Homotopical Patch Theory

(Expanded Version)

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

Edward 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. In homotopy type theory, the propositional equality type becomes proof-relevant, and corresponds to paths in a space. This allows for a new class of datatypes, called higher inductive types, which are specified by constructors not only for points but also for paths. In this paper, we consider a programming application of higher inductive types. Version control systems such as Darcs are based on the notion of patches—syntactic representations of edits to a repository. We show how patch theory can be developed in homotopy type theory. Our formulation separates formal theories of patches from their interpretation as edits to repositories. A patch theory is presented as a higher inductive type. Models of a patch theory are given by maps out of that type, which, being 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.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

General Terms Languages, Theory

1. Introduction

Martin-Löf's intensional type theory (MLTT) is the basis of proof assistants such as Agda [29] and Coq [9]. Homotopy type theory is an extension of MLTT based on a correspondence with homotopy theory and higher category theory [4, 11, 13, 14, 24, 36–38]. 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 represented 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 those homotopies, etc. This correspondence has suggested several extensions to type theory. One is Voevodsky's *univalence axiom* [17, 37], which describes the path structure of the universe (the type of small types). Another is *higher inductive types* [25, 26, 32], 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 [19, 20, 23, 35].

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 [5, 6, 22, 33]. 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 [21] 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 example of using higher inductive types in programming. The example we consider is *patch theory* [7, 10, 15, 16, 28, 31], inspired by the version control system Darcs [31]. Intuitively, a patch is a syntactic representation of a function that changes a repository. A patch ("delete file f") applies

* 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.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628158

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 perhaps inverses (undo) — which is present in some formulations of patch theory but not others. These satisfy certain general laws—composition is associative; inverses cancel. Moreover, there are domain-specific patch laws about the 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. Several syntactic transformations on patches are considered, such as merging, which reconciles divergent edits to a repository, and cherry-picking, which selects a subset of changes to merge. The semantics and syntactic transformations are required to satisfy certain laws, such as the fact that applying a composition of patches has the same effect as the composition of applying the patches (facilitating optimization), and that merging is a symmetric operation, so that independently computed merges agree (facilitating collaboration).

Building on this work, we develop patch theory in the context of homotopy type theory, using paths to represent aspects of patch theory. Specifically, we represent *patches as paths* — making use of the proof-relevant notion of equality in homotopy type theory — and we represent the laws that patches and transformations must satisfy as paths-between-paths. 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 [18]. 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 merging, 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 recursion 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 recursion principle for patches is analogous to calculations of homotopy groups in homotopy 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 them.

Homotopy type theory is still under development, and one of our goals in this paper is to provide a worked example that can motivate future work on it. First, we use an informal concrete syntax for higher inductive types and pattern-matching functions on them; this is similar to the informal type theory used in the Homotopy Type Theory book [35], but with a more programming-oriented notation. Our development using this syntax could be translated to Agda or Coq, using techniques to simulate higher inductives, but we have not yet implemented the examples in this paper in a proof assistant. Second, because a full interpretation of homotopy type theory as a programming language is work in progress, we do not have a formal operational semantics that we can use to run the programs in this paper. However, we will speculate on how we expect these specific programs to run, based on existing work on this topic [5, 6, 22, 33]. One interesting issue that arises is that several of the examples in the paper are functions into contractible types, which are types with exactly one inhabitant up to homotopy. Thus, up to paths/propositional equality, such a function can return any element of the type, but based on existing work on the computational interpretation, we expect that the functions we define will in fact compute the elements we intend them to.

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

2. Basics of Homotopy Type Theory

We review some basic definitions; see [35] for more details.

2.1 Paths

In type theory, there are two notions of equality. *Definitional equality* is a proof-irrelevant judgement relating two terms. It is a congruence containing operational steps like β-reduction (`(λ x.e) e'` is definitionally equal to `[e'/x]e`). Uses of definitional equality are not marked in the proof term or program: if `e` has type τ, then `e` also has any other type τ' that is definitionally equal to τ. On the other hand, *propositional equality* is a proof-relevant *type* relating two terms; it is often also called the *identity type*, which we write `e = e'`. Uses of propositional equality *are* explicitly marked in the program: if `e` has type τ and `p` is an element of the identity type $\tau = \tau'$, then `coe p e` has type τ'.

In homotopy type theory, elements of the identity type are used to model a notion of *paths in a space* or *morphisms in a groupoid*. Using the identity type, specified by its introduction rule, reflexivity, and elimination rule, known as path induction or J, one can define path operations including a constant path `refl` (witnessing the reflexivity of equality); composition of paths `q ∘ p` (witnessing the transitivity of equality)[2], and the inverse of a path `! p` (witnessing the symmetry of equality). Moreover, there are *paths between paths*, or *homotopies*, which are represented 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
(r ∘ q) ∘ p = r ∘ (q ∘ p)
(! 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 $\mathtt{x} =_{\mathtt{A}} \mathtt{y}$ to paths $\mathtt{f(x)} =_{\mathtt{B}} \mathtt{f(y)}$. Logically, this expresses that propositional equality is a congruence; homotopically, it expresses that any function has an **a**ction on **p**aths; and categorically, it expresses that functions are *functors*, preserving the path

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

structure of types. `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 (q ∘ p) = (ap f q) ∘ (ap f p)
```

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

```
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)` correlated by a path `p : a1 = a2`. Logically, it is a kind of heterogeneous equality [27]; categorically, it is a path in the total space of the fibration determined by the type family. For `apd`, this kind of 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 parallel paths in `A` are equal—for any two elements `m,n:A`, and any two proofs `p,q : m = n`, there is a path `p = q`. Similarly, a type is a 1-type iff any two paths between parallel paths are equal. A type is a *mere proposition*, or (-1)-type, iff any two elements are equal. A type is *contractible* iff it is a mere 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 $\mathtt{A} =_{\mathtt{Type}} \mathtt{B}$ 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

```
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 between types lifts to a bijection on lists or monoids over those types; 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, and writing `f2 . f1` for (λ `x -> f2(f1(x))`):

```
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
(f2,g2,p2,q2) ∘b (f1,g1,p1,q1) = (f2 . f1, g1 . g2, ...)
```

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

[3] For types that are not sets, univalence requires a notion of *equivalence* that generalizes bijection. However, here we will only use it for sets.

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

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 (q ∘ p) x = coe q (coe p 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)
```

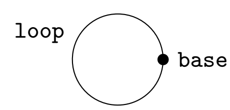
That is, coercing along a path constructed by univalence applies the given bijection. Because `! (ua (f,g,p,q)) = ua (!b (f,g,p,q))`, we also have that

```
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 `coerc`ing 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 `coerc`ed along.

2.4 Higher Inductive Types

Ordinary inductive types are specified by *generators*; for example, the natural numbers have generators zero and successor: `zero : Nat` and `succ : Nat → Nat`. *Higher-dimensional inductive types* (or just *higher inductive types*) [25, 26, 32] 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:

```
space Circle : Type where
  -- point constructor:
  base : Circle
  -- path constructor:
  loop : base = base
```

`base` constructs an element of the inductive type (taking no arguments, just like `zero : Nat`). `loop` generates a path on the circle, which is an element of the identity type $\mathtt{base} =_{\mathtt{Circle}} \mathtt{base}$—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' =`$_{\mathtt{X}}$ `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'     -- does not typecheck!
```

The second equation does not quite make sense, because `f` is a function `Circle → X` but `loop` is a *path* on the circle. Therefore we 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 `l'`.

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 [23, 35] for topological intuition.

3. General Patch Theory

Patch theory [7, 10, 12, 15, 16, 28] provides a general framework for describing properties of version control systems, which allows us to specify the behavior of patches under operations such as composing, reverting and merging. Here, we formulate patch theory in the context of homotopy type theory. This allows us to separate the purely algebraic aspects of a version control system (the laws that it must obey) from its implementation details (how repositories and patches are represented). We refer to a particular algebraic characterization of a version control system as a theory of version control, or a *patch theory*; and to an implementation that obeys the laws of such a theory as a *model* of that theory.

In a patch theory, each patch comes equipped with specified domain and codomain *contexts*, representing respectively, the repository states 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.

3.1 Patch Theories as Higher Inductive Types

Homotopy type theory allows us to present a patch theory as a higher inductive type whose structure encodes both generic aspects of version control (such as the behavior of patches under composition) as well as the aspects particular to the given theory, specifying the basic patches available and the specialized laws that these patches obey. An advantage of this approach is that in homotopy type theory functions are functors that necessarily preserve the path structure of a type, so that any function we define out of the higher inductive type representing a patch theory must validate all the laws of that theory, and thus determines a model for it. An additional benefit of this approach is that the metatheory of homotopy type theory itself enforces the groupoid laws, so that we need not specify the behavior of patches and patch laws under composition—i.e. that all compositions are associative, unital and respect inverses—this all comes for free from the groupoid structure of higher inductive types. In the following sections we will present several examples of patch theories encoded as higher inductive types, together with interpretations for them as functors to a universe of sets.

When encoding a patch theory as a higher inductive type, patch contexts are represented as points of the type. Patches are represented as paths between the representations of their domain and codomain contexts, with the path operations refl, $q \circ p$ and $!p$ representing a no-op patch, patch composition, and undo, respectively. Encoding patches as paths in a higher inductive type imposes the requirement that they have inverses, as opposed to just retractions. As one would expect, applying the inverse of a patch after applying the patch itself ($!p \circ p$) undoes the effect of the patch. But it is also possible to apply the inverse patch first ($p \circ !p$), to an appropriate repository state, and this composition should also be equivalent to doing nothing. This forces us to use some care when defining contexts and patches. In some cases we use inverse patches directly in our theory, while in others they end up getting in the way and we must work around them.

Patch laws are represented as 2-dimensional paths between paths. Patch laws are helpful for reasoning about syntactic transformations on patches, such as an optimizer, which should compute a patch equivalent to the one it is given, or a merge, which, given two divergent edits, computes a pair of patches that reconciles them.

3.2 Merging

At a minimum, merging is an operation that takes a pair of diverging patches or *span*, (f_1, f_2), and returns a pair of converging patches or *cospan*, (g_1, g_2), which is a *reconciliation* of the span in the sense that

$$\mathit{merge}(f_1, f_2) = (g_1, g_2) \Longrightarrow g_1 \circ f_1 = g_2 \circ f_2 :$$

$$\begin{tikzcd} & A \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & \\ B \arrow[dr, dashed, "g_1"'] & & C \arrow[dl, dashed, "g_2"] \\ & D & \end{tikzcd}$$

In order to support distributed version control systems, we will further require that the merge operation be *symmetric*,

$$\mathit{merge}(f_1, f_2) = (g_1, g_2) \Longrightarrow \mathit{merge}(f_2, f_1) = (g_2, g_1)$$

so that your reconciliation of my changes with your changes agrees with my reconciliation of your changes with my changes.

Depending on the circumstances, we may wish to impose other laws on merge as well. For example, in the patch theory underlying the distributed revision control system Darcs[12, 31], the merge operation is required to respect patch inverses in the sense that,

$$\mathit{merge}(f_1, f_2) = (g_1, g_2) \Longrightarrow \mathit{merge}(g_1, !f_1) = (!g_2, f_2)$$

A symmetric reconciliation with this property is equivalent to—indeed, the categorical mate of—an operation known as *pseudo-commutation*, which is the primitive operation in terms of which the other operations of Darcs' patch theory are defined.

It is always possible to define a total merge function, since for any span we may give $\mathit{merge}(f_1, f_2) = (!f_1, !f_2)$, the reconciliation that undoes both changes. This can be used to signal a *merge conflict*, a situation in which we are unable to automatically reconcile the competing changes in a sensible way, and for which human intervention is required.

It is important to realize that a merge function that is a symmetric reconciliation need not respect the groupoid structure of a higher inductive type. For example, we may define merge recursively by *tiling*, that is, define

```
merge (g ∘ f , h) = (h'' , g' ∘ f')
  where
     (h' , f') = merge (f , h)
     (h'' , g') = merge (g , h')
```

$$\begin{tikzcd} & & A \arrow[dl, "f"'] \arrow[dr, "h"] & \\ & B \arrow[dl, "g"'] \arrow[dr, dashed, "h'"'] & & C \arrow[dl, dashed, "f'"] \\ D \arrow[dr, dashed, "h''"'] & & E \arrow[dl, dashed, "g'"] & \\ & F & & \end{tikzcd}$$

If we define merging a patch with a no-op and merging a patch with itself by

```
merge (f , refl) = (refl , f)
merge (f , f) = (refl , refl)
```

then under the assumption that `f` conflicts with `h`, merging `(!f ∘ f , h)` by tiling results in a conflict, whereas first performing the composition yields `(h , refl)`.

Nevertheless, we may still define merge recursively by quotienting syntactic paths by the groupoid and domain-specific patch laws, and choosing a canonical representative for each class. For example, in a theory without any domain-specific laws, we may normalize `(z ∘ (refl ∘ y)) ∘ ((x ∘ !w) ∘ w)` to `z ∘ y ∘ x`, with a canonical association. In the presence of domain-specific laws, these would need to be taken into account as well. We will make use of this technique in section 6.

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. An Elementary Patch Theory

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 to any repository state, 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 this simple Repository and its patches:

```
space R : Type where
  -- point constructor (patch context):
  num  : R
  --  path constructor (basic patch):
  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 [8] to assert that

```
assoc : compose r (compose q p) = compose (compose r q) p
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 = reflb
interp (compose p2 p1) = interp p2 ∘b interp p1
interp (inv p) = !b (interp p)
```

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 represents 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).[4]

Moreover, `interp` takes path operations to the corresponding operations on bijections, because it is defined via `ap`, and `ap` preserves the path operations. For example,

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

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

For example, if we apply[5] 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 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 of our program. This 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 [23, 35]. `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 → Type
I' num = Bool
ap 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 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 [23, 35]). The idea that we can have multiple models of a patch theory (`I` and `I'`) will be exploited in Section 6, when we give a "logging" interpretation that produces a data representation of what happens when a patch is evaluated.

4.2 Merge

Next we implement a merge operation, which satisfies the laws discussed in Section 3. Writing `Patch` for `doc = doc`, and specializing the interface to the setting where we have only one context, we need to implement the following:

```
merge : Patch × Patch → Patch × Patch
reconcile : (f1 f2 g1 g2 : Patch)
          → merge (f1 , f2) = (g1 , g2)
          → g1 ∘ f1 = g2 ∘ f2
symmetric : (f1 f2 g1 g2 : Patch)
           → merge (f1 , f2) = (g1 , g2)
           → merge (f2 , f1) = (g2 , g1)
```

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

```
merge(f1 , f2) = (f2 , f1)
```

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

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

For `symmetric`, because `g1 = f2` and `g2 = f1`, we need to show that `merge (f2,f1) = (f1,f2)`, which is true by definition.

For `reconcile`, we need to prove that `f2 ∘ f1 = f1 ∘ f2`, for any two loops `num = num` on the circle—the group of loops on the circle is abelian. 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 (or strong) 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 [19, 20, 23, 35], which can be applied here.

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 canonical representatives ("add x, for $x \in \mathbb{Z}$") for equivalence classes of patches in the above patch theory, considered modulo the group laws. This is proved by giving functions back and forth that compose to the identity. The function `encode : 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 the `Int`, viewing `Int` as a datatype with three constructors, `0`; `+ n` (where `n` itself is positive) representing positive `n`; and `- n` (where `n` itself is positive) representing negative `n`.

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

The proof that `encode` and `repeat` are mutually inverse is described in [23, 35]. 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 `f2 ∘ f1 = f1 ∘ f2`, 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 merge 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!

One further point to note is that, in this example, we were able to *define* merge without converting paths to integers, while to prove the reconciliation property we needed to reason inductively, using canonical representatives of group-law-equivalence-classes. This is because all patches commute, so we can define `merge(x,y) = (y,x)` without analyzing the structure of patches. In more interesting settings, such as Section 6, we will need to make use of such representatives to define the merge function itself. To illustrate this, we give an alternate definition of `merge`, called `merge'`, which uses a helper function `mergeI` that recursively swaps two integers; writing the code in this way illustrates a method for defining merge by chosing canonical representatives for paths and then analyzing these representatives.

```
merge'(p,q) =
  let (a,b) = mergeI(encode p, encode q)
   in (repeat a , repeat b)

mergeI : Int × Int → Int × Int
mergeI(+ (1+x) , - (1+y)) =
  let (a , b) = mergeI (+ x , - y)
   in (a-1 , b+1)
...
```

The function `merge'` is defined by converting the given paths `p` and `q` (which are considered up to the group laws, such as associativity) to chosen representatives, integers. Paths that are equal according to the group laws are sent to equal representatives; e.g. both `(add1 ∘ add1) ∘ add1` and `add1 ∘ (add1 ∘ add1)` are sent to 3. We may then compose this choice of representatives with any function we want — including functions that do not themselves respect the group laws, as merge in general might not (see Section 3) — and the overall composite still respects the group laws. In this case, we compose with a function `mergeI` that case-analyzes the given integers, and recursively "merges" the two numbers with cases such as the one given above. This case copies a positive successor on the left to a positive successor on the right, and a negative successor on the right to a negative successor on the left—think of it as merging "add 1 and then do x" with "subtract 1 and then do y" by merging `x` and `y` and then moving the "add 1" to the right and the "subtract 1" to the left. Finally, once `mergeI` has computed the merge of two chosen representatives, `merge'` calls `repeat` to convert the resulting integers back to paths. One can prove by induction that `mergeI(x,y) = (y,x)`; and `encode` and `repeat` are mutually inverse, so `merge'` agrees with the original definition of `merge`.

5. A Patch Theory 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 this 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. We consider an interpretation function `I` and a simple patch optimizer; we do not consider merge in this section, because we discuss it for the more general language in Section 6.

5.1 Definition of Patches

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

```
space R : Type where
  -- point constructor (patch context):
  doc : R
  -- path constructor (basic patch):
  _↔_@_ : (s1 s2 : String) (i : Fin n) → (doc = doc)
  -- path-between-path constructors (patch laws):
  indep : (s t u v : String) (i j : Fin n) → (i ≠ j) →
          (s ↔ t @ i) ∘ (u ↔ v @ j)
        = (u ↔ v @ j) ∘ (s ↔ t @ i)
  noop : (s : String) (i : Fin n) 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 line number i. `Fin n` is the type of natural numbers less than n, which we interpret here as line numbers in an n-line document (where we start numbering at 0).

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; 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 (`x ≠ y` is the negation of `x = y`, i.e. `(x = y) → void`).

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 we have the following computation rules

```
f(doc) = doc'
β1 : ap f (s ↔ t @ i) = swap' s t i
β21 : PathOver (x . x = refl) β1
                (ap (ap f) (noop s i))
                (noop' s i)
β22 : PathOver (x,y. x ∘ y = y ∘ x) (β1, β1)
                (ap (ap f) (indep s t u v i j neq))
                (indep' s t u v i j neq)
```

The first computation rule is in fact a definitional equality, while the second is a path. The well-typedness of the third computation rule, which says roughly that `ap (ap f) (noop s i)` equals `(noop' s i)`, depends on the second computation rule, $\beta 1$. This is because `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` by $\beta 1$. Thus, we use the PathOver type state them, which allows for a heterogeneous equality. 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 omit the details of the final two, which are not used 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 patch theory 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 length-n vector of strings, represented by the type `Vec String n`.

```
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 s t u v i j i≠j) =
  GOAL0 : ua(swapat(s,t) i) ∘ ua(swapat(u,v) j)
        = ua(swapat(u,v) j) ∘ ua(swapat(s,t) i)
ap (ap I) (noop s i) = GOAL1 : 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 metavariables GOAL0 and GOAL1 represent *goals*, that is, terms that must still be provided before the proof/program is complete. The image of indep and noop are the goals GOAL0 and GOAL1, 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, GOAL1 is solved by observing that `swapat(s,s) i` is the the identity bijection, and then using the fact that `ua reflb = refl`. GOAL0 is solved by turning both sides into a composition of bijections using the fact that `ua b2 ∘ ua b1 = ua (b2 ∘b b1)` , and then proving the corresponding fact about swapat:

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

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 s i) =
  GOAL0 : opt0 s s i = refl
ap (ap opt1) (indep s t u v i j i≠j) =
  GOAL1 : 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 `GOAL0` and `GOAL1` whose types are given above. The goal `GOAL0` is true because `String.equals s s` will be true, so, after case-analysis, `refl` proves that `opt1 s s i = refl`. The goal `GOAL1` 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 using `R`-induction:

```
opt1-correct : (x : R) → x = opt1 x
opt1-correct doc = refl
apd opt1-correct (s1 ↔ s2 @ i) =
  GOAL0 : PathOver (x. x = opt1 x) (s1 ↔ s2 @ i) refl refl
apd (apd opt1-correct) (noop s i) = GOAL0
apd (apd opt1-correct) (indep s t u v i j i≠j) = GOAL1
```

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: when `s1` is equal to `s2`, we use `noop`; when it is not, 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

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

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 proven.

However, if we do not wish to consider proof-relevant patch equations, we *can* make these goals trivial by a technique called *truncation* [35, Chapter 7]. In this case, this means adding another constructor to `R` of type

```
-- path-between-path-between-path constructor
   (all proofs of patch laws are equal)
trunc : (x y : R) (p q : x = y) (f1 f2 : p = q)
        → f1 = f2
```

This constructor adds a path between any two faces `f1` and `f2`—which allows the above goals to be solved. The price for truncating 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:[6]

```
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

[6] 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

$$\begin{tikzcd} {} \arrow[r, "\mathtt{refl}"] \arrow[d, "\mathtt{p}"'] & {} \arrow[d, "\mathtt{q}"] \\ {} \arrow[r, "\mathtt{refl}"'] & {} \end{tikzcd}$$

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).

```
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 s1 i)
                      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 [35, Lemma 3.11.8]. The identity types of any contractible type are mere propositions, so any two paths are connected by a face. Thus, because we formulated the problem as mapping into a contractible type, the remaining goals are trivial.

This definition of `opt`, consisting of only the three cases given above, is 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 key 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 provided 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)` is 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. This is consistent with prior experience with, for example, function extensionality in type theory [3]. A higher-order computation will compute a *particular* function, not any function with the same graph; computation does not respect extensional equality of functions.

6. A Patch Theory With Richer Contexts

In the previous section we considered patches of the form `s1 ↔ s2 @ i`, which naturally induce 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 a richer language of patches—inserting a string `s` as the `l`th line in a file (`ADD s@l`), and removing the `l`th line of a file (`RM l`).These patches only make sense in certain situations. 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 deletes `s`.

A suitable patch theory must express such constraints on composition using contexts. More than one context is required, because not all patches are composable with one another. For example, it makes sense to classify repositories by the number of lines they contain so that removal from an empty file is ruled inadmissible by the patch theory. This may be achieved by defining the contexts (the points of `R`) to be of the form `doc n`, where `n` is of type `Nat`. The patch that adds a line is, generally in `n`, a path in `R` witnessing `doc n = doc n+1`. Similarly, the patch that deletes a line is a path in `R` witnessing `doc n+1 = doc n`. Although this formulation expresses necessary constraints on the use of the primitive patches, it fails to admit the obvious interpretation of `doc n` as the type of `n`-line files, `Vec n String`. The difficulty is that the type theory demands that the interpretation respect paths, and we have `doc n = doc n+1` in `R`, yet the types `Vec n String` and `Vec n+1 String` are not bijective, and hence are not equated by univalence.

To motivate what follows, let us observe that we may expect there to be an "initial" context describing the empty repository, the initial state of a repository. Because there is only one empty repository, the empty file, we would expect the interpretation of the initial context to be a contractible type whose element is the empty file. Moreover, we would expect there to be a path from the initial context to every other context, based on the idea that it should be possible to reach every repository state by some sequence of patches. Thus, every context would be equal to the initial context, and by functoriality and univalence all contexts must be modeled by a contractible type.

Given that the interpretation of a context should be a type containing repository files as elements, its contractibility, together with univalence, suggest that each context be modeled by the *singleton type*, `S(file)`, containing *only* the file in question. But if the meaning of a context is to be such a singleton, the context must essentially determine the contents of the repository. The obvious way to achieve this is to consider contexts of the form `doc n file`, where `n` is, as before, the number of lines, and `file` is a file of that length, an element of `Vec n String`. The patch `ADD s@l` would then be a path `doc n file = doc n+1 file'`, where `file'` is the result of adding `s` at line `l` in `file`, and similarly for `RM l`. We can interpret `doc n file` as `S(file)` functorially, because any two singletons, being contractible, are equivalent types, and hence equal by univalence.

The trouble with this formulation is that it intermixes the abstract theory of patches with its concrete realization as a file. Although we reject it as a solution, it does suggest another, more satisfactory, formulation. The main idea is that the contexts need only *determine* the contents of the repository, not literally contain them, in order to construct the singletons model. This is achieved by indexing contexts by *patch histories*, which are sequences of composable patches applicable to files of a given length. With respect to any particular realization of patches, a history applicable to the empty file uniquely determines the resulting file's contents. As an added benefit, histories also reify sequences of patches in a way which facilitates certain operations on repositories, such as moving forward or backward in time.

Patch contexts may be understood as types for patches, limiting how they may be composed, and we expect these to be erasable at run-time. This will require us to compute views of identity types that are more amenable to computation, similarly to the way in which we used the characterization of the loop space of the circle in Section 4.

6.1 Definition of Patches

Let `History m n` be the type of patch histories (sequences of patches) applicable to `m`-line files which result in `n`-line files. We will define `History m 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.

```
space History : Nat → Nat → Type where
  -- point constructors:
  [] : History m m
  ADD_@_::_ : {m n : Nat} (s : String) (l : Fin n+1) →
              History m n → History m n+1
  RM_::_    : {m n : Nat} (l : Fin n+1) →
              History m n+1 → History m n
  -- path constructors:
  ADD-ADD-< : {m n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
    (s1 s2 : String) (h : History m 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 m 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.

Histories applicable to the empty file (elements of `History 0 n`) uniquely identify files because the history can be "replayed" from the start. These *complete histories* will serve as the patch contexts in this language—the domain of a patch is a complete history identifying a file to which the patch is applicable, and the codomain is the domain history extended by the patch which was just applied.

```
space R : Type where
  -- point constructor:
  doc : {n : Nat} → History 0 n → R
  -- path constructors:
  addP : {n : Nat} (s : String) (l : Fin n+1)
         (h : History 0 n) → doc h = doc (ADD s@l :: h)
  rmP  : {n : Nat} (l : Fin n+1)
         (h : History 0 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 0 n` with the path `ADD-ADD-<`. Therefore, we can stipulate that the above two paths are equal *over* the `ADD-ADD-<` equation from `History 0 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 0 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 0 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 `Strings`.

```
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
```

We want to define a function `I : R → Type` which models, or interprets, points of `R` (complete 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 0 n1}
         {h2 : History 0 n2} → (doc h1 = doc h2)
         → Bijection (I (doc h1)) (I (doc h2))
```

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 there are now many patch contexts, instead of the single `doc`. As a result, we must choose the interpretation of each `doc h` into the type universe.

As we discussed at the beginning of this section, we cannot simply interpret `doc h` as `Vec String n`, because these types are not bijective. 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 which file is described by this history, 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:

```
tosingleton : (f : A → B) → {M : A} → S(M) → S(f M)
tosingleton 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 (tosingleton (add s l)))
ap I (rmP l h) = ua (single-biject (tosingleton (rm l)))
apd' (ap I) (addP-addP-< l1 l2 s1 s2 h p) =
  <replay respects this patch law>
apd' (ap I) (addP-addP-≥ l1 l2 s1 s2 h p) =
  <replay respects this patch law>
```

where `apd'` is a function which gives the action of a function on a `PathOver` (we omit the details, since they will not be used below), and `replay` is a function which steps through a complete history to compute the file specified by that history:

```
replay : {n : Nat} → History 0 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) =
  GOAL0 : 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) =
  GOAL1 : 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 in `GOAL0` and `GOAL1` 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-biject (tosingleton (\ h → ADD s@l :: h))
ap I' (rmP l h) =
  ua (single-biject (tosingleton (\ 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

interpH : doc h = doc h' → S(h) → S(h')
interpH p = coe (ap I' p)
```

Then `interpH` takes a patch `p`, 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 (contexts)—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 Merge

In Section 3, we said that `merge` takes any pair of diverging patches to a pair of converging patches that reconciles them. Our definition of `merge` in Section 4 accepted arbitrary pairs of patches, because that patch theory had a single context `num`.

Now that we have history-indexed patches, we might expect `merge` to take a pair `(doc h = doc h1) × (doc h = doc h2)` and, if a merge is possible, produce a pair `(doc h1 = doc h') × (doc h2 = doc h')`. However, as discussed in Section 3, even though some divergent patches are impossible to reconcile automatically—for example, given `addP s 0` and `addP s' 0`, we have no reason to favor either `[s,s']` or `[s',s]` over the other—we can produce a valid merge that simply undoes both edits. Therefore merge can be a total function that always returns a pair of patches `(doc h1 = doc h') × (doc h2 = doc h')`. A user-friendly system might recognize when merge undoes the edits, indicating a merge conflict, and prompt for manual intervention.

Defining such a function requires a "view" or derived recursion principle for these types, as we illustrated in Section 4.2. The `History` type characterizes the "forward-pointing" paths in `R` as sequences of composable primitive patches `ADD` and `RM`, modulo patch laws. But to define such a `merge` we would also need a characterization of, for example, the type of the path `!(rmP l' h)`,

```
doc {n} (RM l' :: h) = doc {n+1} h
```

The type `History n n+1` does not characterize this type, because the only elements of `History n n+1` lengthen the history `h` (by prepending sequences of `ADD`s and `RM`s), while this path shortens it. In other words, a history contains only compositions of primitive patches, and not their inverses. On the other hand, a merge involving inverse paths is not sensible in this patch theory. Although `!(rmP l' h)` and `addP s l (RM l' :: h)` have a common domain of `doc (RM l' :: h)`, the former is not part of the patch theory we are studying; instead, it was imposed by the natural symmetry of identity types.

We can avoid these undesirable inverses by restricting merge to divergent paths with shared domain `doc []`. This ensures that a patch `p` to be merged can be mapped to a complete history `interpH p []`. Users of a version control system will always encounter compositions of generating patches starting from the empty file, so this restriction does not come up in practice.

```
merge : {n1 n2 : Nat}
        {h1 : History 0 n1} {h2 : History 0 n2}
        (doc [] = doc h1) → (doc [] = doc h2) →
        Σ(n' : Nat). Σ(h' : History 0 n').
          (doc h1 = doc h') × (doc h2 = doc h')
```

Because this `merge` is more intricate than the one considered in Section 4, we will convert the input paths to complete histories using `interpH`, then define a function `mergeH` which computes merges of complete histories, and convert the resulting histories back into paths.

To reconcile two divergent complete histories, we define the notion that a history `h2` has `h1` as a prefix (or `h2` *extends* `h1`):

```
Extension : {n1 n2 : Nat} → History 0 n1
            → History 0 n2 → Type
Extension h1 h2 = Σ(s : History n1 n2). h1 ++ s = h2
```

Here, `++ : History n1 n2 → History n2 n3 → History n1 n3` appends two histories. Then, if we have a pair of complete histories `h1,h2`, we reconcile them by returning a history `h'` which extends both `h1` and `h2`. The suffixes of `h1` and `h2` yielding `h'` are the pair of converging patches produced by the merge.

```
mergeH : {n n1 n2 : Nat}
         (h1 : History 0 n1) (h2 : History 0 n2) →
         Σ(n' : Nat). Σ(h' : History 0 n').
           Extension h1 h' × Extension h2 h'
```

Once we have defined `mergeH`, we can convert its output back to paths. A complete history can be transformed into a path `doc [] = doc h` by repeated concatenation:

```
toPath : {n : Nat} (h : History 0 n) → doc [] = doc h
toPath [] = refl
toPath (ADD s@l :: h') = addP s l ◦ toPath h'
toPath (RM l :: h') = rmP l ◦ toPath h'
```

To turn an `Extension h h'` into a path, we need only travel from `doc h` to `doc []` and back to `doc h'`:

```
extToPath : {n n' : Nat}
            {h : History 0 n} {h' : History 0 n'} →
            Extension h h' → doc h = doc h'
extToPath _ = (toPath h') ◦ !(toPath h)
```

`extToPath` completely ignores the extension itself; intuitively, this is possible because extensions are more informative than paths, since the former contain only compositions of generators.

Combining these ingredients, we define `merge` as:

```
merge p1 p2 =
  let (n',(h',(e1,e2))) =
    mergeH (interpH p1 []) (interpH p2 [])
    in (n', (h', (extToPath e1, extToPath e2)))
```

This shows that `merge` reduces to `mergeH`. We have not yet defined `mergeH`, but we can reduce it to defining a merge on simple text files. Because `History 0 n` is quotiented by patch laws, we must show that `mergeH` sends equal histories to equal results. One way to handle this is to choose a representative for each equivalence class of histories, and then compute on these representatives. Here, we can use the function `replay` to convert each `History 0 n` to its file contents in `Vec n String`. Then, we could define a merge function directly on files (perhaps using existing algorithms), and then compute extensions of the input histories which result in that those files. Such a `mergeH` would necessarily respect the patch laws because `replay` does.

7. Related Work

Several prior category-theoretic analyses of version control have been considered. Jacobson [16] interprets patches in inverse semigroups, where they are essentially partial bijections. Mimram and Di Giusto [28] analyze merging as a pushout, which provides a canonical merge for every pair of patches, including a primitive representation of merge conflicts. Houston [15] 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 all identity types are 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).

Dagit [10] presents an approach to proving some invariants of a version control implementation using advanced features of Haskell's type system. Camp (Commute And Merge Patches) [7] is an experimental version control system based on Darcs; the Camp project aims to prove the correctness of its patch theory in Coq. We have not yet mechanized the programs described here, but our work provides another possible path to formalization.

Swierstra and Löh [34] explore the use of separation logic [30] for specifying the behavior of patches. In Section 6, we took repository contexts to be patch histories, but it would be interesting to consider using separation logic formulas to describe histories, which would allow for small-footprint specifications of patches.

8. Conclusion

Inspired by the patch theory of Darcs [12], which emphasizes the groupoid structure of patches, we have explored the formulation of patch theory within the framework of homotopy type theory. Patch theories are given as higher inductive definitions in which we specify generators for the points (path contexts), 1-dimensional paths (patches), and 2-dimensional paths (patch laws). The groupoid laws come "for free", and so need not be specified explicitly. An idealized implementation of a patch theory is given by a function mapping the patch theory into a univalent universe of sets and bijections. The sets are concrete repositories and the bijections are the actual actions of the patches on repositories. The mapping is intrinsically functorial, and hence must respect the intrinsic groupoid structure, but is given by the elimination principle for higher-inductive types, which demands that patches be realized by bijections satisfying the patch laws.

Besides these general structural considerations, some homotopy-theoretic concepts play a role in the development. In particular the "encode/decode" functions used to characterize identity types [23, 35] here become an implementation technique. For example, to define the merge of a span of patches (that is, a pair of paths with a common domain), we first pass to a concrete representation of paths, define the merge on the representation, and then pass back to a reconciliation, a cospan of patches (a pair of patches with common codomain) in the identity type. Other interpretations of a patch theory are definable in a similar manner, providing operations, such as logging, of practical interest for revision control.

There is much more to be done. Most importantly, the development of the merge operation in Section 6 is incomplete, because we have not proved the required properties of it. There we define `merge` using a partial characterization of the identity type of `R` as complete histories. Defining functions in either direction is sufficient to pass between these two representations, but not to prove properties of `merge` via properties of `mergeH`, such as the merge laws. For this, a more precise characterization is needed, namely an inductive type which is *equivalent* to the identity type, as $\mathbb{Z}$ is to the identity type of the circle. We leave this to future work.

More broadly, the computational interpretation of homotopy type theory must be developed further, including a fuller understanding of the interplay between definitional equality, propositional equality, and computation, and an understanding of what type erasure would mean in that setting. This includes developing a fuller understanding of "sub-homotopical" computation, meaning the mapping into contractible types.

One aspect of the homotopical framework that we have found limiting is the requirement that paths be symmetric (have inverses). In Darcs, inverses are used to define merging in terms of the elegant concept of pseudocommutation. But demanding inverses, rather than just retractions representing "undo" operations, is both conceptually questionable and practically problematic. We worked around the presence of inverses by refining contexts using patch histories, and using histories it was possible to define merge directly on only "forward" patches, rather than using inverses and pseudocommutation. Despite efforts, we were unable to formulate pseudocommutation in our setting, because to do so seems to require a complete characterization of the space of spans of patches. One direction for future work is to study this problem using the tools of homotopy theory to characterize such spans in a way that is amenable to our purposes. Another is to develop a type theory with non-symmetric paths (as suggested by [21]) grounded in directed homotopy theory, where we could formulate theories of partial patches without refining their contexts, which might simplify the development in Section 6.

Acknowledgments We thank the participants of the 2013 IFIP WG 2.8 meeting for helpful conversations about this work, and the anonymous reviewers for their helpful feedback on this article.

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] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now. In *Programming Languages meets Program Verification Workshop*, 2007.

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

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

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

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

[8] 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.

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

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

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

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

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

[23] 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.

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

[25] 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.

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

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

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

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

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

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

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

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

[34] W. Swierstra and A. Löh. The semantics of version control. Available from `http://www.staff.science.uu.nl/~swier004/`, 2014.

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

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

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

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

A. Addendum

In this addendum, we expand on the relationship among the patch theories considered in the main body of the paper, and clarify the role of histories in defining and analyzing the patch theory given in Section 6. These points were omitted from the published version for lack of time and space.

A.1 Interval

Presenting the patch theories of Sections 4 and 5 as higher inductive types automatically added useful inverse patches to the theory. For example, in Section 4 we only specified a path constructor for the add-one patch `add1`, but the subtract-one patch `(! add1)` was provided for free.

That patch theory had exactly one patch context `num`, so `add1` and `(! add1)` had the same type, `num = num`. In Section 6, not all `ADD` and `RM` patches were composable, so we needed many patch contexts. Moreover, we argued that inverse paths were undesirable in that setting, and restricted our attention to paths `doc [] = doc h`. To see why this effectively eliminates inverse paths from consideration, let us explore the *unit interval*, a simple but analogous example.

The unit interval is a higher inductive type defined as:

```
space I : Type where
  zero : I
  one : I
  seg : zero = one
```

We can regard `I` as a patch theory with two contexts `zero` and `one`, and one patch `seg`, which sends `zero` to `one`.

As usual, paths in `I` are automatically endowed with identities, inverses, and composition. Nevertheless, `zero = one` has no more elements than we put in—all paths of that type are homotopic to `seg`. Intuitively, this is because `(! seg)` goes "backwards" from `one` to `zero`, so any sequence of compositions yielding a path `zero = one` must have one more `seg` than `(! seg)`. For example,

```
seg ∘ ! seg ∘ seg : zero = one
```

but groupoid laws equate this to `seg`.

To prove this result, we will first show that `I` is contractible. It suffices to exhibit a point in `I`, the *center of contraction*, together with a proof that every point in `I` is equal to the center:

```
(x : I) → center = x
```

In this case, we choose the center to be `zero`.

```
is-contr : (x : I) → zero = x
is-contr zero = refl
is-contr one  = seg
apd is-contr seg = refl(seg)
  : PathOver (x. zero = x) seg refl seg
```

The last clause demands that the choice of path given by `is-contr` is continuous in the choice of codomain. That `PathOver` reduces to the type `seg = seg`, by the rules for path-over-a-path in identity types; this is true by `refl(seg)`.

Since `I` is contractible, it is also a mere proposition, and so all its identity types are contractible [35, Lemma 3.11.10]. In particular, to prove that `seg` is the unique element of `zero = one` up to homotopy, we can look at the action on paths of `is-contr`,

```
apd is-contr : {a b : I} (p : a = b)
             → PathOver (x. zero = x) p
               (is-contr a) (is-contr b)
```

If we specialize this to paths `zero = one`, we get

```
apd is-contr {zero} {one}
  : (p : zero = one) → PathOver (x. zero = x) p refl seg
```

This `PathOver` reduces to `p = seg`, yielding

```
apd is-contr {zero} {one} : (p : zero = one) → p = seg
```

which is a proof that all paths `zero = one` are homotopic to `seg`.

A.2 Open Interval

Let us extend the unit interval example by considering patch contexts indexed by natural numbers. The resulting *open interval* is a higher inductive type defined as:

```
space I* : Type where
  doc : Nat → I*
  add1 : (n : Nat) → doc n = doc n+1
```

Topologically, `I*` is the real half-line $[0, \infty)$. This is similar to the universal cover of the circle, discussed in Section 4.1, which corresponds to $(-\infty, \infty)$.

As a patch theory, `I*` has `Nat`-indexed contexts, and a patch from `doc n` to `doc n+1` for each `n`. The idea is that `I*` is the theory of natural number repositories, and the `add1` patch increments a repository's contents. As with the unit interval, inverse paths do not add any additional paths from `doc 0 = doc n`—up to homotopy, the only way to get from the empty document `doc 0` to a document `doc n` is to apply the `add1` patch `n` times. Hence, the codomain context `doc n` determines exactly which sequence of patches has occurred.

Note that this did not occur in Section 4, where we had a single patch context `doc`. Here, we have chosen more expressive contexts to track a repository's history—in this case, this only requires counting how many times we have applied the only applicable patch. (These patch contexts coincide with the repository contents, but that is not true in general.)

To prove there is a unique path `doc 0 = doc n`, we show that `I*` is contractible with center `doc 0`. As with `I`, it follows that `doc 0 = doc n` is also contractible.

We prove `I*` is contractible by `I*`-induction, which means that it suffices to show that, for any number `n`, we can construct a path `doc 0 = doc n` by composing `add1` with itself `n` times, and moreover, this choice of paths is continuous in the choice of codomain.

```
toPath : (n : Nat) → doc 0 = doc n
toPath 0 = refl
toPath (n+1) = add1(n) ∘ toPath n

is-contr : (x : I*) → doc 0 = x
is-contr (doc n) = toPath n
apd is-contr (add1 n) = refl
  : PathOver (x. doc 0 = x) (add1 n)
    (toPath n) (toPath n+1)
```

This last `PathOver` reduces to

```
(add1 n) ∘ (toPath n) = toPath n+1
```

which, once we expand the definition of `toPath n+1`, is true by `refl`.

The type of paths starting at `doc 0` is

```
Σ (n : Nat). doc 0 = doc n
```

This classifies pairs of a number and a path `doc 0 = doc n`, but the latter type is contractible, so it contains no additional information. Therefore, this type is isomorphic to `Nat`—that is, patches applicable to `doc 0` are characterized precisely by the index `n` of their codomain context `doc n`. This makes sense because there is only one path `doc 0 = doc n`.

A.3 Binary Trees

In analogy to Section 6, we will extend our running example to allow two distinct edits at each patch context, `add true` and `add false`. Again, we will index patch contexts by histories—in this case, lists of booleans indicating the sequence of patches applied to the repository. The edit `add x` thus prepends `x` to each history.

```
space R : Type where
  doc : Bool List →  R
  add : (x : Bool) {xs : Bool List} → doc xs = doc x::xs
```

Now the patch theory looks like a tree, where the nodes are histories and the paths label the edges. For example,

$$\begin{tikzcd}[column sep=small] & \texttt{doc []} \arrow[dl, "\texttt{add t}"'] \arrow[dr, "\texttt{add f}"] & \\ \texttt{doc [t]} & & \texttt{doc [f]} \end{tikzcd}$$

As before, inverses will not add new paths from the root `doc []`. The proof that `R` is contractible is basically the same as for `I*`.

```
toPath : (xs : Bool List) → doc [] = doc xs
toPath [] = refl
toPath (x::xs) = add x ○ toPath xs

is-contr : (x : R) → doc [] = x
is-contr (doc xs) = toPath xs
apd is-contr (toPath xs) = refl
  : PathOver (l. doc [] = l) (add x)
    (toPath xs) (toPath x::xs)
```

This `PathOver` is again true by `refl` and the definition of `toPath`.

Again, it follows that paths from `doc []` are characterized by complete histories, in this case `Bool Lists`. That is,

```
Σ (xs : Bool List). doc [] = doc xs
```

is isomorphic to `Bool List`. One direction of this isomorphism is `toPath`; the other projects out the codomain context, sending `p : doc [] = doc xs` to `xs`. (It is possible to define this projection without access to the type indices, as we did in Section 6.3.) Call the latter function `log`.

Now we will define a merge operation for this patch theory, and prove that it satisfies the merge laws. The types of these laws will be somewhat intricate due to patch contexts—unlike in Section 4.2, we cannot just take two paths `num = num` to two paths `num = num`.

Moreover, we will restrict merge to take diverging patches based at `doc []`, to ensure the patches are sequences of `add`s, up to homotopy. As in Section 6, this move allows us to ignore inverse paths, which are not meaningful patches in this setting. Therefore, `merge` will take a span of patches based at `doc []`, and return a cospan reuniting them.

```
merge : {l1 l2 : Bool List}
        (doc [] = doc l1) → (doc [] = doc l2) →
        Σ(l : Bool List).
          (doc l1 = doc l) × (doc l2 = doc l)
```

$$\begin{tikzcd}[column sep=small] & \texttt{doc []} \arrow[dl, "\texttt{p1}"'] \arrow[dr, "\texttt{p2}"] & \\ \texttt{doc l1} \arrow[dr, dashed, "\texttt{q1}"'] & & \texttt{doc l2} \arrow[dl, dashed, "\texttt{q2}"] \\ & \texttt{doc l} & \end{tikzcd}$$

Because `l1` and `l2` characterize paths `doc [] = doc l1` and `doc [] = l2`, it suffices to merge pairs of `Bool Lists`. As in Section 6.4, say we have a function `mergeH` which does this.

```
mergeH : Bool List → Bool List → Bool List
```

We can then define `merge` in terms of `mergeH`.

```
merge p1 p2 =
  let l = mergeH (log p1) (log p2)
  in (l, ((toPath l) ○ !p1, (toPath l) ○ !p2))
```

As discussed in Section 3, we want to prove that `merge` produces commuting squares, and that it is symmetric:

```
reconcile : {l1 l2 l : Bool List}
    (p1 : doc [] = doc l1) → (p2 : doc [] = doc l2)
  → (q1 : doc l1 = doc l) → (q2 : doc l2 = doc l)
  → merge p1 p2 = (l, (q1, q2))
  → q1 ○ p1 = q2 ○ p2
symmetric : {l1 l2 l : Bool List}
    (p1 : doc [] = doc l1) → (p2 : doc [] = doc l2)
  → (q1 : doc l1 = doc l) → (q2 : doc l2 = doc l)
  → merge (p1, p2) = (l, (q1, q2))
  → merge (p2, p1) = (l, (q2, q1))
```

The `reconcile` law is in fact enforced by the type of `merge`, which says that the span and cospan form a square; and the contractibility of `R`, which says that all squares commute. Thus the paths `q1 ○ p1` and `q2 ○ p2` are equal elements of `doc [] = doc l`.

The `symmetric` law reduces to the symmetry of `mergeH`. Given a proof

```
symmetricH : (l1 l2 : Bool List)
           → mergeH l1 l2 = mergeH l2 l1
```

that `mergeH` is symmetric, we can prove `symmetric` for `merge`—the first components of the two merges are equal by `symmetricH`, and the second components, the pairs of paths, are swapped by construction, as they only depend on `l`, `p1`, and `p2`.

A.4 Quotiented Binary Trees

To complete the analogy to Section 6, we will extend this example once more, this time with patch laws. For simplicity, we will say that any two patches commute.

In the binary tree patch theory, `Bool Lists` were an inductive characterization of paths from the initial repository, as those paths were compositions of `add true` and `add false`. Here, since patch composition commutes, we must quotient `Bool Lists` by permutation.

This yields the type of boolean multisets, lists quotiented by "`Ex`"change of adjacent elements, defined as the following quotient higher inductive type:

```
space MS : Type where
  [] : MS
  _::_ : Bool → MS → MS
  Ex : (x y : Bool) (xs : MS) → x::(y::xs) = y::(x::xs)
```

We index patch contexts by `MS`es, rather than `Bool Lists`. As before, patches prepend a boolean to the context.

```
space R : Type where
  doc : MS → R
  add : (x : Bool) {xs : MS} → doc xs = doc x::xs
  ex  : (x y : Bool) (xs : Bool List)
      → PathOver (s. doc xs = doc s) (Ex x y xs)
          (add x ○ add y) (add y ○ add x)
```

The `ex` constructor implements the patch laws, for example, equating `add true ○ add false` and `add false ○ add true`. It creates paths-over-paths because the equated patches have different types:

```
add true ○ add false : doc xs = doc true::(false::xs)
add false ○ add true : doc xs = doc false::(true::xs)
```

However, `true::(false::xs)` and `false::(true::xs)` are equal as multisets, by virtue of `Ex true false xs`. Thus, the patch law `ex true false xs` equates compositions of patches over the fact that their right endpoints are equal.

Now we will show `R` is contractible. Patch laws significantly complicate the proof; in fact, one might even expect `R` to have non-trivial loops:

$$
\begin{tikzcd}
 & \texttt{doc []} \arrow[dl, "\texttt{add x}"'] \arrow[dr, "\texttt{add y}"] & \\
\texttt{doc [x]} \arrow[d, "\texttt{add y}"'] & & \texttt{doc [y]} \arrow[d, "\texttt{add x}"] \\
\texttt{doc [x,y]} \arrow[rr, equal] & & \texttt{doc [y,x]}
\end{tikzcd}
$$

Equations in `MS` (the bottom edge of the diagram) together with inverses in `R` seemingly allow us to construct non-trivial loops `doc [] = doc []` in `R`. However, the patch law `ex x y []` trivializes these loops by equating the two sides.

To prove `R` is contractible, we first construct a path `doc [] = doc s` for each multiset `s`. Because `MS` is a quotient type, we must show that this map respects the quotient.

```
toPath : (xs : MS) → doc [] = doc xs
toPath [] = refl
toPath (x::xs) = add x ∘ toPath xs
apd toPath (Ex x y xs) = GOAL0
  : PathOver (s. doc [] = doc s) (Ex x y xs)
    (toPath (x::y::xs)) (toPath (y::x::xs))
```

`MS`-induction automatically demands this of us, in the form of the third clause. Expanding the definition of `toPath`, `GOAL0` has type

```
PathOver (s. doc [] = doc s) (Ex x y xs)
  (add x ∘ (add y ∘ toPath xs))
  (add y ∘ (add x ∘ toPath xs))
```

By path-over-a-path rules, this reduces to

```
ap doc (Ex x y xs) ∘ (add x ∘ (add y ∘ toPath xs))
  = (add y ∘ (add x ∘ toPath xs))
```

We can reassociate these compositions and cancel the two instances of `toPath xs`, yielding

```
ap doc (Ex x y xs) ∘ (add x ∘ add y) = (add y ∘ add x)
```

But this is exactly the type of `ex x y xs`, once we expand the `PathOver`. This completes our definition of `toPath`.

Morally, this subgoal—that `toPath` respects the path constructor of `MS`—verifies that `ex` fills in the loops we discussed above. Indeed,

```
apd toPath : {xs ys : MS} (p : xs = ys)
 → PathOver (h. doc [] = doc h) p (toPath xs) (toPath ys)
```

is a proof that, whenever `xs` and `ys` are equal multisets, then there is a disc whose boundary is formed by `toPath xs`, `doc xs = doc ys`, and `toPath ys`.

Now that we have defined `toPath`, we can prove that `R` is contractible with center `doc []`:

```
is-contr : (r : R) → doc [] = r
is-contr (doc xs) = toPath xs
apd is-contr (add x xs) = refl
  : PathOver (s. doc [] = s) (add x xs)
      (toPath xs) (toPath x::xs)
apd' (apd is-contr) (ex x y xs) = GOAL0
```

As before, the second clause is true by `refl`, using path-over-a-path reductions and the definition of `toPath`. The subgoal in the third clause is rather complicated as it involves paths over paths-over-paths; we have a machine-checked proof of this[7] but will not discuss it further here.

[7] https://github.com/dlicata335/hott-agda/blob/homotopical-patch-theory-paper/programming/PatchWithHistories.agda

A.5 Merge Laws For Richer Contexts

We would like to end by inductively characterizing the paths in the patch theory of Section 6, and using this characterization to prove the merge laws for the merge operation given in Section 6.4.

The first step will again be to prove that this `R` is contractible. The proof is essentially the same as in the previous subsection; the only difference is that this patch theory has more complex patches and patch laws. Again, we have a machine-checked proof[8] that (a generalized form of) `R` is contractible.

Thus, let us start with a proof that

```
is-contr {n : Nat} (h : History 0 n) → doc [] = doc h
```

By `apd is-contr`, identity types in `R` are also contractible, so paths in `R` are uniquely determined by their endpoints. Thus, the type of paths starting at `doc []`,

```
Σ (n : Nat). Σ (h : History 0 n). doc [] = doc h
```

is equivalent to

```
Σ (n : Nat). History 0 n
```

because the path provides no additional information.

By univalence, all constructions respect equivalence of types; therefore, a merge operation on complete histories suffices to merge paths, and a proof of the merge laws for the former suffices for the latter. In Section 6.4, we manually constructed merge on paths from merge on histories (i.e., without univalence), so here we will prove the merge laws manually as well. Even so, the contractibility of `R` is an important ingredient.

In Section 6 we defined merge on paths as

```
merge : {n1 n2 : Nat}
        {h1 : History 0 n1} {h2 : History 0 n2}
        (doc [] = doc h1) → (doc [] = doc h2) →
        Σ(n' : Nat). Σ(h' : History 0 n').
          (doc h1 = doc h') × (doc h2 = doc h')

merge p1 p2 =
  let (n',(h',(e1,e2))) =
    mergeH (interpH p1 []) (interpH p2 [])
    in (n', (h', (extToPath e1, extToPath e2)))
```

given a merge

```
mergeH : {n1 n2 : Nat}
         (h1 : History 0 n1) (h2 : History 0 n2) →
         Σ(n' : Nat). Σ(h' : History 0 n').
           Extension h1 h' × Extension h2 h'
```

on `Extensions` (complete histories with a particular prefix).

In this patch theory, the merge laws are:

```
reconcile : {n n1 n2 : Nat} {h : History 0 n}
    {h1 : History 0 n1} {h2 : History 0 n2}
  → (p1 : doc [] = doc h1) → (p2 : doc [] = doc h2)
  → (q1 : doc h1 = doc h) → (q2 : doc h2 = doc h)
  → merge p1 p2 = (n, (h, (q1, q2)))
  → q1 ∘ p1 = q2 ∘ p2

symmetric : {n n1 n2 : Nat} {h : History 0 n}
    {h1 : History 0 n1} {h2 : History 0 n2}
  → (p1 : doc [] = doc h1) → (p2 : doc [] = doc h2)
  → (q1 : doc h1 = doc h) → (q2 : doc h2 = doc h)
  → merge p1 p2 = (n, (h, (q1, q2)))
  → merge p2 p1 = (n, (h, (q2, q1)))
```

The `reconcile` law follows from the contractibility of `R`: the type of `merge` specifies that `p1`, `p2`, `q1`, and `q2` form a square, and by contractibility, all squares in `R` commute.

[8] https://github.com/dlicata335/hott-agda/blob/homotopical-patch-theory-paper/programming/PatchWithHistories2.agda

The `symmetric` law follows from the symmetry of `mergeH`. Assume a proof:

```
symmetricH : {n n1 n2 : Nat} {h : History 0 n}
  → (h1 : History 0 n1) (h2 : History 0 n2)
  → {e1 : Extension h1 h} {e2 : Extension h2 h'}
  → mergeH h1 h2 = (n, (h, (e1, e2)))
  → mergeH h2 h1 = (n, (h, (e2, e1)))
```

The first two components of the two `merge`s are equal because the same is true of `mergeH`; the last two components, a pair of paths, are swapped because they depend only on the last two components of the corresponding `mergeH`s, which `symmetricH` ensures are also swapped.