, the core property that needs to be proven is that positive number addition commutes. Fortunately, a proof of this is already available in the form of our earlier `pos_add_comm`. Careful investigation of the goals shows us that, in fact, the only thing separating them from the statement `pos_add_comm p1 p2` is the application of `pos` in branch 5 and `neg` in branch 9. As such, these two branches can be closed off with the tactics `exact (ap pos (pos_add_comm p1 p2))` and `exact (ap neg (pos_add_comm p1 p2))` respectively.

We have now defined the basic types associated with the positive numbers and integers and can start applying the encode-decode principle.

Chapter 4

Encode-Decode

The encode-decode principle is a strategy often employed in homotopy type theory to characterize the path space of a certain type. It proves that a type has a certain h-level by characterizing the path space. In particular this is done by proving that the path space is equivalent to some type with an h-level that is one lower. For instance, in order to show that the integers form a set (a 0-type) the encode-decode principle requires one to show that the paths between integers are equivalent to some type of propositions (a (-1)-type).

The equivalence required by encode-decode is proven using `isequiv_adjointify` by showing that:

1. There exists a function which maps elements from the first type to the second (which we call `encode_`[`type`]).
2. There exists a function which maps elements from the second type to the first (which we call `decode_`[`type`]).
3. That these two functions cancel when composed either of two ways (`encode_decode_`[`type`] and `decode_encode_`[`type`])

This is sufficient to prove that the functions define an equivalence, which proves that the types they map to are equivalent.

If `isequiv_adjointify` is not used it also becomes necessary to prove that the proofs that the functions cancel are analogous. This is used to better characterize the higher path space, which becomes necessary when working with such an equivalence on a deeper level.

4.1 Positive Numbers

4.1.1 Positive Numbers as a Set

Before we develop the elementary patch theory, we must first prove that the positive numbers and the integers form a set. As discussed in the preliminaries (see Section 2.1.5), a set in homotopy type theory is also known as a 0-type, and in order to prove that some type is an n-type it suffices to show that the type of paths between elements of the type forms an

$(n-1)$-type. For the positive numbers and the integers, this implies that the goal is to prove that the paths between elements form a (-1)-type, or a mere proposition. To prove this, we use the encode-decode principle.

Let us start laying out some Coq code. The statement `hset_pos` that the positive numbers form a set is phrased in Coq as follows:

Type 4.1.1

```
Global Instance hset_pos : IsHSet Pos.
```

We use `Global Instance` here to set this property globally, enabling us to implicitly assume it in the future.

As mentioned previously, the statement here is equivalent to the statement that the paths between positive numbers form a (-1)-type. We can transform our goal into this by unfolding `IsHSet` using `intros x, y` and then transforming the internal (-1)-type into a regular one using `refine Trunc_is_trunc`. Coq automatically recognizes that a (-1)-type is a mere proposition due to how the latter is defined, and so we are left with the goal

```
x, y : Pos
______________________________________(1/1)
IsHProp (x = y)
```

As discussed in Section 2.1.6, `istrunc_equiv_istrunc` proves that a type is an `HProp` by proving that it is equivalent to an `HProp`. After applying the tactic `simple refine (istrunc_equiv_istrunc _ _)` we are left with the following three branches

```
x, y : Pos
______________________________________(1/3)
Type
______________________________________(2/3)
?A ≃ x = y
______________________________________(3/3)
IsHProp ?A
```

Here, the first branch requests the other type used in encode-decode, the second a proof of equivalence between the two types, and the third a proof that the type defined in the first branch is an `HProp`. In line with the naming conventions of encode-decode, we name the type of the first branch `code_Pos`.

As `code_Pos` has to be equivalent to the type of paths between positive numbers, its structure needs to be the same. That is, the type has to act on two positive numbers and be identifiable if and only if the two elements are equal. This motivates the following definition:

Type 4.1.2

```
Fixpoint code_Pos (x y : Pos) : Type :=
match x, y with
| one, one ⇒ Unit
| succ _, one ⇒ Empty
| one, succ _ ⇒ Empty
| succ p, succ q ⇒ code_Pos p q
end.
```

The numbers 1 and 1 are always uniquely equal to each other, and so the first case is mapped to the unit type for which we can always identify an element. The number 1 and some number greater than 1 are never equal to each other, and so the second and third cases are mapped to the empty type. By the inductive definition of `Pos`, equality of two numbers greater than 1 can be demonstrated by removing one instance of `succ` from both sides and applying the function again. This results in the recursive call in the fourth case. With the type `code_Pos` now fully defined, we can slot it into the first branch and move on to proving the statement in the second branch.

Next we show that `code_Pos` and the type of paths between positive numbers are equivalent. We express this using the following definition:

Type 4.1.3

```
Definition equiv_Pos_encode {x y : Pos} : Equiv (code_Pos x y) (x = y).
```

As noted in Section 2.1.6, to construct an element of type `Equiv A B` we use the function `Build_Equiv`. The type `Build_Equiv` takes an equivalence function `equiv_fun : A → B` and a proof `equiv_isequiv` that this function can be used to construct an equivalence. As such, we proceed with the tactic `simple refine (Build_Equiv _ _ _ _)`, giving us one branch to slot in `equiv_fun` and one to slot in `equiv_isequiv`. The type `equiv_isequiv` is not easily constructed however, and so we first need to unpack it further using the tactic `simple refine (isequiv_adjointify _ _ _ _)`. This results in the following four branches:

```
x, y : Pos
______________________________________(1/4)
code_Pos x y → x = y
______________________________________(2/4)
x = y → code_Pos x y
______________________________________(3/4)
?equiv_fun o ?g == idmap
______________________________________(4/4)
?g o ?equiv_fun == idmap
```

Here, ?equiv_fun refers to the function of the first branch and ?g to the function of the second branch. In line with the encode-decode principle, we call the functions to be slotted into these branches decode_pos, encode_pos, encode_decode_pos and decode_encode_pos respectively. This completes the basics of our proof that the positive numbers form a set.

4.1.2 Encoding and Decoding

The type encode_pos takes a path between two positive numbers and return an element of code_Pos. As such, it has the following signature:

Type 4.1.4

```
Definition encode_pos (x y : Pos) (p : x = y) : code_Pos x y.
```

As we have a path in the assumptions, we can proceed by path induction using the tactic induction p (see Section 2.2.2). This then gives us the goal

```
x : Pos
______________________________________(1/1)
code_Pos x x
```

The result of code_Pos depends on the value of its argument x. As such we proceed by induction on this argument to obtain:

```
______________________________________(1/2)
code_Pos one one
______________________________________(2/2)
code_Pos (succ x) (succ x)
```

In the first branch the goal unfolds to Unit, for which we can always find an element; tt. We prove this using the tactic exact tt. In the second branch, we have the following situation:

```
x : Pos
IHx : code_Pos x x
______________________________________(1/1)
code_Pos (succ x) (succ x)
```

Unfolding the goal using cbn yields code_Pos x x, which is exactly our inductive hypothesis. As such we can close off the branch using exact IHx.

The function `decode_pos` has the following signature:

Type 4.1.5

```
Definition decode_pos : forall (x y : Pos), code_Pos x y → (x = y).
```

Given that the value of `code_Pos` depends on the value of the positive numbers, we start with induction on these numbers using `induction x, y`. This gives the following branches:

```
_____________________________________(1/4)
code_Pos one one → one = one
_____________________________________(2/4)
code_Pos one (succ y) → one = succ y
_____________________________________(3/4)
code_Pos (succ x) one → succ x = one
_____________________________________(4/4)
code_Pos (succ x) (succ y) → succ x = succ y
```

In each of these branches, we start by applying `intro c` in order to put the `code_Pos` instance we are given into the assumptions. In the first branch, doing this immediately yields a goal that we can solve using `reflexivity`. In the second and third branches, the `code_Pos` instance unfolds to the empty type, and so we can close it off with empty-induction using `induction c` (see Section 2.1.3). In the fourth branch, unfolding the assumption `c` using `cbn in c` gives us in the following proof state:

```
x : Pos
IHx : forall y : Pos, code_Pos x y → x = y
y : Pos
c : code_Pos x y
_____________________________________(1/1)
succ x = succ y
```

As can be seen, we can now apply the inductive hypothesis by using `c` as an argument. We create a new assumption `p : x = y` with the result of this application using the tactic `pose (IHx y c) as p`. This assumption only differs from the goal by the fact that it lacks an application of `succ` on both sides. Therefore, we can close of the branch using the tactic `exact (ap succ p)`. This completes the construction of `encode_pos` and `decode_pos`.

4.1.3 Cancellation of Encode and Decode

As per the encode-decode principle, we now need to prove that `encode_pos` and `decode_pos` cancel when composed either of two ways. This is expressed using the types `encode_decode_pos` and `decode_encode_pos`. The type `encode_decode_pos` has the following signature:

Type 4.1.6

```
Lemma encode_decode_pos (x y : Pos) :
  decode_pos x y o encode_pos x y == idmap.
```

As with `int_succ_pred` and `int_pred_succ` in Section 3.2, we start by extracting the argument using `intro p`. This argument is a path, and so we can apply path induction using `induction p` to obtain the following:

```
x : Pos
______________________________________(1/1)
decode_pos x x (encode_pos x x 1) = 1
```

Both `encode_pos` and `decode_pos` are constructed using induction on their argument, and so we need to apply induction on `x` here as well. This leaves us with the following branches:

```
______________________________________(1/2)
decode_pos one one (encode_pos one one 1) = 1
______________________________________(2/2)
decode_pos (succ x) (succ x) (encode_pos (succ x) (succ x) 1) = 1
```

In the first branch the definitions unfold in such a way as to allow us to immediately apply `reflexivity`. In the second branch we have the following situation:

```
x : Pos
IHx : decode_pos x x (encode_pos x x 1) = 1
______________________________________(1/1)
decode_pos (succ x) (succ x) (encode_pos (succ x) (succ x) 1) = 1
```

It may seem like applying `cbn` would be a logical first step here, with the hope that the definitions once again unfold in such a way that we can apply the inductive hypothesis. However, as `encode_pos` and `decode_pos` are defined in a proof-like style rather than a function-like style, `cbn` would in this case also expand the functions themselves to their proofs. Instead it is necessary to apply the less thorough tactic `simpl` (see Section 2.2.2). This results in the following:

```
x : Pos
IHx : decode_pos x x (encode_pos x x 1) = 1
______________________________________(1/1)
ap succ (decode_pos x x (Pos_rect
  (fun x0 : Pos ⇒ code_Pos x0 x0) tt (fun x0 : Pos ⇒ idmap) x)) = 1
```

While this may still seem to be too far 'folded out' to properly apply the inductive hypothesis, Coq's parser is intelligent enough to realize that the `Pos_rect` is equivalent to an application of `encode_pos`. Since it also holds that `ap succ 1` is equal to 1, we can close off the branch using the tactic `exact (ap (ap succ) IHx)`.

The type `decode_encode_pos` has the following signature:

Type 4.1.7

```
Lemma decode_encode_pos : forall (x y : Pos), encode_pos x y o decode_pos x y == idmap.
```

We start by introducing all arguments, giving us the following state:

```
x, y : Pos
p : code_Pos x y
______________________________________(1/1)
encode_pos x y (decode_pos x y p) = p
```

We know that `code_Pos` is a mere proposition. As such it holds that any element of type `code_Pos v w`, with `v w : Pos`, is equal to any other element of the same type. We know that both sides of the equality are elements of the type `code_Pos x y`, and so we can use this property to immediately close the branch. In this case this is done through the tactic `apply path_ishprop`. This completes the construction of `encode_decode_pos` and `decode_encode_pos`.

4.1.4 Completing the Proof

At the end of Section 4.1.1, we obtained four branches in which the types `encode_pos`, `decode_pos`, `decode_encode_pos` and `encode_decode_pos` needed to be slotted. Now that we have all of these types fully defined, we can slot them in using `apply`'s. This gives us the following final proof of equivalence:

Type 4.1.8

```
Definition equiv_Pos_encode {x y : Pos} : Equiv (code_Pos x y) (x = y).
Proof.
simple refine (Build_Equiv _ _ _ _).
2: simple refine (isequiv_adjointify _ _ _ _).
- apply decode_pos.
- apply encode_pos.
- apply encode_decode_pos.
- apply decode_encode_pos.
Defined.
```

We now only need a proof that `code_Pos` is a mere proposition. Luckily, this task is not as daunting as it initially might seem. The proof that a set is a mere proposition is written down in Coq syntax as follows:

Type 4.1.9

```
Global Instance IsHProp_code_pos : forall (x y : Pos), IsHProp (code_Pos x y).
```

Here, we make use of a `Global Instance`. This typeclass is usually employed to specify the ambient environment of Coq-HoTT, allowing additional tactics to be used without any hassle. In this case, the Global Instance can be used to allow Coq to find the mere proposition property itself any time some function requires it. Additionally, any time the property is all that is required to satisfy the goal it allows for closing the branch using `apply _`.

By applying induction on the two arguments given we obtain the following four branches:

```
______________________________________(1/4)
IsHProp (code_Pos one one)
______________________________________(2/4)
IsHProp (code_Pos one (succ y))
______________________________________(3/4)
IsHProp (code_Pos (succ x) one)
______________________________________(4/4)
IsHProp (code_Pos (succ x) (succ y))
```

We can use the tactic `apply _` in all of these branches to let Coq verify by itself that both of the types that `code_Pos` can unfold to (namely `Unit` and `Empty`) are mere propositions. With this, we are immediately done with the proof and can start filling in the branches of `hset_pos`.

In the type `hset_pos`, we were left with the following three branches at the end of Section 4.1.1:

```
x, y : Pos
______________________________________(1/3)
Type
______________________________________(2/3)
?A ≃ x = y
______________________________________(3/3)
IsHProp ?A
```

Here, `?A` indicates the value of the first branch. The first branch was the type we wanted to apply encode-decode with, which was `code_Pos`. We indicate this using the tactic `apply (code_Pos x y)`. The second branch was the proof of equivalence between `code_Pos` and paths between positive numbers, which we stored in `equiv_Pos_encode`. As such we use `apply equiv_Pos_encode` to complete this branch. The last branch was the proof that `code_Pos` is a mere proposition, which we stored in `hset_pos`. While we could close this off using `apply hset_pos`, it is more usual to let Coq find the proof itself using `apply _`. This completes the proof that the positive numbers form a set.

4.2 Integers

4.2.1 Integers as a Set

In the previous chapter, we defined the integers as follows:

Type 4.2.1

```
Inductive Int : Type :=
| zero : Int
| pos : Pos → Int
| neg : Pos → Int.
```

In order to prove that this type forms a set, we can use the same approach we used to prove that the positive numbers formed a set. As such we create a new definition `hset_int` with the following definition and proof, where `code_Int` (Type 4.2.3), `equiv_Int_encode` (Type 4.2.11) and `IsHProp_code_int` (Type 4.2.12) are functions to be defined.

Type 4.2.2

```
Global Instance hset_int : IsHSet Int.
Proof.
intros x y; refine Trunc_is_trunc.
simple refine (istrunc_equiv_istrunc _ _).
- apply (code_Int x y).
- apply equiv_Int_encode.
- apply IsHProp_code_int.
Defined.
```

For `code_Int` we derive the following definition:

Type 4.2.3

```
Definition code_Int (x y : Int) : Type :=
match x, y with
| zero, zero    ⇒ Unit
| zero, pos _   ⇒ Empty
| zero, neg _   ⇒ Empty
| pos _, zero   ⇒ Empty
| pos p, pos q  ⇒ code_Pos p q
| pos _, neg _  ⇒ Empty
| neg _, zero   ⇒ Empty
| neg _, pos _  ⇒ Empty
| neg p, neg q  ⇒ code_Pos p q
end.
```

Since we once again encode to paths, this definition is quite similar to that of `code_Pos`. Once again, we map equal elements to the unit type and inequal elements to the empty type. For the cases (`pos p`), (`pos q`) and (`neg p`), (`neg q`), we now call `code_Pos` since whether the two integers are equal depends entirely on whether the numbers they contain as arguments are equal.

4.2.2 Encoding and Decoding

Analogously to `encode_pos`, `encode_int` has the following signature:

Type 4.2.4

```
Definition encode_int (x y : Int) (p : x = y) : code_Int x y.
```

Given that we have a path in the assumptions, we can apply path induction in order to only have to define the reflexive case. This changes the proof state to the following:

```
x : Int
------------------------------------(1/1)
code_Int x x
```

In order to prevent issues with the definition unfolding too far to be readable, we proceed to make a new lemma `encode_int_refl` for this goal using case distinction. Afterwards, we can complete the branch using the tactic `apply encode_int_refl`.

In the case where the argument is `zero`, we want the function to return an element of `code_Int zero zero` ≡ `Unit`. As such we return `tt`. In the cases where the argument is `pos p` or `neg p`, we need to return an element of type `code_Pos p p`. As such we return `encode_pos p p 1`.

This gives us the following definition:

Type 4.2.5

```
Definition encode_int_refl (x : Int) : code_Int x x :=
match x with
| zero ⇒ tt
| (pos p) ⇒ encode_pos p p 1
| (neg p) ⇒ encode_pos p p 1
end.
```

With this, we have defined the type `encode_int`.

The type `decode_int` has the following signature:

Type 4.2.6

```
Definition decode_int : forall (x y : Int), code_Int x y → (x = y).
```

Same as in `decode_pos`, we perform induction on the two arguments `x` and `y` using the tactic `induction x as [ |p1|p1], y as [ |p2|p2]`. Here, the `as`-clauses are used to give the values of type `Pos` embedded in the `pos` and `neg` numbers proper names. We also apply the tactic `intro c` in each branch in order to have our `code_Int` argument as an assumption rather than the condition of an implication. This gives the following nine branches:

```
c : code_Int zero zero
______________________________________(1/9)
zero = zero
______________________________________(2/9)
zero = pos p2
______________________________________(3/9)
zero = neg p2
______________________________________(4/9)
pos p1 = zero
______________________________________(5/9)
pos p1 = pos p2
______________________________________(6/9)
pos p1 = neg p2
______________________________________(7/9)
neg p1 = zero
______________________________________(8/9)
neg p1 = pos p2
______________________________________(9/9)
neg p1 = neg p2
```

In branches 2, 3, 4, 6, 7, and 8, the definition of `code_Int` unfolds to the empty type. As such, we can close these branches off using empty-induction. In branch 1, we need to return an element of `zero = zero`. As this is a reflexive path, we can return 1 using the tactic `exact 1`. In branches 5 and 9, unfolding the `code_Int` instance using `cbn in c` gives the following proof state, where `pos` is replaced by `neg` in branch 9:

```
p1, p2 : Pos
c : code_Pos p1 p2
______________________________________(1/1)
pos p1 = pos p2
```

Given that we have a `code_Pos` instance here, we can try to use `decode_pos` to transform it into its path equivalent. Doing this using the tactic `pose (decode p1 p2 c) as p` gives a new assumption `p` of type `p1 = p2`. This assumption only differs from the goal by an application of `pos` respectively `neg` on both sides, and so we can close the branches off using `exact (ap pos p)` and `exact (ap neg p)` respectively. This completes the construction of `encode_int` and `decode_int`.

4.2.3 Cancellation of Encode and Decode

The type `encode_decode_int` has the following signature:

Type 4.2.7

```
Lemma encode_decode_int (x y : Int) : decode_int x y o encode_int x y == idmap.
```

We start by extracting the argument using `intro p`. After this we apply path induction using `induction p` to reduce the goal to the case where the arguments are equal, followed by induction on this argument using `induction x`. This results in the following three branches:

```
______________________________________(1/3)
decode_int zero zero (encode_int zero zero 1) = 1
______________________________________(2/3)
decode_int (pos p) (pos p) (encode_int (pos p) (pos p) 1) = 1
______________________________________(3/3)
decode_int (neg p) (neg p) (encode_int (neg p) (neg p) 1) = 1
```

In the first branch, the definitions unfold to immediately give reflexivity. The second and third branches once again only differ by whether `pos` or `neg` is used, and so their proofs proceed largely identically. As such we only consider the proof for the second branch. In this branch we have the following after applying `cbn -[encode_int_refl]`:

```
p : Pos
______________________________________(1/1)
ap pos (decode_pos p p (encode_int_refl (pos p))) = 1
```

The type `decode_pos` always returns a path, and so we can apply path induction on its argument `p`. This results in the following two branches:

```
______________________________________(1/2)
ap pos (decode_pos one one (encode_int_refl (pos one))) = 1
______________________________________(2/2)
ap pos (decode_pos (succ p) (succ p) (encode_int_refl (pos (succ p)))) = 1
```

In the first branch the definitions again unfold to immediately give reflexivity. In the second branch, applying `cbn` unfolds the definitions too much and so we apply `simpl`. This gives the following:

```
p : Pos
IHp : ap pos (decode_pos p p (encode_int_refl (pos p))) = 1
______________________________________(1/1)
ap pos (ap succ (decode_pos p p (Pos_rect (fun x : Pos ⇒ code_Pos x x)
  tt (fun x : Pos ⇒ idmap) p))) = 1
```

We want to apply our inductive hypothesis here, but are prevented because we are not able to move the `ap succ` outside of the `ap pos` (in the third branch we would have the same problem with `neg`). To resolve this, we create a lemma `pos_succ_change` to prove that we can indeed do this. Our lemma is of the following form:

Type 4.2.8

```
Lemma pos_succ_change (x y : Pos) (p : x = y) : ap pos (ap succ p) = ap int_succ (ap pos p).
```

As we have a path in the assumptions, we can try applying path induction to obtain the following:

```
x : Pos
______________________________________(1/1)
ap pos (ap succ 1) = ap int_succ (ap pos 1)
```

At this point the left and right sides unfold to the same term, allowing us to close off the proof using `reflexivity`.

After applying our lemma using the tactic `rewrite pos_succ_change`, we obtain the following:

```
p : Pos
IHp : ap pos (decode_pos p p (encode_int_refl (pos p))) = 1
______________________________________(1/1)
ap int_succ (ap pos (decode_pos p p (Pos_rect
  (fun x : Pos ⇒ code_Pos x x) tt (fun x : Pos ⇒ idmap) p))) = 1
```

As we have that `ap int_succ 1` is equal to 1, we can close off the branch using `exact (ap (ap int_succ) IHp)`. This completes the proof of `encode_decode_int`.

The type `decode_encode_int` has the following signature:

Type 4.2.9

```
Lemma decode_encode_int (x y : Int) : decode_int x y o encode_int x y == idmap.
```

We start by applying induction on the two arguments `x` and `y` using `induction x as [ | p1|p1]`, `y as [ | p2|p2]`, followed by bringing `c` into the assumptions in all branches using `intro c`. This results in the following nine branches:

```
c : code_Int zero zero
_____________________________________(1/9)
encode_int zero zero (decode_int zero zero c) = c
_____________________________________(2/9)
encode_int zero (pos p2) (decode_int zero (pos p2) c) = c
_____________________________________(3/9)
encode_int zero (neg p2) (decode_int zero (neg p2) c) = c
_____________________________________(4/9)
encode_int (pos p1) zero (decode_int (pos p1) zero c) = c
_____________________________________(5/9)
encode_int (pos p1) (pos p2) (decode_int (pos p1) (pos p2) c) = c
_____________________________________(6/9)
encode_int (pos p1) (neg p2) (decode_int (pos p1) (neg p2) c) = c
_____________________________________(7/9)
encode_int (neg p1) zero (decode_int (neg p1) zero c) = c
_____________________________________(8/9)
encode_int (neg p1) (pos p2) (decode_int (neg p1) (pos p2) c) = c
_____________________________________(9/9)
encode_int (neg p1) (neg p2) (decode_int (neg p1) (neg p2) c) = c
```

For branches 2-4 and 6-8 the assumption `c` unfolds to the empty type, and so we can immediately close them off using empty-induction with `induction c`. For the first branch, unfolding the definitions in the goal as well as `c` using the tactics `cbn in c`; `cbn` gives us the following:

```
c : Unit
_____________________________________(1/1)
tt = c
```

The only element of `Unit` is `tt`, and so we can close this branch off with `induction c` followed by `reflexivity`.

In the fifth branch, unfolding the definitions gives the following:

```
p1, p2 : Pos
c : code_Pos p1 p2
_____________________________________(1/1)
encode_int (pos p1) (pos p2) (ap pos (decode_pos p1 p2 c)) = c
```

Here, we see that while `decode_int (pos p1) (neg p2) c` did unfold to `ap pos (decode_pos p1 p2 c)`, a similar thing did not happen to `encode_int`. We can make this happen by creating and

applying a lemma `encode_int_pos` with the following signature:

Type 4.2.10

```
Lemma encode_int_pos {x y : Pos} {p : x = y}
  : encode_int (pos x) (pos y) (ap pos p) = encode_pos x y p.
```

This lemma immediately holds by `reflexivity` after applying path induction on `p`. Applying the lemma to our goal gives the following:

```
p1, p2 : Pos
c : code_Pos p1 p2
______________________________________(1/1)
encode_pos p1 p2 (decode_pos p1 p2 c) = c
```

This is exactly `decode_encode_pos`, and so we can close off the branch using the tactic `apply decode_encode_pos`. The proof of the ninth branch once again proceeds in much the same way, with us constructing an identical lemma for `neg` and finding that the resulting goal is equal to `decode_encode_pos`. As such, we will not cover this proof in detail. With this, we have reached the end of the proofs that `encode_int` and `decode_int` cancel.

4.2.4 Completing the Proof

Our final proof of equivalence proceeds in much the same way as that in Section 4.1.4, with us slotting in respectively `decode_int`, `encode_int`, `decode_encode_int` and `encode_decode_int`. This gives us the following construction:

Type 4.2.11

```
Definition equiv_Int_encode {x y : Int} : Equiv (code_Int x y) (x = y).
Proof.
simple refine (Build_Equiv _ _ _ _).
2: simple refine (isequiv_adjointify _ _ _ _).
- apply decode_int.
- apply encode_int.
- apply encode_decode_int.
- apply decode_encode_int.
Defined.
```

The proof that `code_Int` is a mere proposition also proceeds in the same way as `code_Pos`, giving the following construction:

Type 4.2.12

```
Global Instance IsHProp_code_int : forall (x y : Int), IsHProp (code_Int x y).
Proof.
induction x, y; apply _.
Defined.
```

With this, we have defined all components necessary to construct `hset_int` as elaborated upon in Section 4.2.1 and have completed the application of the encode-decode principle for the integers.

Chapter 5

Patch Theory

In this chapter we build on the foundations constructed in Chapters 3 and 4 to construct the first patch theory outlined in the paper *Homotopical patch theory* [13]. We construct the higher inductive type of the first patch theory (Section 5.1), construct interpreters for integers and booleans (Section 5.2), and prove all the properties outlined in the paper (Section 5.2.1). In particular, we prove the following properties:

1. `addsucc`; that the interpretation of the path operation in the integers is the successor operations for integers (Section 5.2.1).
2. `addVsucc`; that the interpretation of the inverse path operation in the integers is the predecessor operation for integers (Section 5.2.1).
3. `intcomp`; that the concatenation of two path operations are interpreted in the integers as the concatenation of the interpretations of the two operations (Section 5.2.1).
4. `intzero`; that the interpretation of the path operation with its inverse leaves the integer zero unchanged (Section 5.2.1).
5. `boolrefl1`; that the interpretation of the path operation in the booleans cancels when composed with itself (Section 5.2.1).

5.1 The HIT `PT1`

Now that we have a useful model for the integers, it becomes possible to define the higher inductive type representing patches in the first patch theory. We call this HIT `PT1` and define it analogously to the first type `R` in *Homotopical patch theory*. The type `PT1` is a higher inductive type containing a point constructor `num` as well as a path constructor `add1`. In Coq this becomes the following (see Section 2.2.3 for details):

Type 5.1.1

```
Module Export PT1.
Private Inductive PT1 : Type :=
| num : PT1.
Axiom add1 : num = num.
```

As we have a higher inductive type, we require both a recursion principle and an induction principle (see also Section 2.2.3). It would be possible to define the recursion in terms of the induction principle since it is strictly weaker, however as this would result in some more work later on we do not do this. The recursion and induction principles are implemented as follows:

Type 5.1.2

```
Definition PT1_rec {A : Type} (a : A) (p : a = a) (n : PT1) : A :=
match n with
| num ⇒ a
end.
Axiom PT1_rec_add1 : forall {A : Type} (a : A) (p : a = a),
                     ap (PT1_rec a p) add1 = p.

Definition PT1_ind (P : PT1 → Type) (b : P num) (l : add1 # b = b) (x : PT1) : P x :=
match x with
| num ⇒ b
end.
Axiom PT1_ind_add1 : forall (P : PT1 → Type) (b : P num) (l : add1 # b = b),
                     apD (PT1_ind P b l) add1 = l.
End PT1.
```

It should be noted that these types are defined in a module named PT1. This is to make the higher inductive type opaque, which prevents exposing that the type only has one element through making the induction principle invisible. With this, the type is able to accommodate more complex patches than merely `add1` by not specifying the right side of the equality.

5.2 Interpreters

With the higher inductive type defined we can create some interpreters. For this we use the recursion principle with the tactic `simple refine (PT1_rec _ _)`, where we supply:

1. A point `x`.
2. A path `x = x` between points.

We use this to define our basic interpreter `I`. After that we can use `I` to manually define the corresponding interpreter `interp` for equivalences. For the interpreter of integers, our point is the type of integers and the path the univalence axiom applied to the successor function.

This gives the following function:

Type 5.2.1

```
Definition I : PT1 → Type.
Proof.
simple refine (PT1_rec _ _).
- exact Int.
- exact (path_universe_uncurried equiv_int_succ).
Defined.

Definition interp : (num = num) → (Int ≃ Int) :=
fun patch ⇒ equiv_path Int Int (ap I patch).
```

With this, our patch theory `PT1` can be interpreted as the type of integers with addition. In other words, we can interpret the nodes in our patch theory as being arbitrary integers and the operation of moving between nodes as integer addition.

We also define an interpreter for the booleans. In this type, our point is the type of booleans and the path the univalence axiom applied to the negation function. This gives the following types:

Type 5.2.2

```
Definition I' : PT1 → Type.
Proof.
simple refine (PT1_rec _ _).
- exact Bool.
- exact (path_universe_uncurried equiv_negb).
Defined.

Definition interp' : (num = num) → (Bool ≃ Bool) :=
fun patch ⇒ equiv_path Bool Bool (ap I' patch).
```

With this, our patch theory `PT1` can be interpreted as the booleans with negation. The nodes are now the values `true` and `false`, which we can move between using the negation operation.

5.2.1 Properties of Interpreters

Using these interpreters, all the properties posed in the paper by Anguili, Morehouse, Licata and Harper can be proven. We start with `addsucc`, which can be written as the following proposition:

Type 5.2.3

```
Proposition addsucc : interp add1 = equiv_int_succ.
```

We would like to introduce an argument on both sides of the equality. However, due to both sides being an equivalence attempting this using the tactic `funext x` fails. Instead, we need to first extract the function implicit in the equivalence using `apply path_equiv`. After that we can indeed apply `funext x` followed by `cbn` to obtain the following:

```
x : Int
______________________________________(1/1)
transport idmap (ap I add1) x = int_succ x
```

In our definition of `I`, we defined the path constructor `add1` of the higher inductive type to correspond to `path_universe_uncurried equiv_int_succ`. As such our term `ap I add1` should translate to `path_universe_uncurried equiv_int_succ`. Indeed, when we apply the tactic `rewrite PT1_rec_add1`, we obtain the following goal:

```
x : Int
______________________________________(1/1)
transport idmap (path_universe_uncurried equiv_int_succ) x = int_succ x
```

At this point, the tactic `transport_path_universe_uncurried` can be used to close off the branch using the tactic `exact (transport_path_universe_uncurried equiv_int_succ x)` (see Section 2.1.7).

While the property is not listed in the paper *Homotopical patch theory*, it will later on also be necessary to prove an analogous statement that the interpretation of the inverse of `add1` is the inverse of the successor function. As such we also include the proof of this property here. The property can be phrased as the following proposition in Coq:

Type 5.2.4

```
Proposition addVsucc : interp (add1^) = (equiv_inverse equiv_int_succ).
```

It should be the case that the interpretation of the inverse of some action on patches corresponds to the inverse in the corresponding interpretation. For this we create the following lemma:

Type 5.2.5

```
Lemma interp_inverse (p : num = num) : interp p^ = equiv_inverse (interp p).
```

Unfolding `interp` here using the tactic `unfold interp` gives us the following:

```
p : num = num
______________________________________(1/1)
equiv_path Int Int (ap I p^) = ((equiv_path Int Int (ap I p))^-1)%equiv
```

The goal here is to move the inverse first outside the application of `ap` and then outside the application of `equiv_path`. This we can do with the tactics `rewrite ←  inverse_ap` and `apply equiv_path_V` respectively. The latter tactic also closes the branch, completing the proof.

We can use the just-created lemma in `addVsucc` using the tactic `rewrite (interp_inverse add1)`. This gives the following:

```
______________________________________(1/1)
((interp add1)^-1)%equiv = (equiv_int_succ^-1)%equiv
```

This is exactly the Type 5.2.4 with an application of the inverse on both sides, and so we can close the branch off using the tactic `exact (ap equiv_inverse addsucc)`.

The next property we need to prove is `intcomp`, which is defined as the following proposition:

Type 5.2.6

```
Proposition intcomp : forall (p q : num = num), interp (q @ p) = (interp p) oE (interp q).
```

Unfolding the `interp` function results in the following proof state:

```
p, q : num = num
______________________________________(1/1)
equiv_path Int Int (ap I (q @ p)) =
  equiv_path Int Int (ap I p) oE equiv_path Int Int (ap I q)
```

Now, the goal is clearly to move the composition operation outside of both the `ap` function and the `equiv_path` function. In order to move it out of the `ap` function, we can use the tactic `rewrite (ap_pp I q p)`. This gives the following:

```
p, q : num = num
______________________________________(1/1)
equiv_path Int Int (ap I q @ ap I p) =
  equiv_path Int Int (ap I p) oE equiv_path Int Int (ap I q)
```

To move the composition outside of `equiv_path` and close off the branch, we use the tactic `exact (equiv_path_pp (ap I q) (ap I p))`.

The next property we need to prove is `intzero`, with the following signature:

Type 5.2.7

```
Proposition intzero : interp (add1 @ add1^) zero = zero.
```

A function composed with its inverse should not result in any change. We indicate this fact using the tactic `rewrite concat_pV` to obtain the following:

```
______________________________________(1/1)
interp 1 zero = zero
```

This goal then holds immediately through reflexivity, since `interp` maps 1 to an identity mapping for the integers and so the value `zero` remains unchanged.

Lastly, we need to prove the property `boolrefl1`. This type has the following signature:

Type 5.2.8

```
Proposition boolrefl1 : (ap I' add1) @ (ap I' add1) = 1.
```

Applying the computation rule for `add1` gives us the following goal:

```
______________________________________(1/1)
path_universe_uncurried equiv_negb @ path_universe_uncurried equiv_negb = 1
```

We would like to move the composition here to within the application of `path_universe_uncurried` so we can compose the two applications of `equiv_negb` together. This can be done using the tactic `rewrite` ← `(path_universe_compose_uncurried equiv_negb equiv_negb)`. This results in the following goal:

```
______________________________________(1/1)
path_universe_uncurried (equiv_compose equiv_negb equiv_negb) = 1
```

In order to be able to use the property that double negation cancels we need to replace this `path_universe_uncurried` with a `path_universe`. There is no set tactic for this, and so we instead `replace` the left hand side and then immediately prove the second branch using the tactic 2: `reflexivity`.

In order to proceed from this point on we need to show that a double negation results in net no action. For this we create a new lemma `equivs_cancel`. This lemma has the following signature:

Type 5.2.9

```
Lemma equivs_cancel : (equiv_negb oE equiv_negb) == 1%equiv.
```

For this lemma we can use the tactic `intros [ | ]`. This essentially starts a proof by case distinction, and since we are working with booleans here we obtain the following two branches:

```
______________________________________(1/2)
(equiv_negb oE equiv_negb) true = 1%equiv true
______________________________________(2/2)
(equiv_negb oE equiv_negb) false = 1%equiv false
```

Both of these we can immediately close off using `reflexivity`, completing the proof.

We can now apply the lemma to the goal using the tactic `rewrite (path2_universe equivs_cancel)`. This results in the following goal:

```
______________________________________(1/1)
path_universe 1%equiv = 1
```

This is proven in the proposition `path_universe_1`, and so we close off the proof using the tactic `apply path_universe_1`. With this, we have proven all properties in *Homotopical Patch Theory* that are necessary to construct the merge principle.

Chapter 6

Merge

Before we are able to properly tackle the merge principle as described in the paper *Homotopical patch theory*, we require two key components:

- An application of the encode-decode principle on the patch theory (Types 6.2.3 and 6.2.4).
- A proof that `decode` translates composition of patch histories to integer addition (Type 6.3.1).

In the first three sections of this chapter, we define these principles and create the scaffolding that will later be useful to prove some of the more challenging propositions that the merge principle requires. In the following two, we implement the merge operation itself along with its two key properties; the reconcile principle stating that the merge operation indeed unifies the two patches and the symmetry principle stating that merging is commutative.

6.1 The Encode-Decode Principle

We define our function `encode` as follows:

Type 6.1.1

```
Definition encode {x : PT1} : (num = x) → (I x) :=
fun p ⇒ transport I p zero.
```

In this function, we first transport our earlier-defined interpreter over the patch, translating the patch to an operation +n: `Int` → `Int` which maps each integer `x` to the integer `x`+`n`. After this, we apply this operation to zero in order to obtain the integer `n`. We choose to phrase the argument here as an element of `num = x` instead of `num = num` in order to be able to use path induction and enable it to be applied to arbitrary patch histories.

We define the function `decode` as follows:

Type 6.1.2

```
Definition decode {x : PT1} : (I x) → (num = x).
```

While we would like to use integer induction here, we cannot since the interpreter is applied to an arbitrary element rather than `num`. We can get around this by first using the induction principle of the patch theory, however this means that we also need to provide an analogue function for the element `add1`. Applying the induction principle using the tactics `revert x` and `simple refine (PT1_ind _ _ _)` thus results in the following two branches:

```
______________________________________(1/2)
(fun x : PT1 ⇒ I x → num = x) num
______________________________________(2/2)
transport (fun x : PT1 ⇒ I x → num = x) add1 ?b = ?b
```

For the first branch, simplifying the goal using `cbn` gives us the following:

```
______________________________________(1/1)
Int → num = num
```

We need to provide a function with the given signature, which we call `looppow`. We define the function as follows:

Type 6.1.3

```
Definition looppow (i : Int) : (num = num) :=
match i with
| zero ⇒ 1
| pos p ⇒ looppow_pos p
| neg p ⇒ (looppow_pos p)^
end.
```

where `looppow_pos` is defined as follows:

Type 6.1.4

```
Fixpoint looppow_pos (p : Pos) : (num = num) :=
match p with
| one ⇒ add1
| (succ p) ⇒ (looppow_pos p) @ add1
end.
```

Later on, we will need to show that these functions commute with `add1`. We define the type `looppow_pos_comm_add1` as follows:

Type 6.1.5

```
Lemma looppow_pos_comm_add1 (p : Pos) : add1 @ (looppow_pos p) = (looppow_pos p) @ add1.
```

We start off with induction on the argument `p`. This gives us the following two branches:

```
______________________________________(1/2)
add1 @ looppow_pos one = looppow_pos one @ add1
______________________________________(2/2)
add1 @ looppow_pos (succ p) = looppow_pos (succ p) @ add1
```

The first branch holds immediately through `reflexivity`. For the second branch, we transform `looppow_pos (succ p)` into `looppow_pos p @ add1` using an application of `enough`. This gives us the following:

```
p : Pos
IHp : add1 @ looppow_pos p = looppow_pos p @ add1
______________________________________(1/1)
add1 @ (looppow_pos p @ add1) = (looppow_pos p @ add1) @ add1
```

We next use the tactic `concat_p_pp` to make the left side `(add1 @ looppow_pos p) @ add1`, then remove the concatenation with `add1` on both sides using the tactic `refine (ap (fun z ⇒ z @ add1) _)`. This gives us the goal `add1 @ looppow_pos p = looppow_pos p @ add1`, which is exactly the inductive hypothesis.

We define the type `looppow_comm_add1` as follows:

Type 6.1.6

```
Lemma looppow_comm_add1 (p : Pos) : add1 @ (looppow (pos p)) = (looppow (pos p)) @ add1.
```

Unfolding using the tactic `cbn -[looppow_pos]` gives us the goal `add1 @ looppow_pos p = looppow_pos p @ add1`, which immediately holds through `looppow_pos_comm_add1`. As such we can close off the branch using the tactic `exact (looppow_pos_comm_add1 p)`.

We can now provide `looppow` to the first branch of the definition of `decode` using the tactic `exact looppow`. The second branch of this definition then becomes the following:

```
______________________________________(1/1)
transport (fun x : PT1 ⇒ I x → num = x) add1
  (looppow : (fun x : PT1 ⇒ I x → num = x) num) =
(looppow : (fun x : PT1 ⇒ I x → num = x) num)
```

Now, we can move the argument given to `looppow` to the assumptions using `funext i` and `intro i` and change it from `I num` to `Int` using `cbn in i`. This gives us the following:

```
i : Int
______________________________________(1/1)
transport (fun x : PT1 ⇒ I x → num = x) add1 looppow i = looppow i
```

We can derive the left hand side here as follows:

```
transport (fun x : PT1 ⇒ I x → num = x) add1 looppow i
= ⟨rewrite transport_arrow⟩
transport (paths num) add1 (looppow (transport I add1^ i)
= ⟨rewrite transport_paths_r⟩
looppow (transport I add1^ i) @ add1
= ⟨rewrite transport_idmap_ap⟩
looppow (transport idmap (ap I add1^) i) @ add1
= ⟨replace by path_universe_uncurried_equiv_path⟩
looppow (transport idmap (path_universe_uncurried
(equiv_path Int Int (ap I add1^))) i) @ add1
= ⟨replace by addVsucc⟩
looppow (transport idmap (path_universe_uncurried equiv_int_succ^−1) i) @ add1
= ⟨rewrite path_universe_V_uncurried⟩
looppow (transport idmap (path_universe_uncurried equiv_int_succ)^ i) @ add1
= ⟨rewrite transport_path_universe_V_uncurried⟩
looppow (equiv_int_succ^−1 i) @ add1
= ⟨cbn⟩
looppow (int_pred i) @ add1
```

This gives us the goal `looppow (int_pred i) @ add1 = looppow i`. At this point, we can apply induction on the argument using `induction i` and simplify the resulting branches. This gives the following:

```
______________________________________(1/3)
add1^ @ add1 = 1
______________________________________(2/3)
looppow match p with
        | one ⇒ zero
        | succ q ⇒ pos q
        end @ add1 = looppow_pos p
______________________________________(3/3)
(looppow_pos p @ add1)^ @ add1 = (looppow_pos p)^
```

The first branch holds immediately through the lemma `concat_Vp`. In the second branch, applying induction on the argument `p` and simplifying the branches using `cbn` results in the following two branches:

```
_____________________________________(1/2)
1 @ add1 = add1
_____________________________________(2/2)
looppow_pos p @ add1 = looppow_pos p @ add1
```

The first branch holds immediately through the lemma `concat_1p`. The second branch holds through `reflexivity`.

For the third branch, applying induction and simplifying the branches gives the following:

```
_____________________________________(1/2)
(add1 @ add1)^ @ add1 = add1^
_____________________________________(2/2)
((looppow_pos p @ add1) @ add1)^ @ add1 = (looppow_pos p @ add1)^
```

In the first branch, we have the following derivation:

```
(add1 @ add1)^ @ add1 = add1^
⇒ ⟨rewrite inv_pp⟩
(add1^ @ add1^) @ add1 = add1^
⇒ ⟨rewrite concat_pp_p⟩
add1^ @ (add1^ @ add1) = add1^
⇒ ⟨rewrite concat_Vp⟩
add1^ @ 1 = add1^
⇒ ⟨rewrite concat_p1⟩
add1^ = add1^
```

where the last term holds by reflexivity.

In the second branch, we have the following derivation:

```
((looppow_pos p @ add1) @ add1)^ @ add1 = (looppow_pos p @ add1)^
⇒ ⟨rewrite looppow_comm_add1)
((add1 @ looppow_pos p) @ add1)^ @ add1 = (looppow_pos p @ add1)^
⇒ ⟨rewrite !inv_pp)
(add1^ @ ((looppow_pos p)^ @ add1^)) @ add1 = add1^ @ (looppow_pos p)^
⇒ ⟨rewrite concat_pp_p)
add1^ @ (((looppow_pos p)^ @ add1^) @ add1) = add1^ @ (looppow_pos p)^
⇒ ⟨refine (ap (fun z ⇒ add1^ @ z) _))
((looppow_pos p)^ @ add1^) @ add1 = (looppow_pos p)^
⇒ ⟨rewrite concat_pp_p)
(looppow_pos p)^ @ (add1^ @ add1) = (looppow_pos p)^
⇒ ⟨rewrite concat_Vp)
(looppow_pos p)^ @ 1 = (looppow_pos p)^
```

where the last term holds through the tactic `apply concat_p1`. With this, we have fully defined `decode`.

6.2 Proofs of Equivalence

Our type `decode_encode` has the following signature:

Type 6.2.1

```
Proposition decode_encode (x : PT1) (p : num = x) : decode (encode p) = p.
```

Once again, the reason we are not specifying one of the ends of the path is so that we can apply path induction using `induction p` and process an arbitrary patch history as argument. After `induction p`, the proof fortunately immediately holds through `reflexivity`.

The type `encode_decode` has the following signature:

Type 6.2.2

```
Proposition encode_decode : forall (x : PT1) (c : I x), encode (decode c) = c.
```

As we have an arbitrary element of the patch theory `x` and another element `I x`, we start the proof with the induction principle of the patch theory. This is done using the tactic `simple refine (PT1_ind (fun x ⇒ forall c : I x, encode (decode c) = c) _ _)` and gives the following two branches:

```
______________________________________(1/2)
(fun x : PT1 ⇒ forall c : I x, encode (decode c) = c) num
______________________________________(2/2)
transport (fun x : PT1 ⇒ forall c : I x, encode (decode c) = c) add1 ?b = ?b
```

Here, the second branch immediately holds through the tactic `apply path_ishprop`. For the first branch, bringing `c` into the assumptions using `intro c` (transforming it into an element of type `I num`) and applying induction gives us the following branches:

```
______________________________________(1/3)
encode (decode zero) = zero
______________________________________(2/3)
encode (decode (pos p)) = pos p
______________________________________(3/3)
encode (decode (neg p)) = neg p
```

The first branch holds through reflexivity. Applying induction on the positive number `p` in the other two branches gives us the following four new branches:

```
______________________________________(1/4)
encode (decode (pos one)) = pos one
______________________________________(2/4)
encode (decode (pos (succ p))) = pos (succ p)
______________________________________(3/4)
encode (decode (neg one)) = neg one
______________________________________(4/4)
encode (decode (neg (succ p))) = neg (succ p)
```

For the first branch we have the following derivation:

```
encode (decode (pos one))
= ⟨unfold encode⟩
transport I (decode (pos one)) zero
= ⟨rewrite transport_idmap_ap⟩
transport idmap (ap I (decode (pos one))) zero
= ⟨rewrite PT1_rec_add1⟩
transport idmap (path_universe_uncurried equiv_int_succ) zero
= ⟨apply transport_path_universe_uncurried⟩
pos one
```

For the second branch we have an inductive hypothesis `IHp : encode (decode (pos p)) = pos p` and the following derivation:

```
encode (decode (pos (succ p)))
= ⟨cbn⟩
encode (looppow_pos p @ add1)
= ⟨unfold encode⟩
transport I (looppow_pos p @ add1) zero
= ⟨rewrite transport_idmap_ap⟩
transport idmap (ap I (looppow_pos p @ add1)) zero
= ⟨rewrite ap_pp⟩
transport idmap (ap I (looppow_pos p) @ ap I add1) zero
= ⟨rewrite transport_pp⟩
transport idmap (ap I add1) (transport idmap (ap I (looppow_pos p)) zero)
= ⟨rewrite PT1_rec_add1⟩
transport idmap (path_universe_uncurried equiv_int_succ)
(transport idmap (ap I (looppow_pos p)) zero)
= ⟨rewrite (transport_path_universe_uncurried equiv_int_succ _)⟩
```

```
equiv_int_succ (transport idmap (ap I (looppow_pos p)) zero)
= ⟨rewrite ← transport_idmap_ap⟩
equiv_int_succ (transport I (looppow_pos p) zero)
= ⟨unfold encode in IHp; rewrite IHp⟩
equiv_int_succ (pos p)
= ⟨cbn⟩
pos (succ p)
```

For the third branch we have the following derivation:

```
encode (decode (neg one))
= ⟨unfold encode⟩
transport I (decode (neg one)) zero
= ⟨rewrite transport_idmap_ap⟩
transport idmap (ap I (decode (neg one))) zero
= ⟨rewrite ap_V⟩
transport idmap (ap I (looppow_pos one))^ zero
= ⟨rewrite PT1_rec_add1⟩
transport idmap (path_universe_uncurried equiv_int_succ)^ zero
= ⟨rewrite transport_path_universe_V_uncurried⟩
neg one
```

For the fourth branch we again have an inductive hypothesis `IHp`, to which this time we apply `cbn` and `unfold encode` in order to obtain the assumption `IHp : transport I (looppow_pos p)^ zero = neg p`. We then have the following derivation:

```
encode (decode (neg (succ p)))
= ⟨cbn⟩
encode (looppow_pos p @ add1)^
= ⟨rewrite ← looppow_comm_add1⟩
encode (add1 @ looppow (pos p))^
= ⟨rewrite inv_pp⟩
encode ((looppow (pos p))^ @ add1^)
= ⟨unfold encode⟩
transport I ((looppow (pos p))^ @ add1^) zero
= ⟨rewrite transport_idmap_ap⟩
transport idmap (ap I ((looppow (pos p))^ @ add1^)) zero
= ⟨rewrite !ap_pp⟩
```

```
transport idmap (ap I (looppow (pos p))^ @ ap I add1^) zero
= ⟨rewrite !transport_pp⟩
transport idmap (ap I add1^) (transport idmap (ap I (looppow (pos p))^) zero)
= ⟨rewrite !ap_V⟩
transport idmap (ap I add1)^ (transport idmap (ap I (looppow (pos p)))^ zero)
= ⟨rewrite PT1_rec_add1⟩
transport idmap (path_universe_uncurried equiv_int_succ)^
(transport idmap (ap I (looppow (pos p)))^ zero)
= ⟨rewrite (transport_path_universe_V_uncurried equiv_int_succ _)⟩
equiv_int_succ^−1 (transport idmap (ap I (looppow (pos p)))^ zero)
= ⟨rewrite ← ap_V⟩
equiv_int_succ^−1 (transport idmap (ap I (looppow (pos p))^) zero)
= ⟨rewrite ← transport_idmap_ap⟩
equiv_int_succ^−1 (transport I (looppow (pos p))^ zero)
= ⟨rewrite IHp⟩
equiv_int_succ^−1 (neg p)
= ⟨cbn⟩
neg (succ p)
```

With this, we have defined `encode_decode`.

Now all that remains are the proofs that `encode` and `decode` form equivalences, or in Coq-HoTT syntax that it holds that `IsEquiv encode` and `IsEquiv decode`. Similar to the proofs of equivalence for the integers, we apply the lemma `isequiv_adjointify` for this. In order to prove the lemmas `encode o decode == idmap` and `decode o encode == idmap` that are thusly created, we successively apply the tactics `cbn`, `intro x0` and either `apply encode_decode` or `apply decode_encode` depending on the lemma. This gives us the following constructions, where we specify the implicit arguments in order to allow the equivalences to be applied for arbitrary patch histories:

Type 6.2.3

```
Global Instance equiv_encode (x : PT1) : @IsEquiv (num = x) (I x) encode.
Proof.
simple refine (isequiv_adjointify _ _ _ _).
- exact decode.
- cbn.
  intro x0.
  apply encode_decode.
```

```
- cbn.
  intro x0.
  apply decode_encode.
Defined.
```

Type 6.2.4

```
Global Instance equiv_decode (x : PT1) : @IsEquiv (I x) (num = x) decode.
Proof.
simple refine (isequiv_adjointify _ _ _ _).
- exact encode.
- cbn.
  intro x0.
  apply decode_encode.
- cbn.
  intro x0.
  apply encode_decode.
Defined.
```

These two types are Global Instances, meaning they are typically not directly applied but instead used as typeclasses. See also Section 2.2.1. In this case, this allows for the tactics `equiv_inj encode` and `equiv_inj decode` to be applied without any hassle.

6.3 The Type `concat_to_add`

We now need to prove that `decode` translates function composition to integer addition. As usual, we start off by creating a lemma, which we call `concat_to_add`.

Type 6.3.1

```
Lemma concat_to_add (n m : I num) : (decode n) @ (decode m) = decode (int_add n m : I num).
```

The first thing to notice here is that every instance of `decode` reduces to `looppow`. As such we create an analogous lemma `concat_to_add_looppow` and close off `concat_to_add` using the tactic `exact concat_to_add_looppow`. The type `concat_to_add_looppow` then has the following signature:

Type 6.3.2

```
Lemma concat_to_add_looppow (n m : I num) : looppow n @ looppow m = looppow (int_add n m).
```

We start off with case distinction using the tactic `induction n as [ |p1|p1], m as [ |p2|p2]; cbn -[looppow]`. This results in the following nine branches:

```
_____________________________________(1/9)
looppow zero @ looppow zero = looppow (int_add zero zero)
_____________________________________(2/9)
looppow zero @ looppow (pos p2) = looppow (int_add zero (pos p2))
_____________________________________(3/9)
looppow zero @ looppow (neg p2) = looppow (int_add zero (neg p2))
_____________________________________(4/9)
looppow (pos p1) @ looppow zero = looppow (int_add (pos p1) zero)
_____________________________________(5/9)
looppow (pos p1) @ looppow (pos p2) = looppow (int_add (pos p1) (pos p2))
_____________________________________(6/9)
looppow (pos p1) @ looppow (neg p2) = looppow (int_add (pos p1) (neg p2))
_____________________________________(7/9)
looppow (neg p1) @ looppow zero = looppow (int_add (neg p1) zero)
_____________________________________(8/9)
looppow (neg p1) @ looppow (pos p2) = looppow (int_add (neg p1) (pos p2))
_____________________________________(9/9)
looppow (neg p1) @ looppow (neg p2) = looppow (int_add (neg p1) (neg p2))
```

Of these branches, only the first holds immediately through reflexivity. Additionally, the second and third hold through the tactic `apply concat_1p` and the fourth and seventh hold through the tactic `apply concat_p1`. For the remaining branches 5, 6, 8 and 9, we have to create new lemma's which we will call respectively `ctpa_pos_pos`, `ctpa_pos_neg`, `ctpa_neg_pos` and `ctpa_neg_neg`.

One thing that branches 5 and 9 have in common is that both of them on some level require that `looppow_pos` translates composition to positive number addition. We create a proof for this using a new lemma `concat_to_pos_add` with the following signature:

Type 6.3.3

```
Lemma concat_to_pos_add : forall (p1 p2 : Pos), looppow_pos p1 @ looppow_pos p2
                                                = looppow_pos (pos_add p1 p2).
```

Once again, we start by induction on the arguments and simplify the branches. This gives us the following:

```
______________________________________(1/4)
add1 @ add1 = add1 @ add1
______________________________________(2/4)
add1 @ (looppow_pos p2 @ add1) = (looppow_pos p2 @ add1) @ add1
______________________________________(3/4)
(looppow_pos p1 @ add1) @ add1 = (looppow_pos p1 @ add1) @ add1
______________________________________(4/4)
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1) =
(looppow_pos (pos_add p1 p2) @ add1) @ add1
```

Of these, the first and the third branches hold immediately through application of `reflexivity`. For the second branch, we have the following derivation:

```
add1 @ (looppow_pos p2 @ add1)
= ⟨rewrite concat_p_pp⟩
(add1 @ looppow_pos p2) @ add1
= ⟨rewrite looppow_pos_comm_add1⟩
(looppow_pos p2 @ add1) @ add1
```

Meanwhile for the fourth branch, we have the following derivation:

```
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1 = )
(looppow_pos (pos_add p1 p2) @ add1) @ add1
⇒ ⟨rewrite concat_p_pp⟩
((looppow_pos p1 @ add1) @ looppow_pos p2) @ add1 =
(looppow_pos (pos_add p1 p2) @ add1) @ add1
⇒ ⟨refine (ap (fun z ⇒ z @ add1) _)⟩
(looppow_pos p1 @ add1) @ looppow_pos p2 = looppow_pos (pos_add p1 p2) @ add1
⇒ ⟨rewrite concat_pp_p⟩
looppow_pos p1 @ (add1 @ looppow_pos p2) = looppow_pos (pos_add p1 p2) @ add1
⇒ ⟨rewrite looppow_pos_comm_add1⟩
looppow_pos p1 @ (looppow_pos p2 @ add1) = looppow_pos (pos_add p1 p2) @ add1
⇒ ⟨rewrite concat_p_pp⟩
(looppow_pos p1 @ looppow_pos p2) @ add1 = looppow_pos (pos_add p1 p2) @ add1
```

Where the last the last term holds through the tactic `exact (ap (fun z ⇒ z @ add1) (IHp1 p2))`.

Similar to the commonality between branches 5 and 9, branches 6 and 8 have in common that they on some level require a proof that the composition of `looppow_pos` with `looppow_pos^` translates to the function `int_pos_sub`. We phrase this property as the following lemma:

Type 6.3.4

```
Lemma concat_to_pos_sub : forall p1 p2 : Pos, looppow_pos p1 @ (looppow_pos p2)^
                                              = looppow (int_pos_sub p1 p2).
```

Again, we start by induction on the arguments and simplify the resulting branches. This gives us the following four branches:

```
______________________________________(1/4)
add1 @ add1^ = 1
______________________________________(2/4)
add1 @ (looppow_pos p2 @ add1)^ = (looppow_pos p2)^
______________________________________(3/4)
(looppow_pos p1 @ add1) @ add1^ = looppow_pos p1
______________________________________(4/4)
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1)^ = looppow (int_pos_sub p1 p2)
```

The first branch holds immediately through the lemma `apply concat_pV`. The third branch holds immediately through the lemma `apply concat_pp_V`. For the second branch, we have the following derivation:

```
add1 @ (looppow_pos p2 @ add1)^
= ⟨rewrite inv_pp⟩
add1 @ (add1^ @ (looppow_pos p2)^)
= ⟨rewrite concat_p_pp⟩
(add1 @ add1^) @ (looppow_pos p2)^
= ⟨rewrite concat_pV⟩
1 @ (looppow_pos p2)^
= ⟨rewrite concat_1p⟩
(looppow_pos p2)^
```

For the fourth branch, we have the following derivation for the left hand side:

```
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1)^
= ⟨rewrite inv_pp⟩
(looppow_pos p1 @ add1) @ (add1^ @ (looppow_pos p2)^)
= ⟨rewrite concat_p_pp⟩
((looppow_pos p1 @ add1) @ add1^) @ (looppow_pos p2)^
= ⟨rewrite concat_pp_V⟩
looppow_pos p1 @ (looppow_pos p2)^
```

At this point we have the goal `looppow_pos p1 @ (looppow_pos p2)^ = looppow (int_pos_sub p1 p2)`, which holds through the inductive hypothesis.

Before being able to prove the four remaining branches, we need one more thing. This is to prove that the composition of a `looppow` term with its inverse is commutative. We call this lemma `looppow_pos_pV_comm` and define it with the following signature:

Type 6.3.5

```
Lemma looppow_pos_pV_comm : forall p1 p2 : Pos, (looppow_pos p2)^ @ (looppow_pos p1)
                                                = (looppow_pos p1) @ (looppow_pos p2)^.
```

Applying induction on the arguments and simplifying the branches gives us the following four branches:

```
______________________________________(1/4)
add1^ @ add1 = add1 @ add1^
______________________________________(2/4)
(looppow_pos p2 @ add1)^ @ add1 = add1 @ (looppow_pos p2 @ add1)^
______________________________________(3/4)
add1^ @ (looppow_pos p1 @ add1) = (looppow_pos p1 @ add1) @ add1^
______________________________________(4/4)
(looppow_pos p2 @ add1)^ @ (looppow_pos p1 @ add1)
= (looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1)^
```

For the first branch we have the following derivation:

```
add1^ @ add1 = add1 @ add1^
⇒ ⟨rewrite concat_pV⟩
add1^ @ add1 = 1
```

Where the last term holds through `concat_Vp`. For the second branch we have the following derivation:

```
(looppow_pos p2 @ add1)^ @ add1 = add1 @ (looppow_pos p2 @ add1)^
⇒ ⟨rewrite looppow_pos_comm_add1⟩
(add1 @ looppow_pos p2)^ @ add1 = add1 @ (looppow_pos p2 @ add1)^
⇒ ⟨rewrite !inv_pp⟩
((looppow_pos p2)^ @ add1^) @ add1 = add1 @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite concat_p_Vp⟩
((looppow_pos p2)^ @ add1^) @ add1 = (looppow_pos p2)^
```

Where the last term holds through `concat_pV_p`. For the third branch we have the following derivation:

```
add1^ @ (looppow_pos p1 @ add1) = (looppow_pos p1 @ add1) @ add1^
= ⟨rewrite looppow_pos_comm_add1⟩
add1^ @ (add1 @ looppow_pos p1) = (looppow_pos p1 @ add1) @ add1^
= ⟨rewrite concat_pp_V⟩
add1^ @ (add1 @ looppow_pos p1) = looppow_pos p1
```

Where the last term holds through `concat_V_pp`. For the fourth branch we have the following derivation:

```
(looppow_pos p2 @ add1)^ @ (looppow_pos p1 @ add1) =
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1)^
⇒ ⟨rewrite looppow_pos_comm_add1⟩
(add1 @ looppow_pos p2)^ @ (looppow_pos p1 @ add1) =
(looppow_pos p1 @ add1) @ (looppow_pos p2 @ add1)^
⇒ ⟨rewrite !inv_pp⟩
((looppow_pos p2)^ @ add1^) @ (looppow_pos p1 @ add1) =
(looppow_pos p1 @ add1) @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite ← looppow_pos_comm_add1⟩
((looppow_pos p2)^ @ add1^) @ (add1 @ looppow_pos p1) =
(add1 @ looppow_pos p1) @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite concat_p_pp⟩
(((looppow_pos p2)^ @ add1^) @ add1) @ looppow_pos p1 =
(add1 @ looppow_pos p1) @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite concat_pV_p⟩
(looppow_pos p2)^ @ looppow_pos p1 =
(add1 @ looppow_pos p1) @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite looppow_pos_comm_add1⟩
(looppow_pos p2)^ @ looppow_pos p1 =
(looppow_pos p1 @ add1) @ (add1^ @ (looppow_pos p2)^)
⇒ ⟨rewrite concat_pp_p⟩
(looppow_pos p2)^ @ looppow_pos p1 =
looppow_pos p1 @ (add1 @ (add1^ @ (looppow_pos p2)^))
⇒ ⟨rewrite concat_p_Vp⟩
(looppow_pos p2)^ @ looppow_pos p1 = looppow_pos p1 @ (looppow_pos p2)^
```

Where the last term holds through the inductive hypothesis.

We can now at last prove `ctpa_pos_pos`, `ctpa_pos_neg`, `ctpa_neg_pos` and `ctpa_neg_neg` (see earlier in Section 6.3). Unfolding `ctpa_pos_pos` using `cbn` gives us the following:

```
forall p1 p2 : Pos, looppow_pos p1 @ looppow_pos p2 = looppow_pos (pos_add p1 p2)
```

Which immediately holds through `concat_to_pos_add`. The same holds for `ctpa_pos_neg` and `concat_to_pos_sub`

Unfolding `ctpa_neg_neg` gives us the following:

```
forall p1 p2 : Pos, (looppow_pos p1)^ @ (looppow_pos p2)^ = (looppow_pos (pos_add p1 p2))^
```

After bringing `p1` and `p2` into the assumptions using `intros p1 p2`, we have the following derivation:

```
(looppow_pos p1)^ @ (looppow_pos p2)^ = (looppow_pos (pos_add p1 p2))^
⇒ ⟨rewrite ← inv_pp⟩
(looppow_pos p2 @ looppow_pos p1)^ = (looppow_pos (pos_add p1 p2))^
⇒ ⟨rewrite pos_add_comm⟩
(looppow_pos p2 @ looppow_pos p1)^ = (looppow_pos (pos_add p2 p1))^
```

Where the last term holds through the tactic `exact (ap inverse (concat_to_pos_add p2 p1))`.

Unfolding `ctpa_neg_pos` gives us the following:

```
forall p1 p2 : Pos, (looppow_pos p1)^ @ looppow_pos p2 = looppow (int_pos_sub p2 p1)
```

After we once again bring the arguments into the assumptions the proof holds through the tactics `rewrite looppow_pos_pV_comm` and `apply concat_to_pos_sub`. With this, we have proven all branches of `concat_to_add_looppow` and by extension `concat_to_add`.

6.4 The Merge Principle

Now that we have made certain that the type `PT1` can indeed be used in in the way described in the paper and defined the necessary scaffolding, it becomes time to define the merge principle for patches. We can define the principle itself as follows:

Type 6.4.1

```
Definition merge :
  (num = num) * (num = num) → (num = num) * (num = num) :=
fun start ⇒ (snd start, fst start).
```

Here, the arguments of the function are two patch histories `f1` and `f2` which are assumed to start from the same node `n`. The function returns two different patch theories `g1` and `g2` which come to the same node `n'` when applied to `f1` respectively `f2`. In the function definition, `g1` is set to `f2` and `g2` to `f1`. Indeed, it holds that applying the actions in `f1` followed by those in `f2` is for the purposes of our patch theory equivalent to applying the actions in `f2` followed by those in `f1`. To prove this, we construct a new assertion `reconcile` with the following signature:

Type 6.4.2

```
Definition reconcile (f1 f2 g1 g2 : num = num) (merger : (merge (f1, f2)) = (g1, g2)) :
                                                  (g1 @ f1) = (g2 @ f2).
```

We start by unfolding the merge principle using the tactics `unfold merge in merger; cbn in merger` to obtain the following proof state:

```
f1, f2, g1, g2 : num = num
merger : (f2, f1) = (g1, g2)
______________________________________(1/1)
g1 @ f1 = g2 @ f2
```

Next, we need to construct a lemma which to destructs an element of the form (_, _) → (_, _) when it is an assumption. We define the lemma `pairs_equal` as follows:

Type 6.4.3

```
Lemma pairs_equal {A B : Type} (x y : A * B) (pxy : x = y) : (fst x = fst y) * (snd x = snd y).
```

Splitting the goal using the tactic `split` gives us the following two branches:

```
A : Type
B : Type
x, y : A * B
pxy : x = y
______________________________________(1/2)
fst x = fst y
______________________________________(2/2)
snd x = snd y
```

The goals in the branches only differ from the assumption `pxy` by the application of `fst` respectively `snd`. Therefore we can close them off using the tactics `exact` (`ap fst pxy`) and `exact` (`ap snd pxy`) respectively. This completes the proof of `pairs_equal`.

Applying our newly constructed lemma and simplifying the assumption using `cbn in merger` gives us the following proof state:

```
f1, f2, g1, g2 : num = num
merger : (f2 = g1) * (f1 = g2)
______________________________________(1/1)
g1 @ f1 = g2 @ f2
```

At this point we can split up the assumption using `destruct merger as (eq1 & eq2)` to obtain the following:

```
f1, f2, g1, g2 : num = num
eq1 : f2 = g1
eq2 : f1 = g2
______________________________________(1/1)
g1 @ f1 = g2 @ f2
```

We prove this goal by translating the terms in the goal to their integer equivalents using `encode`, and then applying that integer addition is commutative. For this we create a new lemma `encode_concat` and give it the following signature:

Type 6.4.4

```
Lemma encode_concat (f g : num = num) : encode (f @ g) = int_add (encode f) (encode g).
```

We now have the following derivation:

```
encode (f @ g) = int_add (encode f) (encode g)
⇒ ⟨apply (equiv_inj decode)⟩
decode (encode (f @ g)) = decode (int_add (encode f) (encode g))
⇒ ⟨rewrite decode_encode⟩
f @ g = decode (int_add (encode f) (encode g))
⇒ ⟨rewrite ← concat_to_add⟩
f @ g = decode (encode f) @ decode (encode g)
⇒ ⟨rewrite !decode_encode⟩
f @ g = f @ g
```

Where the last term holds by reflexivity. We can now prove the reconcile principle as follows:

```
g1 @ f1 = g2 @ f2
⇒ ⟨rewrite eq1⟩
g1 @ f1 = g2 @ g1
⇒ ⟨rewrite eq2⟩
g1 @ g2 = g2 @ g1
⇒ ⟨apply (equiv_inj encode)⟩
encode (g1 @ g2) = encode (g2 @ g1)
⇒ ⟨rewrite !encode_concat⟩
int_add (encode g1) (encode g2) = int_add (encode g2) (encode g1)
```

Where the last term holds through the lemma `int_add_comm`.

6.5 The Symmetry Principle

Now we can move on to the symmetry principle, which asserts that the merge operation does not care about the order of its arguments. In order for our patch theory to be consistent with `darcs` patch theory, this is a property that must hold. We define the symmetry principle as follows:

```
Definition symmetric (f1 f2 g1 g2 : num = num) :
  (merge (f1, f2) = (g1, g2)) → (merge (f2, f1) = (g2, g1)).
```

Same as with `reconcile`, we start by splitting the assumption `merger` using the tactics `unfold merge in merger`, `cbn in merger`, `apply pairs_equal in merger`, `cbn in merger`, and `destruct merger as (eq1 & eq2)`. This leads to the following state:

```
f1 : num = num
f2, g1 : num = num
g2 : num = num
eq1 : f2 = g1
eq2 : f1 = g2
______________________________________(1/1)
merge (f2, f1) = (g2, g1)
```

After applying the assumptions `eq1` and `eq2` using the tactics `rewrite eq1` and `rewrite eq2`, we are then left with a goal which is exactly the definition of `merge`. As such, we can close the proof off using `reflexivity`. Now that the symmetry property has been proven, we have fully defined the merge principle as described in the paper *Homotopical patch theory*. This also completes our formalization of the first patch theory.

Chapter 7

Conclusion

In this paper we provided a full implementation of the first patch theory outlined in the paper *Homotopical Patch Theory*. In order to accomplish this we re-implemented the integers and used the encode-decode principle to obtain the property that they form a set. We also provided implementations of two proposed interpreters and proved several properties regarding them in order to ensure the model of the patch theory behaves as expected.

For our implementation, we deemed it necessary to deviate from the proofs laid out in the paper in order to improve readability. However, given that the types fundamentally did not change we feel confident that our version control system operates in every regard in the same way as the system put forth in the guiding paper. This is further reinforced by the fact that the interpreters also behaved as expected.

As for future potential, our implementation serves to demonstrate that it is indeed possible to implement a version control system in homotopy type theory. This opens up new avenues for implementing and analyzing programs more broadly using the theory, paving the road to potentially further improvements. For instance, researchers capturing some program in the form of a higher inductive type [23], [24] could use all the existing tools discovered for such objects to find deep insights by looking at their fundamental groups [25], [26], or by seeing what tools from isomorphic groups could be carried over [9].

Bibliography

[1] M. Hofmann and T. Streicher, "The groupoid model refutes uniqueness of identity proofs," in *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994, pp. 208–212. DOI: 10.1109/LICS.1994.316071.

[2] V. Voevodsky, *A very short note on the homotopy λ-calculus*, 2006.

[3] S. Awodey and M. A. Warren, "Homotopy theoretic models of identity types," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 146, no. 1, pp. 45–55, 2009. DOI: 10.1017/S0305004108001783.

[4] D. R. Licata and G. Brunerie, "A cubical approach to synthetic homotopy theory," in *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, IEEE Computer Society, 2015, pp. 92–103. DOI: 10.1109/LICS.2015.19. [Online]. Available: https://doi.org/10.1109/LICS.2015.19.

[5] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: A constructive interpretation of the univalence axiom," in *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, T. Uustalu, Ed., ser. LIPIcs, vol. 69, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, 5:1–5:34. DOI: 10.4230/LIPIcs.TYPES.2015.5. [Online]. Available: https://doi.org/10.4230/LIPIcs.TYPES.2015.5.

[6] T. U. F. Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. [Online]. Available: https://homotopytypetheory.org/book/.

[7] E. Rijke, *Introduction to homotopy type theory*, 2022. arXiv: 2212.11082 [math.LO].

[8] T. Altenkirch and L. Scoccola, "The integers as a higher inductive type," in *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, Eds., ACM, 2020, pp. 67–73. DOI: 10.1145/3373718.3394760. [Online]. Available: https://doi.org/10.1145/3373718.3394760.

[9] D. R. Licata and M. Shulman, "Calculating the fundamental group of the circle in homotopy type theory," in *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, IEEE Computer Society, 2013, pp. 223–232. DOI: 10.1109/LICS.2013.28. [Online]. Available: https://doi.org/10.1109/LICS.2013.28.

[10] K. Sojakova, "The equivalence of the torus and the product of two circles in homotopy type theory," *ACM Trans. Comput. Log.*, vol. 17, no. 4, p. 29, 2016. DOI: 10.1145/2992783. [Online]. Available: https://doi.org/10.1145/2992783.

[11] M. Stosberg *et al.*, "Features," 2019. [Online]. Available: http://darcs.net/Features.

[12] G. Sittampalam *et al.*, "Some properties of darcs patch theory," 2005. [Online]. Available: http://urchin.earth.li/darcs/ganesh/darcs-patch-theory/theory/formal.pdf.

[13] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper, "Homotopical patch theory," *J. Funct. Program.*, vol. 26, e18, 2016. DOI: 10.1017/S0956796816000198. [Online]. Available: https://doi.org/10.1017/S0956796816000198.

[14] R. L. Constable, S. F. Allen, M. Bromley, *et al.*, *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986, ISBN: 978-0-13-451832-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=10510.

[15] T. C. D. Team, *The Coq Proof Assistant*, version 8.15, Jan. 2022. DOI: 10.5281/zenodo.5846982. [Online]. Available: https://doi.org/10.5281/zenodo.5846982.

[16] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters, "The hott library: A formalization of homotopy type theory in coq," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Y. Bertot and V. Vafeiadis, Eds., ACM, 2017, pp. 164–172. DOI: 10.1145/3018610.3018615. [Online]. Available: https://doi.org/10.1145/3018610.3018615.

[17] *Coq-hott wiki*, 2023. [Online]. Available: https://github.com/HoTT/Coq-HoTT/wiki.

[18] W. Swierstra and A. Löh, "The semantics of version control," in *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, Eds., ACM, 2014, pp. 43–54. DOI: 10.1145/2661136.2661137. [Online]. Available: https://doi.org/10.1145/2661136.2661137.

[19] S. Mimram and C. D. Giusto, "A categorical theory of patches," in *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013*, D. Kozen and M. W. Mislove, Eds., ser. Electronic Notes in Theoretical Computer Science, vol. 298, Elsevier, 2013, pp. 283–307. DOI: 10.1016/j.entcs.2013.09.018. [Online]. Available: https://doi.org/10.1016/j.entcs.2013.09.018.

[20] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029817.

[21] J. Dagit, "Type-correct changes—a safe approach to version control implementation," 2009. [Online]. Available: https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/47429f27z.

[22] S. Awodey, N. Gambino, and K. Sojakova, "Inductive types in homotopy type theory," in *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, IEEE Computer Society, 2012, pp. 95–104. DOI: 10.1109/LICS.2012.21. [Online]. Available: https://doi.org/10.1109/LICS.2012.21.

[23] H. Basold, H. Geuvers, and N. van der Weide, "Higher inductive types in programming," *J. Univers. Comput. Sci.*, vol. 23, no. 1, pp. 63–88, 2017. [Online]. Available: http://www.jucs.org/jucs%5C_23%5C_1/higher%5C_inductive%5C_types%5C_in.

[24] N. van der Weide, "Constructing higher inductive types as groupoid quotients," in *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, Eds., ACM, 2020, pp. 929–943. DOI: 10.1145/3373718.3394803. [Online]. Available: https://doi.org/10.1145/3373718.3394803.

[25] D. R. Licata and E. Finster, "Eilenberg-maclane spaces in homotopy type theory," in *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, T. A. Henzinger and D. Miller, Eds., ACM, 2014, 66:1–66:9. DOI: 10.1145/2603088.2603153. [Online]. Available: https://doi.org/10.1145/2603088.2603153.

[26] D. R. Licata and G. Brunerie, "π n (S n) in homotopy type theory," in *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, G. Gonthier and M. Norrish, Eds., ser. Lecture Notes in Computer Science, vol. 8307, Springer, 2013, pp. 1–16. DOI: 10.1007/978-3-319-03545-1_1. [Online]. Available: https://doi.org/10.1007/978-3-319-03545-1%5C_1.