Homotopical Patch Theory

Carlo Angiuli *
Carnegie Mellon University
cangiuli@cs.cmu.edu

Ed Morehouse*
Carnegie Mellon University
edmo@cs.cmu.edu

Daniel R. Licata
Wesleyan University
dlicata@wesleyan.edu

Robert Harper*
Carnegie Mellon University
rwh@cs.cmu.edu

Abstract

Homotopy type theory is an extension of Martin-Löf type theory, based on a correspondence with homotopy theory and higher category theory. The propositional equality type becomes proof-relevant, and acts like paths in a space. Higher inductive types are a new class of datatypes which are specified by constructors not only for points but also for paths. In this paper, we show how patch theory in the style of the Darcs version control system can be developed in homotopy type theory. We reformulate patch theory using the tools of homotopy type theory, and clearly separate formal theories of patches from their interpretation in terms of basic revision control mechanisms. A patch theory is presented by a higher inductive type. Models of a patch theory are functions from that type, which, because function are functors, automatically preserve the structure of patches. Several standard tools of homotopy theory come into play, demonstrating the use of these methods in a practical programming context.

1. Introduction

Martin-Löf's intensional type theory (MLTT) is the basis of proof assistants such as Agda [28] and Coq [8]. Homotopy type theory is an extension of MLTT based on a correspondence with homotopy theory and higher category theory [3, 10, 12, 13, 23, 33–35]. In homotopy theory, one studies topological spaces by way of their points, paths (between points), homotopies (paths or continuous deformations between paths), homotopies between homotopies (paths between paths between paths), and so on. In type theory, a space corresponds to a type A. Points of a space correspond to elements $a, b : A$. Paths in a space are modeled by elements of the identity type (propositional equality), which we notate $p : a =_A b$. Homotopies between paths p and q correspond to elements of the iterated identity type $p =_{a=_A b} q$. The rules for the identity type allow one to define the operations on paths that are considered in homotopy theory. These include identity paths $\mathsf{refl} : a = a$ (reflexivity of equality), inverse paths $! p : b = a$ when $p : a = b$ (symmetry of equality), and composition of paths $q \circ p : a = c$ when $p : a = b$ and $q : b = c$ (transitivity of equality), as well as homotopies relating these operations (for example, $\mathsf{refl} \circ p = p$), and homotopies relating these homotopies, etc. This correspondence has suggested several extensions to type theory. One is Voevodsky's *univalence axiom* [16, 34], which describes the path structure of the universe (the type of small types). Another is *higher inductive types* [24, 25, 30], which are a new class of datatypes, specified by constructors not only for points but also for paths. Higher inductive types were originally introduced to permit basic topological spaces such as circles and spheres to be defined in type theory, and have had significant applications in a line of work on using homotopy type theory to give computer-checked proofs in homotopy theory [18, 19, 22, 32].

The computational interpretation of homotopy type theory as a programming language is a subject of active research, though some special cases have been solved, and work in progress is promising [4, 5, 21, 31]. The main lesson of this work is that, in homotopy type theory, proofs of equality have computational content, and can influence how a program runs. This suggests investigating whether there are programming applications of computationally relevant equality proofs. Some preliminary applications have been investigated. For example, Licata and Harper [20] apply ideas related to homotopy type theory to modeling variable binding. Altenkirch [2] shows that containers [1] in homotopy type theory can be used to represent more data structures than in MLTT, such as sets and bags. However, at present, the programming side is less well-developed than the mathematical applications.

In this paper, we present an extended example of applying higher inductive types in programming. The example we consider is *patch theory* [6, 9, 14, 15, 27, 29], as developed for the version control system Darcs [29]. Intuitively, a patch is a syntactic representation of a function that changes a repository. A patch ("delete file f") applies in certain repository contexts (where the file f exists), and results in another repository context (where the file f no longer exists)—so the contexts act as types for patches. Patches are closed under identity (a no-op), composition (sequencing), and inverses (undo). These satisfy certain general laws—composition is associative; inverses compose to the identity. Moreover, there are domain-specific patch laws about the basic basic patches ("the order of edits to independent lines of a file can be swapped"). The *semantics* of a patch explains how to apply it to change a repository.

* This research was sponsored in part by the National Science Foundation under grant number CCF-1116703. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

A *pseudocommutation* operation on patches allows for merging divergent edits to a repository [11]; this is an example of a syntactic transformation on patches. The semantics and syntactic transformations satisfy certain laws, such as the fact that applying a composition of patches is the same as the composition of applying the patches, and that the patches produced by merging two edits leave the two repositories in the same state.

Building on this work, we develop patch theory in the context of homotopy type theory, using paths to model aspects of patch theory. First, we use paths to model the laws that patches and transformations must satisfy. However, we go further than this, and model *patches themselves as paths*, making use of the proof-relevant notion of equality in homotopy type theory. We make an explicit distinction between patch theories[1] and models. A patch theory is presented by a higher inductive type, where the points of the type are repository contexts, the paths in the type are patches, and the paths between paths are patch laws. This presentation of a patch theory consists of only the basic patches ("add / remove files") and laws about them. Identity, inverse, and composition operations are provided by the higher inductive type, and automatically satisfy the desired laws.

Models of a patch theory are represented as functions from the higher inductive type representing it. Because functions in homotopy type theory are always functorial, such models are a *functorial semantics* in the sense of Lawvere [17]. These models depend crucially on the proof-relevance of paths, assigning proofs of equalities a computational meaning as functions acting on repositories. Functoriality implies that a model must respect identity, inverses, and composition (e.g. sending composition of patches to composition of functions) and validate the patch laws. So a patch theory is a formal object, a particular higher inductive type, and the theory is realized by a formal object, a mapping into another type. One syntactic theory of patches can have many different models, e.g. ones that maintain different metadata. Syntactic transformations on patches, such as patch optimization or pseudocommutation can be implemented as functions on paths. Some of these operations can be defined directly in a functorial way, whereas others require developing a derived induction principle for patches.

Our work shows what standard homotopy-theoretic tools mean in a practical programming setting. For example, our first example of a patch theory is actually the circle. Defining the semantics of patch theories uses a programming technique derived from homotopy-theoretic examples. The derived induction principle for patches is analogous to calculations of homotopy groups in homotopy theory. Moreover, our work illustrates some elements of a computational interpretation of homotopy type theory. We hope that this paper will make higher inductive types more accessible to the functional programming community, so that programmers can begin to consider applications of this new class of datatypes.

In Section 2, we provide a brief introduction to homotopy type theory and higher inductive types. In Section 3, we review Darcs patch theory, and describe our approach to representing it in homotopy type theory. In Sections 4 and 5 and 6, we discuss three successively more complex patch languages.

2. Basics of Homotopy Type Theory

We review some basic definitions; see [32] for a thorough introduction.

[1] There is an unfortunate terminological coincidence here: "Patch theory" means "the study of patches," just as "group theory" is the study of groups. "A patch theory" means "a specific language of patches," just as "a theory in first-order logic" is a specific collection of terms and formulae.

2.1 Paths

In homotopy type theory, proofs of equality, or elements of the identity type `x = y`, are used to model a notion of *paths in a space* or *morphisms in a groupoid*. Using the identity type (specified by reflexivity and the J elimination rule), one can define path operations including a constant path `refl` (reflexivity of equality); composition of paths `p ∘ q` (transitivity of equality)[2], and the inverse of a path `! p` (symmetry of equality). Moreover, there are *paths between paths*, or *homotopies*, which are modeled by proofs of equality in identity types. For example, there are homotopies expressing that the path operations satisfy the group(oid) laws:

```
refl ∘ p = p = p ∘ refl
(p ∘ q) ∘ r = p ∘ (q ∘ r)
(! p ∘ p) = refl = (p ∘ ! p)
```

Any simply-typed function `f : A → B` determines a function

```
ap f : x = y → f(x) = f(y)
```

that takes paths `x` $=_{\texttt{A}}$ `y` to paths `f(x)` $=_{\texttt{B}}$ `f(y)`. Logically, this expresses that propositional equality is a congruence; homotopically, it expresses that any function has an **a**ction on **p**aths. `ap f` preserves the path operations, in the sense that there are homotopies

```
ap f (refl(x)) = refl(f x)
ap f (! p) = ! (ap f p)
ap f (p ∘ q) = (ap f p) ∘ (ap f q)
```

For a family of types `B : A → Type` and a dependent function `f : (x : A) → B(x)`, there is

```
apd : (p : x = y) → PathOver B p (f x) (f y)
```

`PathOver B p b1 b2` represents a path in the dependent type `B` between `b1 : B(a1)` and `b2 : B(a2)` that "lies over" `p : a1 = a2`; logically, it is a kind of heterogeneous equality [26] relative to a particular path relating the type of its endpoints. For `apd`, heterogeneous equality is necessary because `f x : B(x)` whereas `f y : B(y)`.

2.2 n-types

A type `A` is a *set*, or 0-type, iff any two paths in `A` are equal—for any two elements `m,n:A`, and any two proofs `p,q : m = n`, there is a homotopy `p = q`. Similarly, a type is a 1-type iff any two paths between paths are equal. A type is a proposition, or (-1)-type, iff any two elements are equal. A type is contractible if it is a proposition and moreover it has an element.

2.3 Univalence

Writing `Type` for a type of (small) types, Voevodsky's univalence axiom states that, for sets `A` and `B`, the paths `A` $=_{\texttt{Type}}$ `B` are given by bijections between `A` and `B`.[3] That is, define `Bijection A B` to be the type of quadruples

```
(f : A → B, g : B → A,
 p : (x : A) → g (f x) = x, q : (y : B) → f (g y) = y)
```

consisting of two functions that are mutually inverse up to paths. Then one consequence of univalence is that there is a function

```
ua : Bijection A B → A = B
```

which says that a bijection between `A` and `B` determines a path between `A` and `B`. The force of this is to stipulate that *all constructions respect bijection*; for example, if `C[X]` is a parametrized type (e.g. `C` could be `List`, `Tree`, `Monoid`, etc.), then given a bijection `b : Bijection A B`, we have

[2] Composition is in function-composition order, `(p:y=z) ∘ (q:x=y)`.

[3] For non-sets, univalence requires a notion of *equivalence* that generalizes bijection. However, here we will only use it for sets.

```
ap C (ua b) : C[A] = C[B]
```

which is a bijection between `C[A]` and `C[B]`. In plain MLTT, one would need to spell out how a bijection lifts to a bijection on lists or monoids; with univalence, this lifting is given by a new generic program in the form of `ap`. This generic program is one of the sources of computational applications of homotopy type theory.

We can define the identity, inverse, and composition of bijections directly (focusing on the underlying functions):

```
reflb : Bijection A A
reflb = ((\ x -> x), (\ x -> x) , ...)

!b : Bijection A B → Bijection B A
!b (f,g,p,q) = (g,f,q,p)

_∘b_ : Bijection B C → Bijection A B → Bijection A C
(f1,g1,p1,q1) ∘b (f2,g2,p2,q2) = (f1 . f2, g2 . g1, ...)
```

Applying path operations to univalence is homotopic to applying the corresponding operations to bijections:

```
ua reflb = refl
! (ua b) = ua (!b b)
ua b1 ∘ ua b2 = ua (b1 ∘b b2)
```

When `p : A = B`, we write `coe p : A → B` for the function, defined by identity type elimination, that "coerces" along the path `p`. `coe` is functorial, in the sense that

```
coe refl x = x
coe (p ∘ q) x = coe p (coe q x)
```

`coe p` is a bijection, with inverse `coe !p`; we write `coe-biject p : Bijection A B` when `p : A = B`. The univalence axiom additionally asserts that there is a computation rule

```
coe (ua (f,g,p,q)) x = f(x)
```

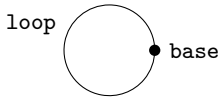
That is, coercing along a path constructed by univalence applies the given bijection. Because `! (ua (f,g,p,q)) = ua (!b (f,g,p,q))`, we also have that

```
coe (! (ua (f,g,p,q))) x = g x
```

Because of these rules, in the presence of univalence, paths can have non-trivial computational content. A bijection `(f,g,p,q)` determines a path `ua(f,g,p,q)`, and `coe`rcing along this path applies `f`. Thus, two different bijections `(f,g,p,q)` and `(f',g',p',q')` determine two paths `ua(f,...)` and `ua(f',...)` that behave differently when `coe`rced along.

2.4 Higher Inductive Types

Ordinary inductive types are specified by *generators*; for example, the natural numbers are have generators zero and successor: `zero : Nat` and `succ : Nat → Nat`. *Higher-dimensional inductive types* (or just higher inductive types) [24, 25, 30] generalize inductive types by allowing generators not only for points (terms), but also for paths. For example, one might draw the circle like this:



This drawing has a single point, and a single non-identity loop from this base point to itself. This translates to a higher inductive type with two generators:

```
base : Circle
loop : base = base
```

`base` is like an ordinary constructor for an inductive type, which takes no argument. `loop` generates a path on the circle, which is an element of the identity type `base` $=_{\texttt{Circle}}$ `base`—think of this as "going around the circle once clockwise". The paths of higher inductive types are constructed from generators, such as `loop`, using the path operations described above. The intuition is that `refl` stands still at the base point, whereas `loop ∘ loop` goes around the circle twice clockwise, and `! loop` goes around the circle once counter-clockwise.

2.4.1 Circle Recursion

The fact that the type of natural numbers is *inductively* generated by zero and successor is encoded in its elimination rule, primitive recursion. Primitive recursion says that to define a function `f : Nat → X`, it suffices to map the generators into `X`, giving `x0 : X` and `x1 : X → X`. Then the function `f` satisfies the equations

```
f zero = x0
f (succ n) = x1(f n)
```

Similarly, the circle is inductively generated by `base` and `loop`, so to define a function from the circle into some other type, it suffices to map these generators into that type, which means giving a point and a loop in that type. That is, to define a function `f : Circle → X`, it suffices to give `b' : X` and `l' : b' =`$_{\texttt{X}}$ `b'`.

For an inductive type, the β-reduction rules state that applying the elimination rule to a generator computes to the corresponding branch. Thus, by analogy, the computation rules for the circle should say that, for a function `f : Circle → X` that is defined by giving `b'` and `l'`,

```
f base = b'
f loop = l'
```

However, the second equation does not quite make sense, because `f` is a function `Circle → X` but `loop` is a *path* on the circle. It is therefore necessary to use `ap` (defined above) to denote `f`'s action on paths:

```
ap f loop = l'
```

This computation rule preserves types because its left-hand side is a proof of `f base = f base`, which by the first computation rule equals `b' = b'`, which is the type of `loop'`.

EXAMPLE 2.1. As a first example, we write a function to "reverse" a path on the circle—to send the path that goes around the circle n times clockwise to the path that goes around the circle n times counter-clockwise, and vice versa. Because a path on the circle is represented by the identity type `base = base`, we seek a function

```
revPath : (base = base) → (base = base)
```

such that, for example, `revPath (loop ∘ loop) = ! loop ∘ ! loop` and `revPath (! loop ∘ ! loop) = loop ∘ loop`. We could define this function by `revpath p = ! p`, but because the goal is to illustrate circle recursion, we instead give an equivalent definition that analyzes `p`.

To define this function using circle recursion, we need to rephrase the problem as constructing a function `Circle → X` for some type `X`. The key idea is to define a function `rev : Circle → Circle` and then to define `revPath` to be `ap rev`. That is, to define a function on the *paths* of the circle, we define a function on the circle itself, whose action on paths is the desired function. In this case, we define

```
rev : Circle → Circle
rev base = base
ap rev loop = ! loop

revPath p = ap rev p
```

One technical issue about higher inductive types is whether the computation rule `ap f loop = l'` is a definitional equality or a

path/propositional equality. Current models and implementations justify only the latter, so we will take it to be a propositional equality. When we illustrate how programs run in this paper, we will do it by giving a sequence of propositional equalities relating a program to a value, so the rule still functions as a "computation" step—as do the rules mentioned above, which state that `ap` behaves homomorphically on paths built from the group operations. For example, one can calculate

```
  revPath (loop ∘ loop)
= ap rev (loop ∘ loop)
= (ap rev loop) ∘ (ap rev loop)
= ! loop ∘ ! loop
```

Just as the recursion principle for the natural numbers can be generalized to an induction principle, the full form of the circle elimination rule is a principle of "circle induction": to define a dependent function `f : (x : Circle) → C(x)`, it suffices to give `b' : C(base)` and `l' : PathOver C loop b' b'`. We refer the reader to [22, 32] for topological intuition.

3. Patch Theory

The developers of the Darcs distributed revision control system [29] have proposed a partially formalized theory of versioned repositories, called *patch theory* [6, 9, 11, 14, 15, 27], which specifies properties of patches under operations such as composing, reverting and merging. Patch theory provides a general framework for describing the behavior of various version control systems. Here, we formulate Darcs patch theory in the context of homotopy type theory, clearly separating its semantic aspects (what repositories and patches *are*) from its purely algebraic properties (how they *behave*). We describe how to present a theory of version control as a higher inductive type whose structure encodes both the generic aspects common to all such theories (as set out in Darcs patch theory) as well as the aspects particular to a given theory, specifying the types of patches available and the specialized laws that they obey. In the following sections, we illustrate this method with a number of examples.

In Darcs patch theory, each patch has well-defined domain and codomain *contexts*, which represent, respectively, the states of the repository on which a patch is applicable, and the states resulting from such an application. For example, a patch that deletes a file is applicable only to states in which the file exists, and results in a state in which it does not. In addition, patches respect certain laws that relate sequences of patches to equivalent sequences of patches – equivalent, in the sense that the two sequences have the same effect on the state of a repository.

One of the properties of patches in Darcs patch theory is that they are all invertible. Applying the inverse of a patch after applying the patch itself ($p \cdot !p$)[4] undoes the effect of the patch, leaving the repository in its original state. This seems very natural. But it is also possible to apply the inverse patch first ($!p \cdot p$), to an appropriate repository state, and this composition should also be equivalent to doing nothing. This seems less natural, and forces us to use some care when defining contexts and patches. The reason that Darcs patch theory requires inverses – as opposed to just retractions – for patches is that doing so is the basis for the Darcs approach to reordering of patches, a critical ingredient in defining the merge of two disparate patches.

However, the requirement that patches be invertible fits very nicely into homotopy type theory, where the path structure of types is undirected. We exploit this coincidence to encode theories of version control as higher inductive types. Contexts are represented as points of a type. Patches are represented as paths between points, with the path operations `refl` and `p · q` and `! p` representing a no-op patch, patch composition, and undo, respectively. Patch laws are represented as 2-dimensional paths between paths. Patch laws are necessary to reason about syntactic transformations on patches, such as an optimizer, which should compute a patch equal to the one it is given, or a merge, which given two divergent edits, should compute two additional patches that reconcile them.

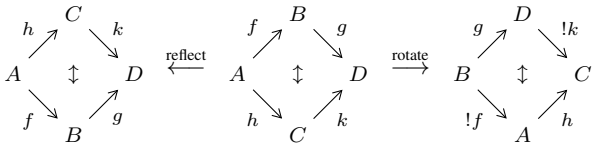
Encoding a theory of patches as a higher inductive type immediately imposes some reasonable laws on patches, namely, the groupoid laws. But a theory of version control is more than an arbitrary groupoid. We would like a version control system to provide operations such as the "cherry picking" of only selected patches from a sequence and the merging of divergent patches. In Darcs patch theory these operations are derived from an operation known as *pseudocommutation*[5] that reorders adjacent patches. Intuitively, a pair of composable patches $f \cdot g$ pseudocommutes with a parallel pair of composable patches $h \cdot k$ if h has the same effect as g, but in the domain context of f, whereas k has the same effect as f but in the codomain context of h. In general there appears to be no canonical way to pseudocommute a composable pair of patches. Instead, we give some criteria (adapted from Darcs) that a choice of a pseudocommutation of a composable pair of patches should satisfy.

We define *pseudocommutation* (pc) to be a function on composable pairs of paths that yields a parallel pair of composable paths such that:

- the two compositions are equal:
  $pc(f, g) = (h, k) \Longrightarrow f \cdot g = h \cdot k$,
- the function is an involution:
  $pc(f, g) = (h, k) \Longrightarrow pc(h, k) = (f, g)$,
- the function respects path inverses:
  $pc(f, g) = (h, k) \Longrightarrow pc(g, !k) = (!f, h)$.

Because of the involution requirement, we adopt the symmetric notation "$(f, g) \leftrightarrow (h, k)$" to mean that $pc(f, g) = (h, k)$.

Each of these properties can be expressed diagrammatically. Composition equality means that the two composites form a commuting square, involution means that the square can be reflected through its start and end points, and respect for path inverses means that the square can be rotated through an edge:



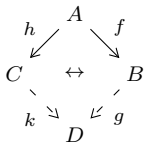
Together, these properties ensure that any isometry of a pseudocommuting square (reversing edge orientations as needed) is also pseudocommuting. Based on their diagrammatic representations we will refer to the latter properties as *reflection* and *rotation*, respectively. The reflection and rotation properties are in fact two-way implications because the composition of two reflections, respectively, four rotations, in the identity function. Thus a reflection is also an unreflection and an unrotation is just the composition of three rotations. Note that it is always possible to define a pseudocommutation function, because the identity is one such.

Using pseudocommutation we can implement an abstract merge operation. Given a span (h, f), we seek a cospan (k, g) such that

[4] In this section we composition in *diagrammatic order* ("$f \cdot g$" for $g \circ f$) to better match the diagrams to follow.

[5] or sometimes just "commutation", though this term is technically inaccurate.

$(f, g) \leftrightarrow (h, k)$:



We can regard this span as a composition either as $!h \cdot f$ or as $!f \cdot h$. In the first case, if we call the result of applying pseudocommutation "$(k, !g)$", then:

$$(!h, f) \leftrightarrow (k, !g) \xrightarrow{\text{rotate}} (f, g) \leftrightarrow (h, k)$$

In the second case, if we call the result of applying pseudocommutation "$(g, !k)$", then:

$$(!f, h) \leftrightarrow (g, !k) \xrightarrow{\text{rotate}} (h, k) \leftrightarrow (f, g) \xrightarrow{\text{reflect}} (f, g) \leftrightarrow (h, k)$$

So we may compute g and k by pseudocommuting either $(!h, f)$ or $(!f, h)$, and the fact that reflection and rotation are invertible ensures that the result is well-defined.

Pseudocommutation gives us a merge operation that is well-defined, symmetric ($merge(h, f) = merge(f, h)$) and reunites the two branches of a span, but this is not enough to guarantee that we get the merges that we might expect. For example, if we take pesudocommutation to be the identity function then the merge of a span (h, f) is the cospan $(!h, !f)$. That is, the induced merge operation simply undoes both changes, reverting the repository to the state from which it diverged (which is, after all, a valid reconciliation of two competing changes, if not the most desirable one in general!)

Next, we present several examples of patch theories as higher inductive types. We show how to implement their semantics, and additionally some examples of patch optimization and merging, to illustrate syntactic transformations.

4. Patches as Paths

First, we define a very simple language of patches, to illustrate the basic technique: we take the repository to be a single integer, and the patches to be adding or subtracting some number n from it. Because all patches apply in any context, we need only a single patch context, which we call `num`. Patches will then be represented as paths `num = num`, which represents the fact that every patch can be applied to context `num` and results in context `num`. Suppose we have a patch `add1` that represents adding 1 to the repository. Then, because paths can be constructed from identity, inverses, and composition, we also have paths `refl`, which represents adding 0, and `add1 ∘ add1`, which represents adding 2, and `! add1`, which represents subtracting 1, and so on. In fact, the patches adding n for any integer n are *generated* by `add1`, because the integers are the free group on one generator. This motivates the following higher inductive definition of patches:

```
R : Type
num  : R
add1 : num = num
```

This is, of course, just a renaming of the circle!

REMARK 4.1. By presenting it using a higher inductive type, the patch theory automatically includes identity, inverses, and composition. Without higher inductive types, one would need syntax constructors for identity, composition, and inverses; e.g. using a datatype as follows:

```
data Patch where
  add1 : Patch
  id   : Patch
  compose : Patch → Patch → Patch
  inv : Patch → Patch
```

Then, to achieve the correct equational theory of patches, one would need to impose the group laws on this type; this could be done using a quotient type [7] to assert that

```
assoc : compose p (compose q r) = compose (compose p q) r
invr : compose p (inv p) = id
invl : compose (inv p) p = id
unitr : compose p id = p
unitl : compose id p = p
```

By representing a patch theory as a higher inductive type, the group operations and laws are provided by the ambient type theory, so the definition need not include these boilerplate constructors.

4.1 Interpreter

Next, we define an interpreter, which explains how to apply a patch to a repository. Because the intended semantics is that the repository is an integer, we would like to interpret the repository context `num` as the type `Int` of integers. Because patches are invertible, we would like to interpret each patch as an element of the type `Bijection Int Int`.

REMARK 4.2. To build intuition, consider writing the interpreter "by hand", for the quotient type `Patch` defined in Remark 4.1, which includes constructors for identity, inverse, and composition. We would first define:

```
interp : Patch → Bijection Int Int
interp add1 = successor
interp id = idb
interp (compose p1 p2) = interp p1 ∘b interp p2
interp (inv p) = !b (interp p1)
```

where `successor : Bijection Int Int` is the bijection given by `(\ x → x + 1, \ x → x - 1, ...)` Then, to show that this definition is well-defined on the quotient of patches by the group laws, we would need to do a proof with 5 cases for the 5 group laws, where in each case we appeal to the inductive hypotheses and the corresponding group law for bijections.

Returning to our higher-inductive representation of patches, we define the interpreter using the recursion principle for `R`, which is of course the same as circle recursion, as discussed in Section 2. We want to interpret each point of `R`, which represent a repository context, as the type of repositories in that context, and each path as a bijection between the corresponding types. In this case, that means we would like to interpret `num` as `Int` and `add1` as the successor bijection. `R`-recursion says that to define a function `f : R → X`, it suffices to find a point `x0 : X` and a loop `p : x0 = x0`. Thus, we can represent the interpretation by a function `R → Type`, because a point of `Type` is a type, and a loop in `Type` is, by univalence, the same as a bijection! This motivates the following definition:

```
I : R → Type
I num = Int
ap I add1 = ua (successor)

interp : (num = num) → Bijection Int Int
interp p = coe-biject (ap I p)
```

Up to propositional equality, this definition satisfies the defining equations of `interp` as defined in Remark 4.2. First, we can calculate that `interp add1 = successor`,

```
  interp add1
= coe-biject (ap I add1)    [definition]
= coe-biject (ua successor) [ap I on add1]
= successor                 [coe on ua successor]
```

using the computation rules for `ap I` on `add` (from higher inductive elimination) and `coe` on `ua b` (from univalence).[6]

Moreover, even though we have not included them as equations, it takes path operations to the corresponding operations on bijections. For example,

```
  interp (p ∘ q)
= coe-biject (ap I (p ∘ q))
= coe-biject (ap I p ∘ ap I q)  [ap on ∘]
= coe-biject (ap I p) ∘b (coe-biject (ap I q))
= interp p ∘b interp q
```

`interp refl = idb` and `interp (! p) = !b (interp b)` are similar. That is, the semantics is functorial.

For example, if we apply[7] a patch `add1∘!add1` to a repository whose contents are 0, we have

```
  (interp (add1 ∘ ! add1)) 0
= ((interp add1) ∘b interp (! add1)) 0
= ((interp add1) ∘b !b interp add1) 0
= (successor ∘b !b successor) 0
= (successor ∘b !b successor) 0
= successor (!b successor 0)
= successor -1
= 0
```

Comparing this definition of `interp` with Remark 4.2, we see that the recursion principle for the higher-inductive representation of patches provides an elegant way to express the semantics of a patch theory, where much of the code in Remark 4.2 is provided "for free". We needed to give only the key case for `add1`, and not the inductive cases for the group operations—the semantics of the basic patches is automatically lifted functorially to the patch operations. Moreover, we did not need to prove that bijections satisfy the group laws—this fact is necessary for the univalence axiom to make sense, so it is effectively part of the metatheory of homotopy type theory, rather than our program. Moreover, the example illustrates that *univalence can be used to extract computational content from a path*, by mapping the path into a path in the universe, which by univalence can be given by a bijection.

Because `R` is the circle, one may wonder about the topological meaning of this interpreter. In fact, the type family `I` defined here is called the *universal cover of the circle*, and is discussed further in [22, 32]. `interp` computes what is called the *winding number* of a path on the circle, which can be thought of as a normal form that counts how many times that path goes around the circle, after "detours" such as `loop ∘ ! loop` have been reduced.

It is also worth noting that, although we were thinking of `num` as an integer and `add1` as successor, there is nothing forcing this interpretation of the syntax: we can give a sound interpretation `I` in any type with a bijection on it. For example,

```
I' : R → Bool
I' num = Bool
I' add1 = ua notb
```

where `notb : Bijection Bool Bool = (not , not , ...)`. That is, we interpret the patches in `Bool` instead of `Int`, and we interpret `add1` as adding 1 modulo 2. This semantics satisfies additional equations that are not reflected in the patch theory, such as

```
ap I' add1 ∘ ap I' add1 = ua (notb ∘b notb) = refl
```

In the next section we show how to augment a patch theory with equations such as these—but doing so would of course rule out the previous semantics in `Int`, because adding 1 to an integer is not self-inverse. The equational theory of `R` is *complete* for the interpretation as `Int`, which in homotopy theory is known as the fact that the fundamental group of the circle is $\mathbb{Z}$ (see [22, 32]).

[6] We also use that fact that two bijections are equal iff their underlying functions are equal, because inverses are unique up to homotopy.

[7] We elide the projection from `Bijection A B` to `A → B`.

4.2 Merge

As discussed in Section 3, merge follows from a pseudocommutation operation. Writing `Patch` for `doc = doc`, and specializing the interface to the setting where we have only one context, we need to implement the following:

```
pcom : Patch × Patch → Patch × Patch
square : (f g h k : Patch) → pcom (f , g) = (h , k)
       → g ∘ f = k ∘ h
rot : (f g h k : Patch) → pcom (f , g) = (h , k)
    → pcom (g, ! k) = (! f, h)
reflect : (f g h k : Patch) → pcom (f , g) = (h , k)
       → pcom (h, k) = (f, g)
```

In this simple setting, any two patches commute, essentially because addition is commutative. Thus, we define

```
pcom(f, g) = (g, f)
```

For `rot`, because we know `h = g` and `k = f`, we need to show that `pcom(g,!f) = (!f,g)`, which is true by definition. For `reflect`, because `h = g` and `k = f`, we need to show that `pcom (g, f) = (f , g)`, which is also true by definition.

For `square`, we need to prove that `g ∘ f = f ∘ g`, for any two loops `num = num` on the circle. It is not immediately obvious how to do this, because homotopy type theory does not provide a direct induction principle for the loops in a type. That is, there is no built-in elimination rule that allows one to, for example, analyze a loop `f` as either `add1`, or the identity, or an inverse, or a composition—because such a case-analysis would additionally need to respect all equations on paths, which differ from type to type. Instead, such induction principles for paths are *proved* for each type from the basic induction principles for the higher inductive types—roughly analogously to how, for the natural numbers, course-of-values induction is derived from mathematical induction. Moreover, proving these induction principles is sometimes a significant mathematical theorem. In homotopy theory, it is called calculating the homotopy groups of a space, and even for spaces as simple as the spheres some homotopy groups are unknown. However, we have developed some techniques for calculating homotopy groups in type theory [18, 19, 22, 32], which can be applied here.

In fact, for this particular example, the calculation has already be done: we know that the fundamental group of the circle is $\mathbb{Z}$. Specifically, we know that the type `num = num` of loops at `num`, which we use to represent patches, is in bijection with `Int`. That is, the integers give normal forms ("add x, for $x \in \mathbb{Z}$") for patches in the above patch theory. This is proved by giving functions back and forth that compose to the identity. The function `num=num → Int` is exactly `λ p → interp p 0`, for `interp p` as defined above. The function `repeat : Int → num=num` is defined by induction on `x`, such that

```
repeat 0 = refl
repeat (+ n) = add1 ∘ add1 ∘ ... ∘ add1 (n times)
repeat (- n) = !add1 ∘ !add1 ∘ ... ∘ !add1 (n times)
```

The proof that these two functions are mutually inverse is described in [22, 32]. Moreover, they define a group homomorphism, which means that `repeat (x + y) = repeat x ∘ repeat y`.

The bijection between `num=num` and `Int` induces a derived induction principle, which says that to prove `P(p)` for all paths `p:num=num`, it suffices to prove `P(repeat n)` for all integers `n`—any patch can be viewed as `repeat n` for some `n`. Applying this (twice) to the goal `f ∘ g = g ∘ f`, it suffices to show

```
repeat x ∘ repeat y = repeat y ∘ repeat x
```

This is proved as follows:

```
  repeat x ∘ repeat y
= repeat (x + y) [group homomorphism]
= repeat (y + x) [commutativity of addition]
= repeat y ∘ repeat x
```

Thus, for this language of patches, the correctness of pseudo-commutation follows from the fact that the fundamental group of the circle is $\mathbb{Z}$—our first example of a software correctness proof being a corollary of a theorem in homotopy theory!

5. Patches with Laws

In this section, we consider a slightly more complex patch theory, to illustrate how patch laws are handled. In the intended semantics of this theory, the repository consists of one document with a fixed number `n` of lines, and there is one basic patch, which modifies the string at a particular line. To fit such a basic into a framework of bijections, we take the patch `s1 ↔ s2 @ i` to mean "permute `s1` and `s2` at position `i`". That is, applying this patch replaces line `i` with `s2` if it is `s1`, or with `s1` if it is `s2`, or leaves it unchanged otherwise. We impose some equational laws on this patch—e.g., edits at independent lines commute.

5.1 Definition of Patches

This patch theory is represented by the following higher inductive type:

```
R : Type
doc : R
_↔_@_ : (s1 s2 : String) (i : Fin n) → (doc = doc)
indep : (i ≠ j) →
          (s ↔ t @ i) ∘ (u ↔ v @ j)
        = (u ↔ v @ i) ∘ (s ↔ t @ j)
noop : s ↔ s @ i = refl
```

`doc` should be thought of as a document with `n` lines (for some `n` fixed throughout this section). The path constructor `s1 ↔ s2 @ i` represents the basic patch, swapping `s1` and `s2` at position i. For this language there are some non-trivial patch laws, which are represented by giving generators for *paths between paths*; we show two as an example. The equation `noop` states that swapping `s` with `s` is the identity for all `s`;[8] this is useful for justifying a simple optimizer, which optimizes away the two string comparisons that executing `s ↔ s @ i` would require. The equation `indep` states that edits to independent lines commute; this is useful for defining `merge` via commutation.

Because `R` is our first example of a type with both paths and paths between paths, we go over its recursion and induction principles in detail. To define a function `f : R → X`, it suffices to give

```
doc'   : X
swap'  : (s1 s2 : String) (i : Fin n) → doc' = doc'
indep' : (s t u v : String) (i j : Fin n) → i ≠ j
       → swap' s t i ∘ swap' u v j
       = swap' u v j ∘ swap' s t i
noop' : (s : String) (i : Fin n)
       → swap' s s i = refl
```

and then

```
f(doc) = doc'
ap f (s ↔ t @ i) = swap' s t i
ap (ap f) (indep s t u v i j neq) ≈
   indep' s t u v i j neq
ap (ap f) (noop s i) ≈ noop' s i
```

[8] We adopt a convention that the free variables appearing in types of these constructors are universally quantified, and treated as implicit arguments.

The final two computation rules are "approximate" because they require some massaging by propositional equality to type check. For example, `ap (ap f) (noop s i)` has type `ap f (s ↔ s @ i) = ap f refl`, but `noop'` has type `swap' s s i = refl`. While `ap f refl` is definitionally equal to `refl`, `ap f (s ↔ s @ i)` is only propositionally equal to `swap' s s i`, because the prior computation rule is only propositional. However, we will not actually need these two computation rules in what follows, so we elide the details. We will use clausal function notation for maps out of `R`, but keep in mind that the types of the right-hand sides of the equations are those of `doc'` and `swap'` and `indep'` and `noop'` above, which (in the latter two cases) are only propositionally equal to the types of the left-hand sides.

The induction principle for `R` states that to define a function `f : (x : R) → C(x)`, it suffices to give

- `c' : C(doc)`
- `s' : PathOver C (s1 ↔ s2 @ i) c' c'`
- A 2-dimensional path over a path as the image of `indep`.
- A 2-dimensional path over a path as the image of `noop`.

We elide the details of the final two clauses, which are not needed below.

5.2 Interpreter

Because patches are represented by the type `doc = doc`, the interpreter for patches is a function

```
interp : (doc = doc)
       → Bijection (Vec String n) (Vec String n)
```

As above, we generalize this to an interpretation of the whole path language `R`, and define a function `I : R → Type` such that

```
interp p = coe-biject (ap I p)
```

To interpret the basic patch `s1 ↔ s2 @ i`, we need a corresponding bijection that permutes two strings at a position in a vector, represented by the type `Vec String n` (length-n vectors of strings):

```
permute : (String × String) → String → String
permute (s1,s2) s | String.equals (s1,s) = s2
permute (s1,s2) s | String.equals (s2,s) = s1
permute (s1,s2) s | _ = s

applyat : (A → A) → Fin n → Vec A n → Vec A n
applyat f i <x1,...xn> = <x1,...,f xi,...,xn>

swapat : (String × String) → Fin n
       → Bijection (Vec A n) (Vec A n)
swapat (s1,s2) i = (applyat (permute (s1,s2)) i, ...)
```

The interpretation `I` is defined as follows:

```
I : R → Type
I doc = Vec String n
ap I (s1 ↔ s2 @ i) = ua (swapat (s1,s2) i)
ap (ap I) (indep i≠j) =
  ?0 : ua(swapat(s,t) i) ∘ ua(swapat(u,v) j)
     = ua(swapat(u,v) j) ∘ ua(swapat(s,t) i)
ap (ap I) noop = ?1 : ua(swapat(s,s) i) = refl
```

We interpret `doc` as `Vec String n`. The image of `s1 ↔ s2 @ i` must be a path in `Type` between `I(doc)` and `I(doc)`—i.e. between `Vec String n` and itself. For this, we choose the bijection `swapat (s1,s2) i`, packed up as a path in the universe using the univalence axiom.

The image of `indep` and `noop` are the goals `?0` and `?1`, with the types written out above—which say that we need to validate the patch laws for the interpretation. These goals can be solved

by equational properties of bijections, combined with the rules about the interaction of univalence with identity and composition described in Section 2. For example, `?1` is solved by observing that `swapat(s,s) i` is the the identity bijection, and then using the fact that `ua idb = refl`. `?0` is solved by turning both sides into a composition of bijections using the fact that `ua b1 ∘ ua b2 = ua (b1 ∘b b2)` , and then proving the corresponding fact about `swapat`:

```
swapat-independent :
  (i ≠ j) → (swapat (s,t) i) ∘b (swapat (u,v) j)
          = (swapat (u,v)  i) ∘b (swapat (s,t) j)
```

As above, we do not need to give cases for the group operations or prove the group laws—these come for free, from functoriality.

5.3 Optimizer

To illustrate using the patch laws, we write a simple optimizer

```
optimize : (p : doc = doc) → Σ (q : doc = doc). p = q
```

The type of `optimize` says that it takes a patch `p` and produces a patch `q` that behaves the same, according to the patch laws, as `p`. The goal is to optimize `s ↔ s @ i` to `refl`, saving ourselves two unnecessary string comparisons when the patch is applied. The optimizer requires analyzing the syntax of patches.

We show two definitions of `optimize`, to illustrate some different aspects of programming in homotopy type theory.

Program then prove. In this definition, we first write a function `optimize1 : doc=doc → doc=doc`, and then prove that this function returns a path that is equal, according to the patch laws, to its input. The idea is to apply the following function `opt0` to each patch `s1 ↔ s2 @ i`:

```
opt0 : String → String → Fin n → doc=doc
opt0 s1 s2 i = if String.equals s1 s2
               then refl
               else (s1 ↔ s2 @ i)
```

To define `optimize1`, we generalize the problem to defining a function `opt1` that acts on all of `R`, and then derive `optimize1` as its action on paths (the same technique as reversing the circle in Section 2.1). This is defined as follows:

```
opt1 : R → R
opt1 doc = doc
ap opt1 (s1 ↔ s2 @ i) = opt0 s1 s2 i
ap (ap opt1) noop =
  ?0 : opt0 s s i = refl
ap (ap opt1) (indep i≠j) =
  ?1 :   opt0 s1 s2 i ∘ opt0 s3 s4 j
       = opt0 s3 s4 j ∘ opt0 s1 s2 i
```

We map `doc` to `doc`, and apply `opt0` to `s1 ↔ s2 @ i`. However, to complete the definition, we must show that the optimization respects the patch laws, via the goals `?0` and `?1` whose types are given above. The goal `?0` is true because `String.equals s s` will be true, so, after case-analysis, `refl` proves that `opt1 s s i = refl`. The goal `?1` requires case-analyzing both `String.equals s1 s2` and `String.equals s3 s4`. If both are true, the goal reduces to `refl ∘ refl = refl ∘ refl`, which is true by `refl`. If the former but not the latter is true, the goal reduces to `refl ∘ s3 ↔ s4 @ j = s3 ↔ s4 @ j ∘ refl`, which is true by unit laws. The third case is symmetric. Finally, if neither are true, then the goal holds by `indep`.

Next, we prove this optimization correct. This is an example of `R`-induction:

```
opt1-correct : (x : R) → x = opt1 x
opt1-correct doc = refl
apd opt1-correct (s1 ↔ s2 @ i) =
  ?0 : PathOver (x. x = opt1 x) (s1 ↔ s2 @ i) refl refl
apd (apd opt1-correct) noop = ?
apd (apd opt1-correct) (indep i≠j) = ?
```

In the case for `doc`, we need to give a path `doc = opt1 doc`, but `opt1 doc` is `doc`, so we give `refl`. In the case for `s1 ↔ s2 @ i`, the induction principle requires an element of the type listed above. It turns out that, by rules for `PathOver`, this type is equivalent to

```
s1 ↔ s2 @ i = opt0 s1 s2 i
```

So this is where we prove that `opt0` preserves the meaning of a patch. This requires two cases, one where `s1` is equal to `s2`, in which case we use `noop`, and one where it is not, in which case we use `refl`.

The remaining two cases require proving that *this proof of correctness of* ***`opt`*** respects the patch laws. In each case, the goal asks us to prove the equality of two proofs of equality of patches. That is, the goal has the form

$$f_1 =_{p=_{\texttt{doc}=\texttt{doc}} q} f_2$$

where p and q are two patches, and f_1 and f_2 are two proofs that these two patches are equal—which homotopically can be thought of as paths-between-paths, or, in more geometrically evocative terminology, as *faces* between *edges*.

One might think that such a goal would be trivial, because f_1 and f_2 are representing proofs that two patches are equal according to the patch laws, and we think of patch equality as a proof-irrelevant relation. But for the definition we have given above, there is nothing that actually forces any two such faces to be identified. For example, we can compose `indep i≠j ∘ indep j≠i`, a proof that `(s ↔ t @ i) ∘ (u ↔ v @ j)` is equal to itself, but there is no reason that this proof, which swaps twice, is necessarily the identity. Thus, although we have not considered any applications of this so far, we could potentially consider proof-relevant identifications between patches—proof-relevant patch laws. If we wished to do so, then these goals would need to be proved.

However, if we do not wish to consider proof-relevant patch equations, we *can* make these goals trivial by a technique called *truncation* [32, Chapter 7]. In this case, we can declare `R` to be a 1-type, which adds a path between any two faces. Then, these goals would be trivial. The price for doing this is that functions defined by `R`-recursion/induction are only permitted when the result is also a 1-type. Fortunately, we can still define `opt1` with this restriction (because `R` is a 1-type), as well as `opt1-correct` (because paths in a 1-type are a 0-type, and therefore a 1-type) and the function `I` used for `interp` (because it interprets the point of `R` as a set, and the collection of all sets is a 1-type). Thus, truncating `R` would be an appropriate and helpful modification in this case.

Program and prove. An alternative, which requires neither truncation nor proving any equations between faces, is to simultaneously implement the optimizer, and prove that it returns a patch equal to its input. To define

```
optimize : (p : doc = doc) → Σ (q : doc = doc). p = q
```

we need to define a function on all of `R`, and derive `optimize` via its action on paths. However, `optimize` is dependently typed, and `ap f` for a simply-typed function `f` never has such a dependent type. Thus, we define a dependently typed function and use the dependent form of `ap`, `apd`. Specifically, we define

```
opt : (x : R) → Σ (y : R). y = x
```

This type has the same shape as the type of `optimize` above, except it is at the level of the *points* of `R` rather than the paths. Its action on paths has the following type:

```
apd opt (p : doc = doc) :
  PathOver (x. Σy:R. y = x) p (opt doc) (opt doc)
```

When the family `B` is known, the type `PathOver B p b1 b2` can be "reduced" (via propositional equalities) to another type. In the case where `B` is `x.Σ (y:R.y = x)`, as above, the rules for path-over-a-path in Σ-types, constant families, and path types, yield an identification `e` as follows:[9]

```
e :   PathOver (x. Σy:R. y = x) p (doc,refl) (doc,refl)
    = Σ (q : doc = doc). p = q
```

Thus, if we define `opt` such that

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

then

```
apd opt (p : doc = doc) :
  PathOver (x. Σy:R. y = x) p (doc , refl) (doc , refl)
```

and we can define `optimize` by composing this with `e`:

```
optimize : (p : doc = doc) → Σ (q : doc = doc). p = q
optimize p = coe (! e) (apd opt p)
```

This reduces the problem to defining `opt`, which we do as follows:

```
opt doc = (doc, refl)
opt (s1 ↔ s2 @ i) = coe e
                    (if String.equals s1 s2
                     then (refl , noop)
                     else (s1 ↔ s2 @ i , refl))
ap (ap opt) _ = <contractibility>
```

We set `opt doc = (doc , refl)`, as motivated above. For the second clause, we need a

```
PathOver (x. Σy:R. y = x) p (doc , refl) (doc , refl)
```

By `e`, it suffices to give a

```
Σ (q : doc = doc). (s1 ↔ s2 @ i) = q
```

Thus, this is where we put the key step that we wanted to make, which is optimizing `s1 ↔ s2 @ i` to `refl` when the strings are equal, and leaving the patch unchanged otherwise—and pairing each with a proof that it is equal to `s1 ↔ s2 @ i`.

For each of the `noop` and `indep` cases, we need to give a face between two specific paths between two specific points in the type `Σ y:R. y = x` (for some `x`). However, the type `Σ y:R.x = y` is in fact *contractible*—it is equivalent to `unit`. Intuitively, any pair `(y,p)` can be continuously deformed to `(x,refl)` by sliding `y` along `p`; see [32, Lemma 3.11.8]. The path space of any contractible type is a a proposition, so any two paths in it are connected by a face. Thus, because we formulated the problem as mapping into a contractible type, we can easily discharge the remaining goals.

This definition of `opt`, consisting of only the three cases given above, is much shorter than our previous attempt. Moreover, for comparison, suppose we instead wrote this optimizer for a datatype of patches that included identity, inverses, and composition as constructors (analogous to the one in Remark 4.1). Then, in addition to giving the "interesting" case for optimizing `s1 ↔ s2 @ i`, we would need to give inductive cases describing how the optimizer acts on identity, inverses, and composition. Here, because the optimizer can be defined as a group homomorphism, we need to give only the "interesting" case; the inductive cases are handled by the framework.

Singleton Types and Computation Because the type `Σ(y:A).x = y` is contractible, we can think of it as a *singleton type*, written `S(x)`. It consists of "everything in `A` that is equal to `x`," or, more precisely, a point in `A` with a path to `x`. One may well wonder what is the point of writing a function into a contractible type? Using the singleton notation we have

```
optimize : (p : doc = doc) → S(p).
```

Because `S(p)` contractible, and hence equivalent to `unit`, isn't this just a triviality? The answer is "no" because even if two elements of a type are connected by a path (and hence cannot be distinguished by any other operation of type theory), the type nevertheless has meaningful computational content in that we may observe its output when it is run and thereby make distinctions that are obscured within the theory. Thus, even though the `optimize` function that we wrote above is equal (i.e., homotopic) to the function that simply returns `p` itself—or, indeed, any other function with that type—we expect, based on work on the computational interpretation of homotopy type theory, that it will in fact compute appropriately—e.g. `optimize (s ↔ s @ i)` will in fact return `refl` because of the way it is programmed.

6. Patches with Types

In the previous sections we have only considered patches of the form `s1 ↔ s2 @ i`, which naturally form total bijections on the type of `n`-line documents. In Section 5, we exploited this fact to model these patches as paths in a higher inductive type, using univalence to map them to bijections on `Vec String n`.

Now we will consider more realistic patches—inserting a string `s` as the `l`th line[10] in a file (`ADD s@l`), and removing the `l`th line of a file (`RM l`).[11] For example, the only patch applicable to an empty file is `ADD s@0`; to the resulting file we may apply one of `ADD s'@0`, `ADD s'@1`, or `RM 0`, which respectively add `s'` before or after `s`, or delete `s`.

These new patches significantly complicate our definition of the patch theory `R` in a number of ways, most obviously because each patch only applies to files of at least a certain length: unlike in our previous patch languages, not all patches are composable. A first cut might be to classify repositories by the number of lines in their file—that is, index the points of `R` by `Nat`, and say that addition of a line is a path `doc n = doc n+1`, and deletion is a path `doc n+1 = doc n`.

This approach fails because addition and deletion aren't bijections between `n` and `n+1`-line files. For example, the function which deletes the first line of a file is not in `Bijection (Vec String n+1) (Vec String n)`, because it sends two files differing only in their first lines to the same file.

Univalence dictates that all the paths in `R` must be interpreted as bijections, so all the points of `R` must be interpreted as isomorphic sets. Because one of the points in `R` ought to represent the unique empty file, all its points must uniquely identify files—only one-element sets are isomorphic to one-element sets.

A solution, therefore, is to index contexts by file contents, i.e., `doc n file : R` where `file : Vec String n`, and `ADD s@l : doc n file = doc n+1 file'`, where `file'` is the result of adding `s` at line `l` in `file`. We reject this approach because it causes

[9] This is because a path over a path in a Σ-type is a path-over-a-path in each component (the second over the first), because a path-over-a-path in a constant family `x.R` is just a path in `R`, and a path-over-a-path in the identity type is a square in the underlying type—specifically, `PathOver (x,y. y = x) (p,q) (refl,refl)` is a square with left/top/right/bottom `p`/`refl`/`q`/`refl`, which is the same as a path between `p` and `q` (this is what motivates the choice of `(doc,refl)` and `(doc,refl)` as the endpoints of the path-over-a-path)

[10] We start our numbering at 0, so the positions in an `n`-line file coincide with `Fin n+1`.

[11] Although this patch language may still seem unreasonably simplistic, the approach taken in this section can scale to many features of real-world version control, in particular multi-file repositories.

the codomains of patches to depend on the concrete implementation of those patches, linking patches' specifications to their intended implementations.

We will instead index contexts by *patch histories*, i.e., sequences of composable patches starting at the empty file. With respect to any particular implementation of patches, histories uniquely identify files, so we still sidestep the issue with bijections described above. As an added benefit, histories also reify sequences of patches in a way which facilitates certain operations on repositories, such as skipping forward or backward in time.

6.1 Definition of Patches

Let `History n` be the type of patch histories (sequences of patches) resulting in `n`-line files. We will define `History n` as a quotient higher inductive type to equate sequences of patches which result in the same changes to a file. For example, two additions in sequence can be commuted if the line numbers are shifted.

```
History : Nat → Type

[] : History zero
ADD_@_::_ : {n : Nat} (s : String) (l : Fin n+1) →
            History n → History n+1
RM_::_    : {n : Nat} (l : Fin n+1) →
            History n+1 → History n

ADD-ADD-< : {n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
  (s1 s2 : String) (h : History n) → l1 < l2 →
  (ADD s2 @ l2 :: ADD s1 @ l1 :: h)
    = (ADD s1 @ l1 :: ADD s2 @ (l2-1) :: h)

ADD-ADD-≥ : {n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
  (s1 s2 : String) (h : History n) → l1 ≥ l2 →
  (ADD s2 @ l2 :: ADD s1 @ l1 :: h)
    = (ADD s1 @ l1+1) :: ADD s2 @ l2 :: h)
```

(For the sake of clarity we have omitted some coercions between different `Fin` types.) To simplify the code in the remainder of this section, we have omitted the paths commuting `ADD-RM`, `RM-ADD`, and `RM-RM`, which can be defined in exactly the same way.

Here, histories serve as the "types" of patches. If we think of a patch as the formal representation of a change to a repository, then the domain of the patch is a history corresponding to a repository to which that change is applicable, and the codomain is the domain history extended by the patch which was just applied.

```
R : Type

doc : {n : Nat} → History n → R

addP : {n : Nat} (s : String) (l : Fin n+1)
       (h : History n) → doc h = doc (ADD s @ l :: h)
rmP  : {n : Nat} (l : Fin n+1)
       (h : History n+1) → doc h = doc (RM l :: h)
```

Next, we would like to insert faces equating commuting sequences of patches, but our definition of histories means that no differing sequences of paths will ever be parallel! For example, when `l1 < l2`, the two paths

```
addP s2 l2 ∘ addP s1 l1
  : h = ADD s2@l2 :: ADD s1@l1 :: h
addP s1 l1 ∘ addP s2 (l2-1)
  : h = ADD s1@l1 :: ADD s2@(l2-1) :: h
```

ought to be "equal" as patches, but it does not even make type sense to state this equation. We rely on the fact that histories are quotiented by the same commutation laws—that is, we already equated those exact elements of `History n` with the path `ADD-ADD-<`. Therefore, we can stipulate that the above two paths are equal *over* the `ADD-ADD-<` equation from `History n`, with respect to the type family `x.h = x`. Thus the faces of `R` are defined as follows:

```
addP-addP-< : {n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
  (s1 s2 : String) (h : History n) → l1 < l2 →
  PathOver (x.doc h = doc x) ADD-ADD-<
    (addP s2 l2 ∘ addP s1 l1)
    (addP s1 l1 ∘ addP s2 (l2-1))

addP-addP-≥ : {n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
  (s1 s2 : String) (h : History n) → l1 ≥ l2 →
  PathOver (x.doc h = doc x) ADD-ADD-≥
    (addP s2 l2 ∘ addP s1 l1)
    (addP s1 (l1+1) ∘ addP s2 l2)
```

6.2 Interpreter

Assume we have functions `add` and `rm` which implement our patches on concrete vectors of `String`'s.

```
add : {n : Nat} (s : String) (l : Fin n+1)
      → Vec String n → Vec String n+1
rm  : {n : Nat} (l : Fin n+1)
      → Vec String n+1 → Vec String n
```

As before, we want to define a function `I : R → Type` which interprets points of `R` (histories) as types, and paths of `R` (patches) as bijections between those types. Then we can define `interp p = coe-biject (ap I p)` and obtain

```
interp : {n1 n2 : Nat} {h1 : History n1} {h2 : History n2}
         → (doc h1 = doc h2) → Bijection ? ?
```

with the idea that `interp (addP s l h)` should in some sense be `add s l`, and `interp (rmP l h)` should be `rm l`.

The type of `interp` has gotten more complex than before, because patches now have many different domains and codomains, instead of a single `doc` object. As a result, we must decide on the interpretation of each `doc h` into the type universe—between which types does each patch induce a bijection?

As we discussed at the beginning of this section, we cannot simply interpret `doc h` as `Vec String n`, because adding and removing lines are not bijections on these types. Instead, we will essentially interpret `doc h` as the exact file which arises from applying the patches in `h`. That is, we will record in its type exactly how the file came into existence, rather than simply regarding it as a plain text file.

We can specialize any function `f : A → B` to a function between singleton types, as follows:

```
to-singleton : {M : A} (f : A → B) → S(M) → S(f M)
to-singleton f (x,p) = (f x, ap f p)
```

Because singleton types are contractible (contain exactly one point, and have trivial higher structure), every function between singleton types is automatically a bijection. Call this fact `single-biject`. Then we can define the interpretation

```
I : R → Type

I (doc h) = S(replay h)
ap I (addP s l h) =
  ua (single-biject (to-singleton (add s l)))
ap I (rmP l h) = ua (single-biject (to-singleton (rm l)))
apd' (ap I) (addP-addP-< l1 l2 s1 s2 h p) = ?0
apd' (ap I) (addP-addP-≥ l1 l2 s1 s2 h p) = ?1
```

where `apd'` is a function which gives the action of a function on a `PathOver`, `replay` is a function which steps through a history to compute the file specified by that history, and `?0` and `?1` are proofs that the implementations of patches satisfy the commutation laws.

```
replay : {n : Nat} → History n → Vec String n
```

```
replay [] = []
replay (ADD s @ l :: h) = add s l (replay h)
replay (RM l :: h) = rm l (replay h)

ap replay (ADD-ADD-< l1 l2 s1 s2 h pf) =
  ?0 : add s2 l2 (add s1 l1 (replay h))
       = add s1 l1 (add s2 (l2-1) (replay h))
ap replay (ADD-ADD-≥ l1 l2 s1 s2 h pf) =
  ?1 : add s2 l2 (add s1 l1 (replay h))
       = add s1 l1+1 (add s2 l2 (replay h))
```

Because histories are quotiented by the commutation laws, we must prove that `replay` sends equal histories to equal files, which amounts to showing that `add` satisfies the same laws as `ADD`.

The implementation of `replay` is needed during typechecking of the definition of `ap I (addP s l h)`, which must be in `Bijection S(replay h) S(replay (ADD s @ l :: h))`. By unrolling the definition of `replay`, the latter type is `S(add s l (replay h))`.

6.3 Logs

The interpreter above suggests that one may implement a version control system in homotopy type theory by storing sequences of patches as paths, and repositories as vectors of strings. A repository can be updated by running `interp` on a new patch. Note that, although the types of the paths include histories which redundantly encode the patch data, these types are only needed to compute the singleton type of the file data, which is not needed at runtime; the file data itself is computed only from the patches themselves. Thus, it would be sensible to discard the histories at runtime, through some erasure mechanism.

Another feature we might like to implement is the ability to print out an explicit representation, or *log*, of all the patches that have been applied to the repository. Logs can't be generated directly from the changes induced by patches on the repository, because we cannot inspect the intensions of functions `S(file) → S(file')`.

Instead, just as we computed changes on repositories by interpreting points of `R` as singleton files, we can compute the changes induced on *histories* through an alternate interpretation of points of `R` as singleton histories:

```
I' : R → Type

I' (doc h) = S(h)
ap I' (addP s l h) =
  ua (single-bject (to-singleton (\ h → ADD s@l :: h))
ap I' (rmP l h) =
  ua (single-bject (to-singleton (\ h → RM l :: h)))
apd' (ap I') (addP-addP-< l1 l2 s1 s2 h p) =
  ADD-ADD-< l1 l2 s1 s2 h p
apd' (ap I') (addP-addP-≥ l1 l2 s1 s2 h p) =
  ADD-ADD-≥ l1 l2 s1 s2 h p
```

Then `interp' p = coe (ap I' p)` takes a patch `p : doc h = doc h'` to a function `S(h) → S(h')` which updates the repository history `h` to the history `h'` which results from applying the patch `p`. As with `interp`, this function computes updates to the repository representation without relying on the endpoints of patches given in their types—this shows that we could recover a history from a patch (and an initial history), if we were to erase histories at run-time.

`I'` is a good example of the benefit of functorial semantics—both `I` and `I'` are models of the patch theory `R`, and the natural functoriality of functions in homotopy type theory ensures that both validate all the patch laws of the theory.

6.4 Example Use of Patch Laws

We know from Section 3 that merge follows from pseudocommutation. Although we have not yet defined pseudocommutation for this patch language, it is worth illustrating how the patch laws described here would be involved in proving its correctness. In this setting `pcom` should have the following type:

```
pcom : {n1 n2 n3 : Nat}
  {h1 : History n1}  {h2 : History n2} {h3 : History n3}
  → doc h1 = doc h2 × doc h2 = doc h3
  → Σ {n2' : Nat}. Σ {h2' : History n2'}.
      doc h1 = doc h2' × doc h2' = doc h3
```

For example, the function `pcom` should act as follows:[12]

```
pcom (addP "a" 0 , addP "b" 1) =
     (addP "b" 0 , (ap doc ADD-ADD-<) ∘ addP "a" 0)
```

(In this call to `com`, `h1 = []` and `h3 = (ADD "b"@1 :: ADD "a"@0)`). This says that pseudocommuting "add a at line 0" with "add b at line 1" gives "add b at line 0" and "add a at line 0" (composed with `ADD-ADD-<` to make its output history match that of the input).

The `square` law, which states that the top and bottom composites of the pseudocommutation square are equal, says that

```
pcom (p,q) = (r,s)  →  q ∘ p = r ∘ s
```

Thus, for this example of how `pcom` should execute, the square law requires that

```
  addP "b" 1 ∘ addP "a" 0
= (ap doc ADD-ADD-<) ∘ addP "b" 0 ∘ addP "a" 0
```

Modulo expanding the definition of `PathOver` in the type family `x.doc h = doc x`, this is exactly what the `addP-addP-<` law states. This illustrates the role that the patch laws defined here would play for verifying the square law for `pcom`.

7. Related and Future Work

We have shown how patch theory [11] in the style of Darcs [29] can be developed in homotopy type theory. The principal contribution of the present work is to reformulate patch theory using the tools of homotopy type theory, namely identity types, higher inductive types, and univalence, to clearly separate the formal theory of patches from its interpretation in terms of basic revision control mechanisms. Many standard tools of homotopy theory come into play, demonstrating the use of these methods in a practical programming context.

Several prior category-theoretic analyses of version control have been considered. Jacobson [15] interpret patches as inverse semigroups, which are essentially partial bijections. Mimram and Di Giusto [27] analyzes merging as a pushout, an alternative to the pseudocommutation-based merging we consider here. Houston [14] also discusses merge as pushout, and a duality with exceptions. Our contribution, relative to these analyses, is to present patch theory in a categorical setting that is also a programming formalism, so it directly leads to an implementation. These analyses consider settings where not all maps are invertible. In homotopy type theory the path space of every type is symmetric, and to fit patch theories into this symmetric setting, we either considered a language where all patches were naturally total bijections on any repository (Section 4 and 5), or used types to restrict patches to repositories where they are bijections (Section 6). If we had a directed homotopy type theory (see [20] for some initial work towards this)—with, for example, a universe of partial bijections—we could perhaps apply these analyses more directly.

Dagit [9] present an approach to proving some invariants of a version control implementation using advanced features of Haskell's type system. Camp (Commute And Merge Patches) [6] is

[12] We leave some additional arguments implicit, relative to the types given above.

an experimental version control system based on Darcs; the Camp project aims to prove the correctness of its patch theory in Coq. It would be interesting to formalize the programs we have written here in Coq or Agda, to investigate the effect of using homotopy type theory relative to this prior work.

To define merging on the patch theory with `ADD` and `RM`, we must complete the definition of `pcom` discussed in Section 6.4. As described in Section 4, to map out of a path type we must provide an induction principle for patches `doc h = doc h'`. Intuitively, this should be possible because a history `h` is essentially a reified path `doc [] = doc h`. In particular, we would certainly have to define clauses of `pcom` for pairs of composable primitive patches `(p,q)`. (The square and rotation laws guide the behavior of `pcom` on compositions or inverses of primitive patches, as described in [15].) We believe we can define `pcom (p,q)` by using the function `interp' (q ∘ p) : S(h) → S(··· :: ··· :: h)` to compute `(···,···)`, a length-2 suffix for the history `h`, corresponding precisely to `(p,q)`.

This fits with a long-range goal for homotopy type theory, which is to develop general characterizations of identity types—which, if it is possible, may have significant applications in algebraic topology.

An additional line of future work would be to consider different patch theories. For example, the patch languages considered here are either unityped, in that they have only one context (Section 4 and Section 5) or singleton-typed, in that each patch context has only one repository in it (Section 6). It would be interesting to consider whether there are any interesting alternatives in between.

Acknowledgments We thank the participants of the 2013 IFIP WG 2.8 meeting for helpful conversations about this work.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretic Computer Science*, 342(1):3–27, 2005.

[2] T. Altenkirch. Containers in homotopy type theory. Talk at Mathematical Structures of Computation, Lyon, 2014.

[3] S. Awodey and M. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 2009.

[4] B. Barras, T. Coquand, and S. Huber. A generalization of Takeuti–Gandy interpretation. To appear in *Mathematical Structures in Computer Science*, 2013.

[5] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. Preprint, September 2013.

[6] Camp Project. `http://projects.haskell.org/camp/`, 2010.

[7] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

[8] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. INRIA, 2009. Available from `http://coq.inria.fr/`.

[9] J. Dagit. Type-correct changes—a safe approach to version control implementation. MS Thesis, 2009.

[10] N. Gambino and R. Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(3):94–109, 2008.

[11] Ganesh Sittampalam et al. Some properties of darcs patch theory. Available from `http://urchin.earth.li/darcs/ganesh/darcs-patch-theory/theory/formal.pdf`, 2005.

[12] R. Garner. Two-dimensional models of type theory. *Mathematical. Structures in Computer Science*, 19(4):687–736, 2009.

[13] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[14] R. Houston. On editing text. `http://bosker.wordpress.com/2012/05/10/on-editing-text/`, 2012.

[15] J. Jacobson. A formalization of darcs patch theory using inverse semigroups. Available from `ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf`, 2009.

[16] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. arXiv:1211.2851, 2012.

[17] F. W. Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the context of Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.

[18] D. R. Licata and G. Brunerie. $\pi_n(S^n)$ in homotopy type theory. In *Certified Programs and Proofs*, 2013.

[19] D. R. Licata and E. Finster. Eilenberg–MacLane spaces in homotopy type theory. Draft available from `http://dlicata.web.wesleyan.edu/pubs.html`, 2014.

[20] D. R. Licata and R. Harper. 2-dimensional directed type theory. In *Mathematical Foundations of Programming Semantics (MFPS)*, 2011.

[21] D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[22] D. R. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *IEEE Symposium on Logic in Computer Science*, 2013.

[23] P. L. Lumsdaine. Weak ω-categories from intensional type theory. In *International Conference on Typed Lambda Calculi and Applications*, 2009.

[24] P. L. Lumsdaine. Higher inductive types: a tour of the menagerie. `http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/`, April 2011.

[25] P. L. Lumsdaine and M. Shulman. Higher inductive types. In preparation, 2013.

[26] C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000.

[27] S. Mimram and C. Di Giusto. A categorical theory of patches. *Electronic Notes in Theoretic Computer Science*, 298:283–307, 2013.

[28] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[29] D. Roundy. Darcs: Distributed version management in haskell. In *ACM SIGPLAN Workshop on Haskell*. ACM, 2005.

[30] M. Shulman. Homotopy type theory VI: higher inductive types. `http://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html`, April 2011.

[31] M. Shulman. Univalence for inverse diagrams, oplax limits, and gluing, and homotopy canonicity. arXiv:1203.3253, 2013.

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

[33] B. van den Berg and R. Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[34] V. Voevodsky. Univalent foundations of mathematics. Invited talk at WoLLIC 2011 18th Workshop on Logic, Language, Information and Computation, 2011.

[35] M. A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008.