JFP **26**, e18, 45 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000198

*Homotopical patch theory**

CARLO ANGIULI and EDWARD MOREHOUSE
Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
(*e-mails:* cangiuli@cs.cmu.edu, edmo@cs.cmu.edu)

DANIEL R. LICATA
Department of Mathematics and Computer Science, Wesleyan University, Middletown, Connecticut, USA
(*e-mail:* dlicata@wesleyan.edu)

ROBERT HARPER
Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
(*e-mail:* 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 is 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.

1 Introduction

Martin-Löf's intensional type theory (MLTT) and its descendants are the basis of proof assistants such as Agda (Norell, 2007) and Coq (Coq Development Team, 2015). Homotopy type theory is an extension of MLTT based on a correspondence with homotopy theory and higher category theory (Hofmann & Streicher, 1998;

* 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. This material is based on research sponsored in part by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government or Carnegie Mellon University.

Voevodsky, 2006; Gambino & Garner, 2008; Warren, 2008; Awodey & Warren, 2009; Garner, 2009; Lumsdaine, 2009; van den Berg & Garner, 2011; Kapulkin *et al.*, 2012). 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 homotopy 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 $p =_{a=_A b} q$. Moreover, one can define all the path operations considered in homotopy theory, including identity paths $refl : a = a$ (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, $refl \circ p = p$), homotopies relating those homotopies, and so forth.

This correspondence has suggested several extensions to type theory. One is Voevodsky's *univalence axiom* (Voevodsky, 2006; Kapulkin *et al.*, 2012), which describes the path structure of the type universe (the type of small types). Another is *higher inductive types* (Lumsdaine, 2011; Shulman, 2011; Lumsdaine & Shulman, 2013), a new class of datatypes specified by constructors not only for points but also for paths. Higher inductive types were originally introduced to permit the type-theoretic definition of basic topological spaces such as circles and spheres, and have had significant applications in a line of work on using homotopy type theory to write computer-checked proofs of theorems from homotopy theory (Licata & Brunerie, 2013; Licata & Shulman, 2013; Univalent Foundations Program, 2013; Hou, 2014; Licata & Finster, 2014; Cavallo, 2015; Licata & Brunerie, 2015).

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 (Licata & Harper, 2012; Bezem *et al.*, 2014; Shulman, 2015; Altenkirch & Kaposi, 2015; Barras *et al.*, 2015; Polonsky, 2015; Cohen *et al.*, 2016). 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 & Harper (2011) apply ideas related to homotopy type theory to modeling variable binding. Altenkirch (2014) shows that containers (Abbott *et al.*, 2005) 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 applications are less developed than the mathematical applications.

In this paper, we present an example of using homotopy type theory to model *patch theory* (Jacobson, 2009; Houston, 2012; Mimram & Di Giusto, 2013), the abstract study of version control systems. Intuitively, a patch is a formal representation of a change to a repository. A patch (for example, "delete file f") applies to a class of repositories (those in which the file f exists), and results in another class of repositories (those in which the file f no longer exists). Such classifications of repositories are called *patch contexts*, and serve as types for patches. Patches are closed under identity (a no-op), composition (sequencing), and inverses (undo). In addition, patches are subject to equations called *patch laws*, which address both general (e.g., composition is associative) and domain-specific considerations (e.g., that the order of edits to independent lines of a file can be swapped).

Then, a patch theory[1] is a collection of such patch contexts, patches, and patch laws, which together abstractly characterize a language of patches. Any correct implementation of those patches must contain a set of repositories for each patch context, functions between those repositories for each patch, and equations between those functions for each patch law.

Our representation of patch theory is inspired by *functorial semantics* in the sense of Lawvere (1963)—in which the axioms of an algebraic theory are represented as a category, and any instance of that algebraic theory is precisely a structure-preserving functor out of that category. Using homotopy type theory, we will represent patch theories as higher inductive types whose points are patch contexts, whose paths are patches, and whose paths between paths are patch laws; and interpretations of a patch theory as functions out of its higher inductive type. Because functions in homotopy type theory always respect path structure, this guarantees that interpretations are sound for their patch theory, and in particular, that interpretations respect patch laws.

We will consider interpretations such as patch interpreters (sending patches to functions on repositories), patch optimizers (consolidating a sequence of patches into a more direct, equivalent sequence), and patch histories (maintaining a list of the patches themselves). That such functions—and others, including merging—are definable underscores the fact that paths in homotopy type theory are proof-relevant, i.e., that we can distinguish, manipulate, and extract computational content from them, unlike ordinary notions of equality.

Our work shows how to apply standard concepts from homotopy theory in a practical programming setting. For example, the first patch theory we discuss is in fact the circle. A key problem in homotopy theory is to algebraically characterize the paths in a space using what is called a homotopy group; similarly, sometimes we characterize an identity type (namely, the patches of a patch theory) using a derived induction principle, in order to define operations on those paths (such as merging). We hope that this paper will make homotopy type theory more accessible to the functional programming community, so that programmers can begin to consider its applications.

Homotopy type theory is still under development, and one of our goals in this paper is to guide future work on it by providing an extended programming application. We use an informal Agda-like concrete syntax, including datatype and pattern-matching syntax for higher inductive types, and marking implicit arguments with braces $\{-\}$. (This is similar to the informal type theory employed in the book *Homotopy type theory* (Univalent Foundations Program, 2013), but with a

[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.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

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.

Because a computational interpretation of homotopy type theory is work in progress, there is no complete operational semantics that can evaluate the programs in this paper. However, we will use a notion of computation-up-to-paths—based on existing work on this topic (Licata & Harper, 2012; Shulman, 2015; Altenkirch & Kaposi, 2015; Cohen *et al.*, 2016)—in order to compute with the programs we define in this paper.

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 through 8, we discuss successively more complex patch theories.

Section 4 is the simplest case: a patch theory with a single patch context and no patch laws. In Section 5, we add patch laws. In Section 6, we consider a theory requiring multiple patch contexts, because not all patches are universally applicable. The theory in Section 7 has both patch laws and multiple patch contexts. Finally, in Section 8, we consider a patch theory of text files, requiring both patch laws and multiple patch contexts.

A preliminary version of this paper appeared in the *Proceedings of the 2014 International Conference on Functional Programming*. We have added two more patch theories (Sections 6 and 7) in order to clarify the concepts needed in Section 8, and discuss some results that were obtained after the final conference version was submitted.

2 Basics of homotopy type theory

In this section, we will review some basic definitions of homotopy type theory. Various formulations of homotopy type theory are currently in development; in this paper, we will use the standard version appearing in *Homotopy type theory* (Univalent Foundations Program, 2013), henceforth “the HoTT Book”, because we expect that any future versions of homotopy type theory will be able to interpret it.

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 β-like reductions expressing that elimination is post-inverse to introduction—for example, `(λx → e) e'` and `[e'/x]e` are definitionally equal. Uses of definitional equality are not marked in the proof term or program: if `e` has type `A`, then `e` also has any other type `A'` that is definitionally equal to `A`. 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 `A` and `p` is an element of the identity type `A = A'`, then `coe p e` has type `A'`.

In homotopy type theory, the identity type is specified by its introduction rule, called reflexivity, and elimination rule, known as path induction or J. Elements of the identity type behave like *paths in a space* or *morphisms in a groupoid*, in the sense that one can define a constant path `refl` (witnessing the reflexivity of equality), composition of paths `q ∘ p` (witnessing the transitivity of equality),[2] and path inversion `! p` (witnessing the symmetry of equality), among other operations. 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 = p
(r ∘ q) ∘ p = r ∘ (q ∘ p)
(! p) ∘ p = refl
p ∘ (! p) = refl
```

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

```
ap f : x = y → f x = f y
```

that takes paths `x` $=_{\mathtt{A}}$ `y` to paths `f x` $=_{\mathtt{B}}$ `f y`. Logically, this expresses that propositional equality is a congruence; homotopically, it expresses that any function has an **a**ction on **p**aths; and categorically, it expresses that functions are *functors*, preserving the path structure of types. The function `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)
```

It is useful to characterize types based on how far "up" their path structure extends. A type `A` is a *set* iff any two parallel paths in `A` are equal—i.e., 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-groupoid* iff any two paths between parallel paths are equal. A type is a *mere proposition* iff any two elements are equal. A type is *contractible* iff it is a mere proposition and moreover it has an element, that is, it has a unique element up to homotopy.

2.2 Univalence

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

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

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

[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.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

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 parameterized 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 ($\mathtt{refl}_b$), inverse ($!_b$), and composition ($\_\circ_b\_$) of bijections directly (focusing on the underlying functions, and where `f2 . f1` is `(λx → f2(f1(x)))`):

```
refl_b : Bijection A A
refl_b = ((λx → x), (λx → 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:

```
refl = ua refl_b
! (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. The function `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 `coeBiject 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 coercing 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 coerced along.

2.3 Paths over paths

Both simply and dependently typed functions preserve path structure, but expressing this fact for the latter requires some additional machinery. If we have a family of types `B : A → Type`, a dependently typed function `f : (x : A) → B(x)`, and a path $\texttt{p : x} =_{\texttt{A}} \texttt{y}$, then for `f` to preserve `p` means that `f x : B(x)` and `f y : B(y)` are equal. But they do not even have the same type!

Luckily, these types are equated by `ap B p : B(x) = B(y)`, because `B` is itself a path-preserving function. So we can express the equality of `f x` and `f y` as a path in `B(y)` by coercing `f x` along `ap B p`:

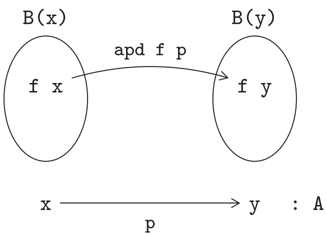
```
coe (ap B p) (f x) = f y
```

or symmetrically, as a path in `B(x)`:

```
f x = coe (! (ap B p)) (f y)
```

We hide this choice behind an interface by defining the type `PathOver B p b1 b2` of *heterogeneous equalities* (McBride, 2000), or *paths over paths*, which classifies paths in the type family `B` between `b1 : B(a1)` and `b2 : B(a2)` correlated by a path `p : a1 = a2`. Then `apd`, the action on paths of dependent functions, has type

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

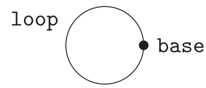


In this paper, we will occasionally invoke lemmas characterizing `PathOver`s in `B` for certain `B`. For example, if `B` is a constant family `λx → C`, then `PathOver B p b1 b2` is equivalent to the type $\texttt{b1} =_{\texttt{C}} \texttt{b2}$. (So when `f` is not dependent, `ap f` and `apd f` have the same type, modulo this equivalence.) We refer to these lemmas as "simplifications" because they are type-driven in a straightforward way; see Chapter 2 of the HoTT Book (2013) for proofs of related results.

2.4 Higher inductive types

Ordinary inductive types are specified by *generators*; for example, the natural numbers are generated by zero and successor: `zero : Nat` and `succ : Nat → Nat`.

Higher dimensional inductive types (or just *higher inductive types*) (Lumsdaine, 2011; Shulman, 2011; Lumsdaine & Shulman, 2013) 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. We define the circle as a higher inductive type with two generators:

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

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

2.4.1 *Circle recursion*

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

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

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

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

```
f base = b'
f loop = l'     -- does not typecheck!
```

The second equation does not quite make sense, because `f` is a function

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

`Circle → X` but `loop` is a *path* in the circle. Therefore, we use `ap` (discussed above) to denote `f`'s action on paths:

```
sap 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’`.

As a first example, we write a function to "reverse" a path in 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 in 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 propositional equality. Current models and implementations justify only the latter, so we will take it to be a propositional equality.

While primitive recursion suffices to define functions `Nat → X`, defining a dependently typed function `(n : Nat) → C(n)` requires natural number *induction*, i.e., specifying `c0 : C(zero)` and `c1 : (n : Nat) → C(n) → C(succ n)`. Analogously, circle induction states that one can define a function `f : (x : Circle) → C(x)` by specifying

```
f base = b’ : C(base)
apd f loop = l’ : PathOver C loop b’ b’
```

We refer the reader to Licata & Shulman (2013), Univalent Foundations Program (2013) for topological intuition.

2.5 Computation

Although MLTT has been used as the basis for dependently typed programming languages (Nordström *et al.*, 1990; Norell, 2007), MLTT itself only defines typing

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

and definitional equality judgments, and no operational semantics. Formally, the purpose of definitional equality is only to give terms more types: if types `A,B` are definitionally equal, then any terms of type `A` also have type `B`. For example, `refl : 1 + 1 = 2` because `refl : 2 = 2` and `1 + 1` is definitionally equal to 2. For this reason, all definitional equalities are also propositional by `refl`.

MLTT admits a *computational interpretation* in the sense that two open terms are definitionally equal exactly when their β-normal forms are equal; and that for closed terms of type `bool`, head reduction suffices and always results in either `true` or `false`. One can therefore think of closed terms as programs, head reduction as their operational semantics, and `bool` as an observable type. (A related way to extract computational meaning from proofs is to interpret the proof rules as closed λ-terms, a technique known as realizability (Kleene, 1945; Kreisel, 1959). Aczel (1977) has constructed realizability interpretations of MLTT; doing the same for homotopy type theory is an open problem.)

Homotopy type theory, as defined in the HoTT Book (2013), extends the typing judgment with univalence and higher inductive types, but does not add any definitional equalities involving non-`refl` paths. This breaks canonicity—the property that all closed terms in `bool` are definitionally equal to either `true` or `false`—by introducing terms like `stuck`:

```
notb  = (not, not, ...)     : Bijection Bool Bool  -- swaps true and false
stuck = coe (ua notb) true : Bool
```

which are propositionally, but not definitionally, equal to `false`. It also breaks MLTT's computational interpretation, because `stuck` is head-normal but neither `true` nor `false`.

It is conjectured that one can restore canonicity and the computational interpretation by adding more definitional equalities; doing so is an active area of research (Licata & Harper, 2012; Altenkirch & Kaposi, 2015; Cohen *et al.*, 2016). (Current attempts change how the identity type is axiomatized, in order to simplify its definitional equalities.) But this raises the question: *which* propositional equalities should be made definitional? We certainly cannot make *all* propositional equalities definitional, because the former are proof-relevant (programs can distinguish them) while the latter are not. As a concrete example, `A * B` and `B * A` are propositionally equal types by univalence, but if we made them definitionally equal, then any term of type `A * B` would also have type `B * A`.

However, there are many particular propositional equalities which we (and others) conjecture *are* computation steps, like `coe (ua (f,g,p,q)) x = f x` (which would fix `stuck`). A traditional computational interpretation would require such equations to be definitional. However, we believe it is possible to describe these equalities as computational even when some of them remain propositional—in plain MLTT, not all definitional equalities are head reductions; in current homotopy type theory, not all computation steps are definitional equalities. In our setting, we run programs by giving a sequence of computational propositional equalities. For example, we calculate

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

where the final two steps are propositional but not definitional equalities.

3 Patch theory in homotopy type theory

Patch theory is the abstract study of version control systems by considering how their patches behave under operations such as composing, reverting, and merging. Patch theory allows us to separate the purely algebraic aspects of a version control system (which patches exist, and which equations they satisfy) 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 of it 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.

Others have employed a variety of mathematical formalisms to represent patch theories, including separation logic (Swierstra & Löh, 2014), category theory (Houston, 2012; Mimram & Di Giusto, 2013), and the language of inverse semigroups (Jacobson, 2009). In this paper, we formulate patch theory in homotopy type theory.

3.1 Patch theories as higher inductive types

In this paper, we represent patch theories as higher inductive types. The patch contexts of a patch theory are represented as points of the corresponding type. Patches are represented as paths between their domain and codomain patch contexts. Patch laws are represented as paths between patches, or homotopies.

Representing patches as paths means that we automatically get a `refl` patch for every patch context, a composite patch `q ∘ p` for any composable patches `p,q`, and an inverse patch `(! p)` for any patch `p`. We use these paths to model the constant patches, composite patches, and inverse patches, respectively, that we expect to exist in every patch theory. Representing patch laws as homotopies means that the groupoid laws for paths (associativity of composition, etc.) automatically hold for these patch operations.

Notice that these inverse patches are two-sided inverses. That `(! p)` is a post-inverse to `p` means that applying a patch and then its inverse is the same as no change; however, that it is a pre-inverse means that we can apply the inverse of a patch *before* the patch itself, also to no effect. We will explore this point in greater detail later.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

3.2 Interpretations of patch theories

A patch theory is a formal specification of patch contexts, patches, and patch laws; a version control system, however, consists of concrete repositories and functions between them. We bridge this gap by ensuring that a version control system faithfully implements its specification, in the sense that we can map each patch context to the collection of repositories it classifies, each patch to a function that updates those repositories appropriately, and each patch law to an equation between those functions. In the terminology of functorial semantics, such a mapping is an *interpretation* or *model* of the patch theory.

In this paper, we represent interpretations of a patch theory as functions out of its higher inductive type `R`, or out of `R`'s identity types. For example, a version control system is a function `I : R → Type`. We define such an `I` using `R`-recursion, which (following Section 2.4.1) means we give a type for each patch context of `R`, a path between types (which is, by the univalence axiom, a bijection between types) for each generating path, and a homotopy between those paths for each generating patch law.

This `I`, like all functions definable in homotopy type theory, preserves the path structure of its domain, so if we prove a theorem about the patch theory, we can send it to a theorem about its interpretation. This is useful because a patch theory may have many interpretations. Other kinds of interpretations we consider are *patch optimizers*, which interpret patches as simpler but equivalent compositions of patches, and *patch histories*, which interpret patches as concrete changelogs, rather than the changes themselves.

Unfortunately, not all seemingly reasonable interpretations are actually functorial. Suppose we wanted a function `countPatches` which takes every (composite) patch to the number of primitive (generating) patches it contains—then for primitive `p`, `countPatches (!p ∘ p) = 2`, and `countPatches refl = 0`. But `(!p ∘ p) = refl`, so `countPatches` would not respect patch laws, and is therefore not definable! We will see how functoriality complicates the definitions of patch histories in Sections 7 and 8.

3.3 Merging

For our purposes, merging is an operation on a patch theory 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

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

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

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

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

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

because the order of two patches should not affect how to reconcile them.

It is always possible to define a total merge function, since for any span we may give $merge(f_1, f_2) = (!f_1, !f_2)$, 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.

In the remainder of this paper, we will consider representations, interpretations, and merge functions for a number of patch theories.

4 An elementary patch theory

First, we define a very simple patch theory, to illustrate our 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 identity, inverse, and composite paths always exist, 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

In ordinary dependent type theory, freely defining this patch theory would require 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 patch laws, one would need to impose the group laws on this type; this could be done using a quotient type (Constable *et al.*, 1986) 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 using homotopy type theory and modeling patches as paths, however, the patch theory automatically includes identity, inverses, composition, and the group laws. □

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. We would first define:

```
interp : Patch → Bijection Int Int
interp add1 = successor
interp id = refl_b
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 `(λx → x+1, λx → 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 five cases for the five 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 2.4.1. 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 = coeBiject (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
= coeBiject (ap I add1)     [definition]
= coeBiject (ua successor) [ap I on add1]
= successor                 [coe on ua successor]
```

using the simplification 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)
= coeBiject (ap I (q ∘ p))
= coeBiject (ap I q ∘ ap I p)   [ap on ∘]
= (coeBiject (ap I q)) ∘b (coeBiject (ap I p))
= interp q ∘b interp p
```

`interp refl = refl`$_b$ 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 interpretations of a patch theory. We needed to give only the key case for `add1`—the semantics of the basic patches is automatically lifted to patch operations, not manually as in Remark 4.2. 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 Licata & Shulman (2013) and the HoTT

[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`.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

Book (2013). The function `interp p` adds to its input what is called the *winding number* of a path `p` in 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 cancelled.

Note that, although we were thinking of `num` as an integer and `add1` as successor, 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 not_b
```

where `not_b : 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 interpretation satisfies additional equations not demanded by the patch theory, such as

```
ap I' add1 ∘ ap I' add1 = ua (not_b ∘_b not_b) = refl
```

This equation does not hold in our original interpretation `I`, because incrementing an integer is not self-inverse. In fact, 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 the integers.

4.2 Merge

Next, we implement a merge operation, which satisfies the laws discussed in Section 3. Writing `Patch` for `num = num`, 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)
```

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`—all loops on the circle commute. It is not immediately obvious how to do this, because homotopy type theory does not provide a direct induction principle for loops. That is, there is no built-in elimination rule that allows one to, for example, analyze `f1` as either `add1`, or the identity, or an inverse, or a composition, because such a case-analysis would need to respect all paths between loops, which differ from type to type.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

Instead, we must prove a *derived induction principle* for the type `num = num` from the induction principle for `R`—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 (Licata & Brunerie, 2013; Licata & Shulman, 2013; Univalent Foundations Program, 2013; Licata & Finster, 2014), which can be applied here.

In this particular case, we already know that the fundamental group of the circle is the integers. That is, the type `num = num` is in bijection with `Int`, and so the integers give canonical representatives (`add n`, for `n : Int`) for equivalence classes of patches in this patch theory, considered modulo the group laws. We establish that bijection by giving functions `winding` and `repeat` that compose to the identity. The function `winding : num = num → Int` is 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 : Nat`) representing $n + 1$, and `- n` (where `n : Nat`) representing $-(n + 1)$.

```
repeat 0 = refl
repeat (+ n) = add1 ∘ add1 ∘ ... ∘ add1 [n+1 times]
repeat (- n) = !add1 ∘ !add1 ∘ ... ∘ !add1 [n+1 times]
```

In fact, `winding` and `repeat` are also group homomorphisms, e.g., `repeat (x + y) = repeat x ∘ repeat y`. The proof that these functions are mutually inverse is described in Licata & Shulman (2013) and the HoTT Book (2013), which contain the full proof that the fundamental group of the circle is the integers.

The bijection between `num = num` and `Int` induces a derived induction principle: since any patch is equal to `repeat n` for some `n`, in order to prove `P : num = num → Type` for all paths, it suffices to prove `P(repeat n)` for all integers `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 patch theory, the correctness of merge follows from the fact that the fundamental group of the circle is the integers—our first example of a software correctness proof being a corollary of a theorem in homotopy theory!

One further point to note is that here we were able to *define* merge without converting paths to integers, but to prove the reconciliation property we needed to reason inductively, using canonical representatives of equivalence classes of paths. This is because all patches commute, so we can define `merge(x, y) = (y, x)` without analyzing the structure of `x` and `y`. In Sections 7 and 8, we will need to analyze the structure of patches in order to even define merging. We end this section by showing an alternate definition of `merge`, which analyzes its input patches in that way.

```
merge' (p, q) =
  let (a, b) = mergeI(winding p, winding 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` to integers. Paths that are equal according to the group laws are necessarily sent to equal representatives; for example, both `(add1 ∘ add1) ∘ add1` and `add1 ∘ (add1 ∘ add1)` are sent to `3`. We may then compose this choice of representatives with any function on integers, and the result will be guaranteed to respect the group laws. Here, we use `mergeI` to recursively "merge" the two integers with cases such as the one given above, which 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. (In effect, it merges "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'` uses `repeat` to convert the resulting integers back to paths. One can prove by induction that `mergeI (x, y) = (y, x)`; and `winding` 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 patch theory with patch laws beyond the groupoid laws. 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 add patch laws stating that edits at independent lines commute, and that swapping `s` with `s` has no effect. We define two interpretations of this patch theory—the intended patch interpreter, and a simple patch optimizer; we do not consider merge in this section, because we discuss it for the more general language in Section 8.

5.1 Definition of patches

This patch theory is represented by the following higher inductive type, where `n : Nat` is fixed throughout this section:

```
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
```

The point constructor `doc` should be thought of as a document with `n` lines. 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).

This language also has non-trivial patch laws, which are represented by giving generators for *paths between paths*. 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 path and path-between-path constructors, 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 (s1 ↔ s2 @ i) = swap' s1 s2 i
β21 : PathOver (λ(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)
β22 : PathOver (λx → x = refl) β1
               (ap (ap f) (noop s i))
               (noop' s i)
```

The first computation rule is in fact a definitional equality, while the second is a path. The third and fourth computation rules are stated as `PathOver`s because their left- and right-hand sides are in different (although propositionally equal) types. For example, in the fourth computation rule, `ap (ap f) (noop s i)` has type `ap f (s ↔ s @ i) = ap f refl`, whereas `noop' s i` has type `swap' s s i = refl`. The right-hand sides match up because `ap f refl` is definitionally equal to `refl`, and the left-hand sides match up over the path $\beta 1$, the second computation rule.

Although we use pattern-matching notation for `R`-recursion, keep in mind that the types of the left-hand sides (e.g., `ap (ap f) (noop s i)`) are in the last two

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

cases only propositionally equal, via `PathOver` simplifications, to the types of the right-hand sides (e.g., `noop' s i`).

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 `PathOver` as the image of `indep`.
- A 2-dimensional `PathOver` as the image of `noop`.

We omit the details of the final two, which are not used below.

5.2 *Interpreter*

Our intended patch interpreter is a function

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

As before, we generalize this to an interpretation of the whole patch theory `R`, and define a function `I : R → Type` such that

```
interp p = coeBiject (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 neq) =
  GOAL5.1 : 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) = GOAL5.2 : 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 `GOAL5.1` and `GOAL5.2` represent *goals*, that is, terms that must still be provided before the program is complete.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

The image of indep and noop are the goals GOAL5.1 and GOAL5.2, with the types written out above—which ensure that the interpretation validates the patch laws. 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, GOAL5.2 is solved by observing that swapat (s,s) i is the identity bijection, and then using the fact that ua $\mathtt{refl}_b$ = refl. GOAL5.1 is solved by turning both sides into a composition of bijections using the fact that ua b2 ∘ ua b1 = ua (b2 $\circ_b$ 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 before, 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

We will also define an alternative interpretation of this theory, a patch optimizer, to illustrate a benefit of domain-specific patch laws:

```
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. Our goal is to optimize s ↔ s @ i to refl, saving ourselves two unnecessary string comparisons when the patch is applied.

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 we used when reversing the circle in Section 2.4.1. This is defined as follows:

```
opt1 : R → R
opt1 doc = doc
ap opt1 (s1 ↔ s2 @ i) = opt0 s1 s2 i
ap (ap opt1) (indep s t u v i j neq) =
  GOAL5.3 : opt0 s t i ∘ opt0 u v j
          = opt0 u v j ∘ opt0 s t i
```

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

```
ap (ap opt1) (noop s i) =
  GOAL5.4 : opt0 s s i = refl
```

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 `GOAL5.3` and `GOAL5.4` whose types are given above. The goal `GOAL5.4` is true because `String.equals s s` will be true, so, after case-analysis, `refl` proves that `opt1 s s i = refl`. The goal `GOAL5.3` requires case-analyzing both `String.equals s t` and `String.equals u v`. 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 ∘ u ↔ v @ j = u ↔ v @ 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:

```
opt1Correct : (x : R) → x = opt1 x
opt1Correct doc = refl
apd opt1Correct (s1 ↔ s2 @ i) =
  GOAL5.5 : PathOver (λx → x = opt1 x) (s1 ↔ s2 @ i) refl refl
apd (apd opt1Correct) (indep s t u v i j neq) = GOAL5.6
apd (apd opt1Correct) (noop s i) = GOAL5.7
```

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. By an argument we suppress, this `PathOver` type simplifies 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

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

where p and q are two patches, and f_1 and f_2 are two patch laws equating these two patches.

In homotopy type theory, equality of points can contain interesting information—after all, we are representing patches as equalities, or paths. Likewise, equality of equalities is not trivial—we can choose to have some patch laws but not others, as we have done in `R`. So there is no reason that equalities of equalities of equalities, like the equation we have above, must necessarily hold.

For example, `indep i≠j ∘ indep j≠i` and reflexivity are both patch laws between the patch `(s ↔ t @ i) ∘ (u ↔ v @ j)` and itself. (The former swaps the order of the patches twice.) But unless we add this equation to `R`, there is no proof that these patch laws are equal to each other; we could even equate certain patch laws but not others!

Truncation (see Chapter 7 of the HoTT Book (2013)) is a technique for trivializing *all* equations of a certain "height" in a type. In this case, we could truncate by adding the following constructor to `R`:

```
-- 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
```

The `trunc` constructor adds a path between any two parallel patch laws `f1` and `f2`. Another way to say this is that `trunc` forces `R` to be a 1-groupoid, because it ensures that any two paths between parallel paths are equal. As usual, each constructor places additional demands on all maps out of `R`; for `trunc`, it says that we can only define maps from `R` to other 1-groupoids.

Fortunately, that restriction would not prevent us from defining this patch optimizer—`opt1` maps into `R` (a 1-groupoid), and `opt1Correct` maps into an identity type of `R` (and the identity types of a 1-groupoid are 1-groupoids). Thus, truncating `R` would be an appropriate modification to make. (All the subsequent patch theories we consider will turn out to be 1-groupoids, without the need for truncation.)

Program and prove. An alternative, which requires neither truncation nor proving any equations between patch laws, is to simultaneously implement the optimizer and prove its correctness. Once again, we define

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

as the action on paths of a function on `R`. However, `optimize` cannot be the `ap` of any function, because `ap` takes simply typed functions to identity types, whereas the codomain of `optimize` is not an identity type, and depends on the input patch `p`. Instead, we will define a dependently typed function

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

and define `optimize` essentially as the `apd` of `opt`.

Recall that `apd` takes a type family `B : R → Type` and a function `f : (x : R) → B(x)` to a function `(p : x = y) → PathOver B p (f x) (f y)`. In this case, the type family is `λx → Σ(y : R). y = x`, and so

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

But this does not look like the type of `optimize p`!

When the family `B` is known, the type `PathOver B p b1 b2` can be simplified in a type-driven way to a propositionally equal one. In this case, `B(x)` is the Σ-type of a constant family `R` with an identity type `y = x`. According to the appropriate lemmas[6]:

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

[6] This is because a `PathOver` in a Σ-type is a pair of `PathOver`s in each component (the second over the first), because a `PathOver` in a constant family `λx → R` is just a path `q` in `R`, and because a `PathOver` in an identity type is a square in the underlying type `R`—specifically, `PathOver (λ(x,y) → y = x)`

Comparing the left-hand side of `simpl` to the type of `apd opt p`, notice that if `opt doc = (doc,refl)`, then

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

and so coercing this along `simpl` will get us the type we wanted:

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

The upshot is that we can define `optimize` once we have defined an `opt` such that `opt doc = (doc,refl)`. We do this using `R`-induction as follows:

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

In the second clause, we need a term of type

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

We obtain one by coercing along `(! simpl)` a term of type

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

We choose a term implementing our optimization—replace the input patch `s1 ↔ s2 @ i` with `refl` when the strings are equal, and leave it unchanged otherwise—and pairing each output with a proof that it is equal to the input `s1 ↔ s2 @ i`.

For each of the `noop` and `indep` cases, we need to give a homotopy between two specific paths between two specific points in the type `Σy:R. y = x` (for some `x`). However, the type `Σy:R. y = x` is in fact *contractible*—it is equivalent to `unit`, because any pair `(y, p)` can be transformed into `(x, refl)` by coercing `y` to `x` along `p` (see Lemma 3.11.8 of the HoTT Book (2013)). The identity types of any contractible type are mere propositions, so any two paths are connected by a homotopy. Thus, we can complete the `noop` and `indep` cases simply by appealing to these facts and the contractibility of `Σy:R. y = x`.

Singleton Types and Computation. The type `Σy:A. x = y` is traditionally called a *singleton type*, written `S(x)`, because it consists of those points in `A` which are equal to `x` (along with a proof that `x = y`). One may well wonder what is the point of writing a function into a singleton type:

`(p,q) refl refl` is a square

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

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 `PathOver`).

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

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

when all the elements of `S(p)` are equal? Isn't this just a triviality, because it must be the identity function?

The answer is *no*, because the point `y` and the path `x = y` can both contain meaningful computational content. Consider defining various sorting algorithms in plain MLTT. We can express the correctness of a sorting algorithm by comparing it to a reference solution:

```
bubblesort : Nat List → Nat List
quicksort : Nat List → Nat List
quicksortCorrect : (xs : Nat List) → bubblesort xs = quicksort xs

qs : (xs : Nat List) → S(bubblesort xs)
qs xs = (quicksort xs, quicksortCorrect xs)
```

Since all sorting algorithms are extensionally equal, they all have type `(xs : Nat List) → S(bubblesort xs)`. Indeed, there is no way inside MLTT to distinguish extensionally equal functions—there is no predicate satisfied by one but failed by the other. Yet, we consider it useful to define `quicksort`, because it computes in a different way than `bubblesort`, and `quicksortCorrect` is of mathematical interest even though it returns `refl` for every `xs`.

Likewise, even though `optimize` is equal to the identity function—as is every function of type `(a : A) → S(a)`—we expect, based on work on the computational interpretation of homotopy type theory, that it will compute differently. That is, `optimize (s ↔ s @ i)` will evaluate to `refl` and not `s ↔ s @ i`, even though these paths are homotopic by `noop s s i`. That homotopy is a prime example of a non-computational propositional equality, as we discussed in Section 2.5.

6 A patch theory with multiple contexts

The patch theories of Sections 4 and 5 only had one patch context each, because their patches were all applicable to every repository state. Realistic patch theories do not share this property—for example, deleting a line requires a file to be non-empty. In this section, we develop a very simple patch theory requiring multiple patch contexts, for a natural number repository that can be incremented or decremented.

In Section 4, compositions of the single path constructor `add1 : num = num` and its inverse allow us to add or subtract arbitrary numbers from the integer repository. In a natural number repository, we cannot subtract a number larger than the current contents. Our solution is to maintain a lower bound on the number in the repository. We define `R` as follows:

```
space R : Type where
  -- point constructor (patch context):
  doc : Nat → R
  -- path constructor (basic patch):
  add1 : (n : Nat) → doc n = doc n+1
```

`R` has `Nat`-indexed contexts, and a patch from `doc n` to `doc n+1` for each `n`. Whereas `doc` in Section 4 classified all repositories, here `doc n` classifies those

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

repositories whose contents are *at least* n, and thus, can safely be decremented n times.

How does this solve the problem? Ignoring compositions of patches for the moment, we need only rule out applying the patch `! (add1 n)` to a repository containing `0`. But `add1 n` is a patch from `doc n` to `doc n+1`, so its inverse is a patch from `doc n+1` to `doc n`, and any repository whose contents are at least `n+1` for some `n` cannot contain `0`. In general, any patch whose behavior is to subtract `m` from a repository must be a patch from `doc n+m` to `doc n`, and so it cannot apply to any repository whose contents are less than `m`.

6.1 Interpreter

As before, we want to define an `interp` function implementing patches—terms of type `doc n = doc m`—as bijections between the types implementing the patch contexts `doc n` and `doc m`. (In Sections 4 and 5, we only had one patch context, so a single type implemented all repositories.)

Following the intuition developed above, we interpret `doc n` as the type of natural numbers which are `AtLeast n`; that is, numbers `m` paired with a proof that `n ⩽ m`.

```
data _⩽_ : Nat → Nat → Type where
  z⩽ : (n : Nat) → 0 ⩽ n
  s⩽ : {n m : Nat} → n ⩽ m → n+1 ⩽ m+1

AtLeast : Nat → Type
AtLeast n = Σ (m : Nat). n ⩽ m
```

We then interpret `add1 n : doc n = doc n+1` as a function `AtLeast n → AtLeast n+1` sending `m` (such that `n ⩽ m`) to `m+1` (which thus satisfies `n+1 ⩽ m+1`).

```
increment : (n : Nat) → Bijection (AtLeast n) (AtLeast n+1)
          increment n = (λ(m,pf) → (m+1, s⩽ pf), ...)
```

The fact that this function is a bijection allows us to define `I : R → Type` as in the previous sections, and therefore `interp` as well:

```
I : R → Type
I (doc n) = AtLeast n
ap I (add1 n) = ua (increment n)

interp : {n m : Nat} → (doc n = doc m) → Bijection (AtLeast n) (AtLeast m)
interp p = coeBiject (ap I p)
```

Notice that, because we must model patches as bijections, we could not have circumvented the issue of subtracting from zero by modeling `! add1` as saturating subtraction. Indeed, saturating subtraction is not a `Bijection Nat Nat`, because it sends both `1` and `0` to the same value.

6.2 Contractibility

As usual, paths in `R` are automatically endowed with identities, inverses, and composition. Nevertheless, `doc n = doc n+1` has no more elements than we put in—all paths of that type are homotopic to `add1 n`. Intuitively, this is because `! (add1 n)` goes "backwards" from `doc n+1` to `doc n`, so any sequence of compositions yielding a path `doc n = doc n+1` must have one more `add1 n` than `! (add1 n)`. For example,

```
add1 n ∘ ! (add1 n) ∘ add1 n : doc n = doc n+1
```

but groupoid laws equate this to `add1 n`. Hence, the type `doc n = doc m` uniquely determines a patch, up to homotopy.

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

```
(x : R) → center = x
```

In this case, we choose the center to be `doc 0`. We prove `R` is contractible by `R`-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, that this choice of paths itself respects paths in `R`.

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

isContr : (x : R) → doc 0 = x
isContr (doc n) = toPath n
apd isContr (add1 n) = refl
  : PathOver (λx → doc 0 = x) (add1 n) (toPath n) (toPath n+1)
```

This last `PathOver` simplifies to

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

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

Since `R` is contractible, it is also a mere proposition, and so by Lemma 3.11.10 of the HoTT Book (2013), all its identity types are contractible. In particular, this implies `doc n = doc m` has exactly one patch up to homotopy.

We can prove this fact directly, using the action on paths of `isContr`,

```
apd isContr : {a b : R} (p : a = b)
             → PathOver (λx → doc 0 = x) p
               (isContr a) (isContr b)
```

If we specialize this to paths `doc n = doc m`, we get

```
apd isContr {doc n} {doc m}
  : (p : doc n = doc m) → PathOver (λx → doc 0 = x) p (toPath n) (toPath m)
```

This `PathOver` reduces to `p ∘ toPath n = toPath m`, yielding

```
apd isContr {doc n} {doc m} : (p : doc n = doc m) → p ∘ toPath n = toPath m
```

Precomposing both sides with `! (toPath n)`, we obtain a proof that all paths `p : doc n = doc m` are homotopic to `toPath m ∘ ! (toPath n)`:

```
(p : doc n = doc m) → p = toPath m ∘ ! (toPath n)
```

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

Patch Histories. A major feature of version control is the ability to clone a repository, a process which duplicates a repository by downloading its complete history of patches, and replaying that history in order to rebuild the current contents of that repository.

In the patch theory of Section 4, a complete sequence of patches is a term `p : num = num`, and replaying that sequence of patches amounts to applying `interp p`, which is a `Bijection Int Int`, to some starting `Int`.

In contrast, a sequence of patches in `R` has type `doc n = doc m` for some `n` and `m`, and is only applicable to a repository state classified by `doc n`. To define the concept of a complete history in this patch theory, we must fix some common domain context to which all repositories are initialized. We choose `doc 0`, because no generating patches have it as a codomain, so it is in a sense the "least patched" repository state. Then, a complete patch in `R` is a term of type:

```
Σ(n : Nat). doc 0 = doc n
```

Since `doc 0 = doc n` is contractible, pairs of this type are uniquely determined by their first projection. In other words, the type of complete patches is in bijection with `Nat`—patches applicable to `doc 0` are characterized precisely by the index `n` of their codomain context `doc n`. One direction of this bijection is `fst`; the other is `toPath`.

Just as in Section 4.2, we used a bijection between `num = num` and `Int` to obtain a derived induction principle for `num = num`, here we can use a bijection between complete patches and `Nat`, which we call the type of *complete histories*, to give a derived induction principle for complete patches.

It is not an accident that the indexing type of the patch contexts is in bijection with complete patches—this is automatically the case in any contractible patch theory, for the same reason as above. In fact, the patch theories we consider in Sections 7 and 8 are both contractible, because (as in the present theory) their patch laws and patch applicability require fairly precise invariants about the repository's contents.

7 A patch theory with laws and multiple contexts

In this section, we consider a patch theory with both patch laws and multiple patch contexts, as a simple setting to consider the issues that will arise in the more realistic patch theory of Section 8.

The previous patch theory we considered had one primitive patch applicable to each patch context. Here, we allow exactly *two* primitive patches at each patch context, `add true` and `add false`, which correspond to incrementing one of two natural numbers constituting the repository. We expect patch histories for this theory to be sequences of booleans indicating the sequence of applied patches, so we index the contexts by `Bool Lists`.

```
space R' : Type where
  -- point constructor (patch context):
  doc : Bool List →  R'
  -- path constructor (basic patch):
  add : (x : Bool) {xs : Bool List} → doc xs = doc x::xs
```

Notice that the codomain of the `add x` patch is its domain history `xs` prepended with `x`. This patch theory can be visualized as a tree, where the nodes are histories and the paths label the edges; for example:

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

In such a semantics, any two patches commute: incrementing the same number twice commutes trivially, and incrementing each number in turn commutes because the numbers are independent.

We would like to capture this fact in `R'` by adding patch laws saying that any two patches commute. As in Section 5, this patch law should take the form of a path-between-path constructor in `R'`. But the patch contexts prevent us from equating differing sequences of patches—they do not even have the same type:

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

7.1 Definition of patches

The issue is that the `Bool List` patch histories record the exact sequence of patches applied; we cannot equate any sequences of patches without equating the corresponding patch histories. In other words, for patch composition to commute, we must quotient the `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
  -- point constructors:
  [] : MS
  _::_ : Bool → MS → MS
  -- path constructor:
  Ex : (x y : Bool) (xs : MS) → x::(y::xs) = y::(x::xs)
```

Then, we can index patch contexts by `MSes`, rather than `Bool Lists`. As before, patches prepend a boolean to the context.

```
space R : Type where
  -- point constructor (patch context):
  doc : MS → R
  -- path constructor (basic patch):
  add : (x : Bool) {xs : MS} → doc xs = doc x::xs
  -- pathover-between-path constructor (patch law):
  ex  : (x y : Bool) {xs : MS} →
        PathOver (λs → doc xs = doc s) (Ex x y xs)
          (add x ∘ add y) (add y ∘ add x)
```

The `ex` constructor implements the patch law stating that patch composition is commutative. `(ex x y)` is a `PathOver` because although the codomains of

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

(add x ∘ add y) and (add y ∘ add x) differ—they are doc x::(y::xs) and doc y::(x::xs)—their codomains' indices are equal as multisets by virtue of Ex x y xs.

7.2 *Interpreter*

As we saw in Section 6, it is not possible to model patches incrementing a natural number as bijections on Nat. By analogy with the interpretation discussed there, we would like to interpret doc xs as the type AtLeast t × AtLeast f, where t (resp., f) is the number of times true (resp., false) occurs in xs. First, we write a function to compute these numbers:

```
replay : MS → Nat × Nat

replay [] = (0, 0)
replay (true::xs) = ((fst (replay xs))+1, snd (replay xs))
replay (false::xs) = (fst (replay xs), (snd (replay xs))+1)
ap replay (Ex true true xs) = refl
ap replay (Ex true false xs) = refl
ap replay (Ex false true xs) = refl
ap replay (Ex false false xs) = refl
```

The action of replay on Ex x y xs is a proof that replay respects the equations on multisets. In each case, this is immediately true by unfolding the definition of replay; for example:

```
ap replay (Ex true false xs) = refl :
    ((fst (replay xs))+1, (snd (replay xs))+1)
  = ((fst (replay xs))+1, (snd (replay xs))+1)
```

Then, we define the interpretation:

```
I : R → Type
I (doc xs) = AtLeast (fst (replay xs)) × AtLeast (snd (replay xs))
ap I (add true)  = ua incr-t
ap I (add false) = ua incr-f
apdP (ap I) (ex x y) = GOAL7.1
```

This interpretation sends add true xs to the bijection which increments the first number and leaves the second the same, and vice versa. This map is a bijection because it is a bijection in each coordinate, between AtLeast t and AtLeast t+1 in the first, and between AtLeast f and itself in the second.

```
pairBiject : Bijection A B → Bijection A' B' → Bijection (A × A') (B × B')
pairBiject (f,g,p,q) (f',g',p',q') =
  (λ(x,x') → (f x,f' x'), λ(y,y') → (g x,g' x'), ...)

incr-t : {t f : Nat} →
  Bijection (AtLeast t × AtLeast f) (AtLeast t+1 × AtLeast f)
incr-t = pairBiject (increment t) refl_b
```

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

```
incr-f : {t f : Nat} →
  Bijection (AtLeast t × AtLeast f) (AtLeast t × AtLeast f+1)
incr-f = pairBiject refl_b (increment f)
```

In the last clause of `I`, `apdP` is the action of a function on a `PathOver`, unlike `apd`, which is the action of a function on an ordinary path. (If we define `PathOvers` as ordinary paths using `coe`, as described in Section 2.3, then this is just `apd`.) The goal `GOAL7.1` says that `I` respects the `(ex x y) PathOver`; Unfolding the definitions, this amounts to saying that `I` sends the commuting triangle `ex x y` to a commuting triangle:

$$\begin{tikzcd}
 & \texttt{doc xs} \arrow[dl, "\texttt{add x} \circ \texttt{add y}"'] \arrow[dr, "\texttt{add y} \circ \texttt{add x}"] \arrow[d, phantom, "\texttt{ex x y}"] \arrow[rrr, mapsto, bend left, "\texttt{I}"] & & & \texttt{I(doc xs)} \arrow[dl, "\texttt{ap I (add x} \circ \texttt{add y)}"'] \arrow[dr, "\texttt{ap I (add y} \circ \texttt{add x)}"] \arrow[d, phantom, "\texttt{GOAL7.1}"] & \\
\texttt{doc x::y::xs} \arrow[rr, "\texttt{ap doc (Ex x y xs)}"'] & & \texttt{doc y::x::xs} & \texttt{I(doc x::y::xs)} \arrow[rr, equal] & & \texttt{I(doc y::x::xs)}
\end{tikzcd}$$

As we saw above, `replay` respects the `Ex x y xs` law exactly, so `I` does as well. Thus, `GOAL7.1` amounts to a proof that `ap I` sends `(add x ∘ add y)` and `(add y ∘ add x)` to equal paths in the universe, that is, they induce equal bijections.

Since multisets are quotiented by permutations, the `Nat × Nat` representation computed by `replay` above is in fact isomorphic to `MS`. Nevertheless, we opt to use the `MS` representation to index the patch contexts, since it precisely captures the structure of composable sequences of primitive patches in `R`. Specifically, the elements of `MS` maintain an explicit log of the order in which patches were applied, even though the paths in `MS` identify those logs which differ only by permutation. In contrast, such a log is not maintained at all by the `Nat × Nat` representation.

7.3 *Contractibility*

In a patch theory with multiple contexts, the type of `merge` is somewhat complicated. If we restrict merge to operate on complete patches, then it takes a span of patches with domain `doc []`, and returns a cospan reuniting them:

$$\begin{tikzcd}
 & \texttt{doc []} \arrow[dl, "\texttt{p1}"'] \arrow[dr, "\texttt{p2}"] & \\
\texttt{doc s1} \arrow[dr, dashed, "\texttt{q1}"'] & & \texttt{doc s2} \arrow[dl, dashed, "\texttt{q2}"] \\
 & \texttt{doc s} &
\end{tikzcd}$$

We can write the type of `merge` as follows:

```
merge : {s1 s2 : MS} (doc [] = doc s1) → (doc [] = doc s2) →
        Σ(s : MS). (doc s1 = doc s) × (doc s2 = doc s)
```

In Section 4, the implementation of `merge` was straightforward, but proving merge laws required a derived induction principle obtained from the bijection between patches and `Int`s. In this section, we will establish a bijection between complete patches and patch histories (here, boolean multisets) not only for the purpose of proving merge laws, but also defining `merge` itself.

Namely, if `R` is contractible, then

```
Σ(s : MS). doc [] = doc s
```

is isomorphic to `MS`, because the patch itself is uniquely determined by `s`. Let `toPath` be the function which computes a complete patch from a `MS`, and `cod` its inverse, which projects the codomain index from a complete patch. (It is possible to define `cod` without directly projecting from the type index, as we will see in Section 8.3.) Then, to define `merge` on complete patches, it would suffice to define a merge operation on histories,

```
mergeH : MS → MS → MS
```

and coerce complete patches to and from `MS`:

```
merge p1 p2 =
  let s = mergeH (cod p1) (cod p2)
  in (s, ((toPath s) ∘ !p1, (toPath s) ∘ !p2))
```

In the remainder of this section, we will put aside the issue of defining `mergeH`, and instead establish the bijection described above, by proving that `R` is contractible. This result is somewhat difficult; in fact, one might even expect `R` to have non-trivial loops of the form:

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

The bottom of this loop, `ap doc (Ex x y [])`, originates from an equation in `MS`. The patch law `ex x y` trivializes this loop by equating the two sides, over the bottom path.

The first ingredient of the proof is `toPath`, which computes a path `doc [] = doc s` for each multiset `s`.

```
toPath : (xs : MS) → doc [] = doc xs
toPath [] = refl
toPath (x::xs) = add x ∘ toPath xs
apd toPath (Ex x y xs) = GOAL7.2
  : PathOver (λs → doc [] = doc s) (Ex x y xs)
    (toPath (x::(y::xs))) (toPath (y::(x::xs)))
```

Here, `GOAL7.2` stands for a proof that `toPath` respects equality of multisets. After

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

expanding the definition of `toPath`, the goal `GOAL7.2` states that the following triangle commutes:

$$
\begin{tikzcd}
 & \texttt{doc []} \arrow[dl, "\texttt{add x} \circ \texttt{add y} \circ \texttt{toPath xs}"'] \arrow[dr, "\texttt{add y} \circ \texttt{add x} \circ \texttt{toPath xs}"] \arrow[d, phantom, "\texttt{GOAL7.2}"] & \\
\texttt{doc x::y::xs} \arrow[rr, "\texttt{ap doc (Ex x y xs)}"'] & {} & \texttt{doc y::x::xs}
\end{tikzcd}
$$

Cancelling the two instances of `toPath xs`, we get

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

But this is exactly the type of `ex x y`, 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 loops of the form discussed above. Indeed,

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

is a proof that, whenever `xs` and `ys` are equal multisets, then there is a commuting triangle bounded by `toPath xs`, `doc xs = doc ys`, and `toPath ys`.

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

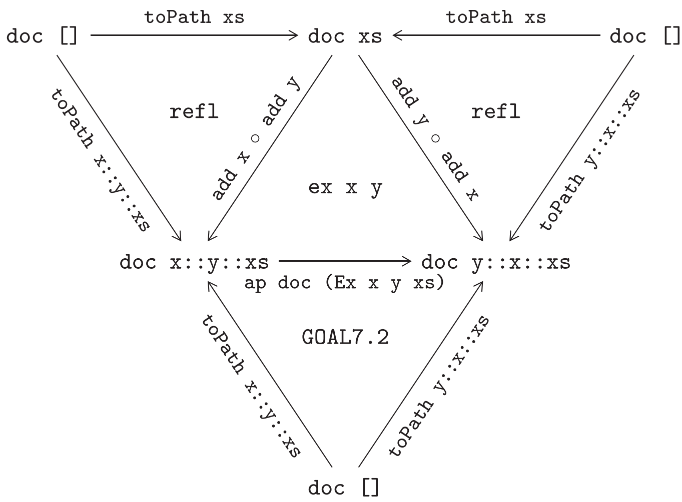
```
isContr : (r : R) → doc [] = r
isContr (doc xs) = toPath xs
apd isContr (add x {xs}) = refl
  : PathOver (λs → doc [] = s) (add x) (toPath xs) (toPath x::xs)
apdP (apd isContr) (ex x y) = GOAL7.3
```

The second clause demands that `isContr` respect `add x`. This is a trivial commuting triangle, because `toPath x::xs` is by definition exactly `add x ∘ toPath xs`. In the third clause, `GOAL7.3` involves paths over paths-over-paths; essentially, it proves that `apd isContr`, which assigns a commuting triangle to every path, respects the patch law `ex x y`, itself a commuting triangle.

The fact that a path-indexed commuting triangle respects another commuting triangle is a commuting *tetrahedron*. Below we have drawn an unfolded version of that tetrahedron; to assemble it, join all three points labeled `doc []` as the apex. The interior of the tetrahedron proves that the top-left and top-right triangles are correlated by the base of the tetrahedron (the middle triangle). We have a machine-checked proof that this tetrahedron commutes[7] but will not discuss it here.

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

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press



8 A patch theory for text files

Finally, we consider a patch theory for a text file (a vector of `String`s), with primitive patches to insert a string `s` as the `l`th line (`ADD s@l`), or remove the `l`th line (`RM l`).

The patch contexts for this theory must at least specify the number of lines in the file, since patches only apply when the specified line number exists—one cannot apply `RM l` to a file with fewer than `l` lines. Thus, our first cut at defining this patch theory is to index patch contexts by the file length, and define `RM l` as a path from `doc n+1` to `doc n`, for any `n+1` at least `l`.

Unfortunately, patches in such a theory cannot be interpreted as bijections between `n`-line files, since deleting a line is not a `Bijection (Vec n+1 String) (Vec n String)`. (An inverse to this function would have to invent the contents of the deleted line.)

Instead, we will index the patch contexts by histories, in this case, sequences of `ADD s@l` and `RM l` for which all the line numbers are within bounds. That is, file lengths determine which histories are well-formed, and histories determine which patches are well-formed!

8.1 *Definition of patches*

Let `History m n` be the type of patch histories which, given `m`-line files, produce `n`-line files. As with `MS` in Section 7, we define `History m n` as a quotient higher inductive type, and equate sequences of patches effecting the same change on files. For example, two `ADD`itions in sequence can be commuted by shifting their line numbers appropriately.

```
space History : Nat → Nat → Type where
  -- point constructors:
  [] : {m : Nat} → History m m
```

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

```
  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, we have also omitted path constructors commuting `ADD-RM`, `RM-ADD`, and `RM-RM`, which can be defined in exactly the same fashion.

We index patch contexts by complete histories, which in this case are elements of `History 0 n`, since they are applicable to empty (length-0) files.

```
space R : Type where
  -- point constructor (patch context):
  doc : {n : Nat} → History 0 n → R

  -- path constructors (basic patches):
  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)

  -- pathover-between-path constructors (patch laws):
  addP-addP-< : {n : Nat} (l1 : Fin n+1) (l2 : Fin n+2)
    (s1 s2 : String) (h : History 0 n) → (pf : l1 < l2) →
    PathOver (λx → doc h = doc x) (ADD-ADD-< l1 l2 s1 s2 h pf)
      (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) → (pf : l1 ⩾ l2) →
    PathOver (λx → doc h = doc x) (ADD-ADD-⩾ l1 l2 s1 s2 h pf)
      (addP s2 l2 ∘ addP s1 l1)
      (addP s1 (l1+1) ∘ addP s2 l2)
```

As in Section 7, the final two constructors stipulate patch laws, over the equations in `History 0 n` to which they correspond. For example, when `l1 < l2`, the first patch law equates the patches

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

in the type family `λx → doc h = doc x`, over the fact that the `ADD-ADD-<` law equates their codomains' histories in `History 0 n+2`.

8.2 Interpreter

Our previous examples of patch contexts determined bounds on the repository's contents—in Section 6, `doc n` classified numbers at least `n`, and in Section 7, `doc xs` classified pairs of numbers pairwise at least `replay xs`.

In contrast, a `History 0 n` *precisely* classifies the repository's contents—exactly one text file can be obtained by applying the specified sequence of patches to the empty file. So while in Section 6 `doc n` is interpreted as the type of numbers `AtLeast n`, here we will interpret `doc h` as the type of text files exactly `replay h`—that is, as the singleton type `S(replay h)`. (Recall from Section 5.3 that for `x:A`, we define `S(x)` as `Σ(y:A).x = y`.)

To compute the file specified by a complete history, we must first implement the primitive patches as functions `add` and `rm` on vectors of `String`s.

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

We use `add` and `rm` to define `replay` as follows:

```
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) =
  GOAL8.1 : 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) =
  GOAL8.2 : add s2 l2 (add s1 l1 (replay h))
          = add s1 l1+1 (add s2 l2 (replay h))
```

Because we have laws equating some histories, `GOAL8.1` and `GOAL8.2` demand that `replay` sends equal histories to equal files, which amounts to showing that `add` satisfies the same laws as `ADD`.

If we interpret `doc h` as the type `S(replay h)`, then we must interpret a patch `p : doc h = doc h'` as a bijection between `S(replay h)` and `S(replay h')`. We can restrict 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)
```

We model `ADD s@l` as `toSingleton (add s l)`, and `RM l` as `toSingleton (rm l)`. These functions are automatically bijections, because any function between contractible types is a bijection. (Name the proof of this fact `singleBiject`.) Putting it all together, we interpret `R` as follows:

```
I : R → Type

I (doc h) = S(replay h)
ap I (addP s l h) = ua (singleBiject (toSingleton (add s l)))
ap I (rmP l h) = ua (singleBiject (toSingleton (rm l)))
```

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

```
apdP (ap I) (addP-addP-< l1 l2 s1 s2 h pf) = <replay respects this law>
apdP (ap I) (addP-addP-≥ l1 l2 s1 s2 h pf) = <replay respects this law>
```

Typechecking `ap I (addP s l h)` requires unfolding the definition of `replay`: it must have type `Bijection S(replay h) S(replay (ADD s@l :: h))`, but by definition, the latter type is `S(add s l (replay h))`.

Then, as before, we can derive the interpretation of patches:

```
interp : {n1 n2 : Nat} {h1 : History 0 n1}
         {h2 : History 0 n2} → (doc h1 = doc h2)
         → Bijection (I (doc h1)) (I (doc h2))
interp p = coeBiject (ap I p)
```

such that `interp (addP s l h)` is `add s l`, and `interp (rmP l h)` is `rm l`.

8.3 Histories

Because `R`'s patch contexts uniquely determine file contents, the type of a complete patch `p : doc [] = doc h` fully specifies its effect! This type information is quite large, and moreover redundant at runtime, in the sense that `interp` can compute the effect of `p` without reference to the type indices. Thus, we hope it is possible to discard the patch contexts at runtime, through some erasure mechanism.

What if, instead of computing the file created by `p`, we want to compute the complete history `h`, it corresponds to (without simply projecting `h` from the type)? Notably, we must compute this information from `p` itself and not `interp p`, because we cannot inspect the intensions of functions `S(file) → S(file')`.

We can do so by means of an alternate interpretation of `R`—just as we computed changes induced on repositories by interpreting patch contexts as singleton files, we can compute the changes induced on *complete histories* by interpreting each `doc h` as `S(h)`:

```
I' : R → Type

I' (doc h) = S(h)
ap I' (addP s l h) =
  ua (singleBiject (toSingleton (λh → ADD s@l :: h))
ap I' (rmP l h) =
  ua (singleBiject (toSingleton (λh → RM l :: h)))
apdP (ap I') (addP-addP-< l1 l2 s1 s2 h p) =
  ADD-ADD-< l1 l2 s1 s2 h p
apdP (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 = coeBiject (ap I' p)
```

As desired, `interpH` takes a patch `p : doc h = doc h'` to a function which, when applied to the unique element of `S(h)`, produces the unique element of `S(h')`. In particular, if `p` is a complete patch, then `fst (interpH p ([],refl))` produces the history `h'`. As with `interp`, `interpH` proceeds recursively on the structure of `p`, without relying on its type information.

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

8.4 Merge

As in Section 7.3, we restrict the merge operation to complete patches:

```
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')
```

Such a function reconciles *all* pairs of complete patches. This may seem impossible, as some patches ordinarily give rise to merge conflicts: For example, given `addP s 0` and `addP s' 0`, neither `[s,s']` nor `[s',s]` is obviously preferable. However, we can always merge conflicting patches by simply undoing both patches. (Of course, a user-friendly interface would ideally recognize this situation and instead prompt the user to manually resolve the conflict.)

In the remainder of this section, we will show how a merge operation `mergeH` for complete histories, and a proof `mergeH` satisfies the merge laws, suffices to define a `merge` satisfying the merge laws. Merging complete histories can be accomplished with standard techniques; for example, using `replay` to convert complete histories into files, defining merging directly on files, and computing a reconciliation which creates the merged file.

To define `merge`, we use `interpH` to convert complete patches to complete histories, then compute the merge of those histories with `mergeH`. The merge of two complete histories `h1` and `h2` is a single history `h'` which has each as a prefix—that is, `h'` reconciles `h1` and `h2` because it is an *extension* of both. Thus, `mergeH` has the type:

```
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'
```

where `Extension h h'` is a proof that `h'` extends `h`:

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

Here, `++ : History n1 n2 → History n2 n3 → History n1 n3` appends two histories.

We complete the definition of `merge` by converting extensions back into paths. First, we convert complete histories to complete patches in the usual way:

```
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'
```

Then, we convert an `Extension h h'` into a path by composing paths 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. Putting all the pieces together, we define `merge` as follows:

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

We can prove the merge laws by observing that `R` is contractible, because then

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

is equivalent to

```
Σ(n : Nat). History 0 n
```

and univalence dictates that all constructions respect equivalence of types. Therefore, since complete histories are equivalent to complete patches, not only does defining a merge on the former automatically result in a merge on the latter, but the merge laws on the former automatically imply the merge laws on the latter. We have a machine-checked proof[8] that (a generalized form of) `R` is contractible, but will not discuss the details here.

But since we manually constructed `merge` from `mergeH` without an appeal to univalence, we will finish the story by proving the merge laws for `merge` manually as well. For 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. The `symmetric` law is not automatic, but rather requires `mergeH` to be symmetric as well:

```
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)))
```

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

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

The first two components of `merge p1 p2` and `merge p2 p1` are equal since `symmetricH` says the same 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.

9 Related work

The first version control system designed around a theory of patches was Darcs (Roundy, 2005; Darcs Project, 2013). For each patch, Darcs computes a (one- or two-sided) inverse patch, and for each composable pair of patches, it attempts to compute a composable pair known as its commutation. The *commutation* of the composable pair (f, g) is another composable pair (g', f') such that $f' \circ g'$ is parallel to $g \circ f$ and has the same effect on a repository state. Additionally, g' has the same effect as g but in the domain context of f, and f' has the same effect as f but in the codomain context of g'. This commutation operation is expected to obey certain laws. Not all patches may be commuted in this way, but those that can may be arbitrarily reordered. Darcs uses this ability to invert and reorder patches to implement operations such as merging and the "cherry-picking" of non-terminal patches from other repositories.

Several efforts have been made to formalize Darcs's patch theory by making precise the laws that patch inverses and commutations should satisfy (Sittampalam, 2005; Roundy, 2009). Dagit (2009) has explored using features of the rich (but not fully dependent) type system of the programming language Haskell to enforce some properties of Darcs's patch theory statically. Closely related to Darcs is an experimental version control system called Camp (Commute And Merge Patches) (Camp Project, 2010), which aims to have its patch theory as well as its implementation verified in the proof assistant Coq (Lynagh, 2012).

Jacobson (2009) explores the interpretation of patch theories similar to that of Darcs in *inverse semigroups*. These are sets equipped with an associative binary operation such that for each element s there is a unique s^* with $ss^*s = s$ and $s^*ss^* = s^*$. Sets with partial bijections form an inverse semigroup that is used to interpret patch theories. The partiality of the maps is used to interpret the domain of applicability of patches, which are thus invertible where they are defined. There is an equivalence between the categories of inverse semigroups and of inductive groupoids, so Jacobson's semantics can be recast in the language of groupoids.

A different approach to interpreting patch theories using mathematical structures results from dropping the requirement of patch invertibility. In this case, patch theories may be interpreted in categories, using the *pushout* construction to interpret merging. This approach has been explored by Houston (2012) and by Mimram & Di Giusto (2013). The latter explicitly construct the category that is the free finite conservative co-completion of a given category of contexts and patches, where the adjoined pushouts signal merge conflicts. The Pijul project (Pijul Project, 2015) is currently developing a distributed version control system based on these ideas.

Löh *et al.* (2007) use the algebra of sets to characterize the repository states associated to a patch and predicate logic to characterize the effects of patch application. This approach is simplified and extended by Swierstra & Löh (2014) using the framework of *separation logic* (Reynolds, 2002), where Hoare triples are used to encode patch applicability and effects and the frame rule is used to specify the part of a repository state that is affected by a patch. This facilitates reasoning about when patches may be composed and when they are independent and thus may be reordered or merged.

10 Conclusion

In this paper, we have defined a number of patch theories within homotopy type theory. We represent patch theories as higher inductive types whose points represent patch contexts, whose paths represent patches between patch contexts, and whose paths between paths represent patch laws. This representation automatically endows patches with a groupoid structure—identity, composition, and inverses, with the corresponding laws—for free, so defining a patch theory requires specifying only the patch contexts, generating patches, and domain-specific patch laws.

We implement a patch theory by mapping it into a univalent universe, thereby sending patch contexts to sets of repositories, and patches to bijections between those sets. Because all functions in homotopy type theory respect paths, such implementations—indeed, all patch operations, like optimizations or merges—automatically satisfy the patch laws. Defining implementations in this way makes essential use of the univalence axiom, which adds a path in the universe for each bijection between sets.

It is possible to use the same guiding principles to define patch theories in a dependently typed programming language lacking univalence and higher inductive types. In Remarks 4.1 and 4.2, we illustrated this for a very simple example. One way to replicate this construction for other patch theories would be to copy and paste the general operations (constructors for identity, inverse, and composition, and their laws) between datatypes; a better way would be to use the abstraction mechanisms already present in dependently typed programming languages to avoid the repetition. For example, one could define a generic datatype of patches, parameterized by a signature describing the repository contexts, the primitive patches, and the equations specific to the primitive patches; the generic datatype would provide identity, inverse, composition, and their laws. Then, to mimic the higher inductive eliminators, one would need a type family identifying other types that have the structure necessary to map into them from a patch theory—a type equipped with a binary relation that has identity, inverse, and composition operations, satisfying the necessary laws.

However, in more abstract terms, such a type family amounts to defining groupoids inside of type theory—the datatype of patches is a construction of the free groupoid on some generators for objects (repository contexts), morphisms (patches), and equations between morphisms (patch laws); and the mapping-out principle for patch theories is the universal property of free groupoids. These constructions are built into homotopy type theory—all types are groupoids, and higher inductive types specify free ones—so we can avoid both explicitly defining free groupoids, and proving that types we map those groupoids into are equipped with a groupoid structure.

A second advantage is that, when programming inside a language where types denote groupoids, many types and terms are simpler than when programming with a construction of groupoids inside of a host language. For example, to refer to the product of two groupoids or the functor category, we need only refer to the product and function *types*, and to write maps between groupoids, we can use λ-terms that simultaneously specify the action both on objects and on morphisms, rather than defining functors by a combinator library (which is effectively in de Bruijn form).

On the other hand, the disadvantage of working with a language of groupoids is that some definitions and operations may not fit naturally into such a framework. For example, modeling patch theories as groupoids forces all patches to have full inverses. While patches typically have post-inverses which undo them, they typically do not have pre-inverses: you cannot delete a file before it is created! In order to represent patches as paths, we had to either choose a theory whose patches were already invertible (Sections 4 and 5), or else restrict the types of patches in order to make them invertible (Sections 6– 8). One solution to this problem is to take the more explicit approach sketched above by using a library for categories inside of homotopy type theory (see Chapter 9 of the HoTT Book (2013)). Another is to scale the language-based approach we studied here to non-invertible patches by using a directed homotopy type theory, following the preliminary work by Licata & Harper (2011).

Another difficulty with modeling patches as paths in a higher inductive type arises when one wants to case-analyze paths to define functions like merging. Unlike in the explicit approach sketched above, where the type of morphisms in a groupoid is a separate type with its own elimination principles, neither path induction nor the induction principle for the patch theory apply directly. Instead, we need to prove derived induction principles, often by establishing bijections with ordinary inductive types (Licata & Shulman, 2013; Univalent Foundations Program, 2013). While this showcases how to apply homotopy-theoretic techniques to programming problems, it is generally more challenging than defining maps out of either ordinary inductive types or quotient types.

Our representation of patch theories requires points, paths, and homotopies; reasoning about these patch theories can require paths between homotopies (e.g., the commuting tetrahedron in Section 7). Because we only use three dimensions of structure, it might be advantageous to work inside a dimensionally truncated homotopy type theory (Licata & Harper, 2012), or explicitly truncate all types (as discussed in Section 5.3).

The computational interpretation of homotopy type theory remains open, but we believe that programming applications will lend insight into the problem. Our work has led us to consider a model of computation in which some steps are propositional equalities, rather than restricting reduction to a subrelation of definitional equality. However, not all propositional equalities are computational—the patch optimizer in Section 5.3 illustrates that propositionally equal terms can compute differently, even though no predicate within homotopy type theory can distinguish them. This is analogous to how, in a non-homotopical type theory with function extensionality (Altenkirch *et al.*, 2007), extensionally equal functions may compute in different ways on the same argument. Accordingly, functions may contain meaningful computational content even when they map into or out of contractible types.

Acknowledgments

We thank the participants of the 2013 IFIP WG 2.8 meeting for helpful conversations about this work, and the anonymous reviewers of ICFP 2014 and this journal for their helpful feedback on this article.

References

Abbott, M., Altenkirch, T. & Ghani, N. (2005) Containers: Constructing strictly positive types. *Theor. Comput. Sci.* **342**(1), 3–27.

Aczel, P. (1977) The strength of Martin-Löf's intuitionistic type theory with one universe. In Proceedings of the Symposium on Mathematical Logic, Oulu, 1974. Dept. of Philosophy, University of Helsinki, Helsinki, Finland. Report No. 2, pp. 1–32.

Altenkirch, T. (2014) *Containers in Homotopy Type Theory*. Lyon: Talk at Mathematical Structures of Computation.

Altenkirch, T. & Kaposi, A. (2015) *Towards Cubical Type Theory*. Preprint. Available at: `http://akaposi.bitbucket.org/nominal.pdf`. (last accessed date 25/08/2016)

Altenkirch, T., McBride, C. & Swierstra, W. (2007) Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification (PLPV '07)*. New York, NY, USA: ACM, pp. 57–68.

Awodey, S. & Warren, M. A. (2009) Homotopy theoretic models of identity types. *Math. Proc. Camb. Phil. Soc* **146**(1), 45–55.

Barras, B., Coquand, T. & Huber, S. (2015) A generalization of the Takeuti–Gandy interpretation. *Math. Struct. Comput. Sci.* **25**(5), 1071–1099.

Bezem, M., Coquand, T. & Huber, S. (2014) A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, Matthes, R. & Schubert, A. (eds), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, vol. 26, pp. 107–128.

Camp Project. (2010) Available at: `http://projects.haskell.org/camp/`. (last accessed date 25/08/2016)

Cavallo, E. (2015) *Synthetic Cohomology in Homotopy Type Theory*. M.Phil. Thesis, Carnegie Mellon University.

Cohen, C., Coquand, T., Huber, S. & Mörtberg, A. (2016) *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. Preprint. Available at: `http://www.cse.chalmers.se/~coquand/cubicaltt.pdf`. (last accessed date 25/08/2016)

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, Douglas J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, James T. & Smith, S. F. (1986) *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall.

Coq Development Team. (2015) *The Coq proof assistant reference manual, version 8.5*. INRIA. Available at: `http://coq.inria.fr/`. (last accessed date 25/08/2016)

https://doi.org/10.1017/S0956796816000198 Published online by Cambridge University Press

Dagit, J. (2009) *Type-Correct Changes—A Safe Approach to Version Control Implementation.* MS Thesis. Corvallis, Oregon, USA: Oregon State University.

Darcs Project. (2013) Available at: `http://darcs.net/`. (last accessed date 25/08/2016)

Gambino, N. & Garner, R. (2008) The identity type weak factorisation system. *Theor. Comput. Sci.* **409**(3), 94–109.

Garner, R. (2009) Two-dimensional models of type theory. *Math. Struct. Comput. Sci.* **19**(4), 687–736.

Hofmann, M. & Streicher, T. (1998) The groupoid interpretation of type theory. *Twenty-Five Years of Constructive Type Theory*. Oxford University Press.

Hou, K.-B. (Favonia) (2014) *Covering spaces in homotopy type theory*. Talk at TYPES 2014.

Houston, R. (2012) *On editing text*. Available at: `http://bosker.wordpress.com/2012/05/10/on-editing-text/`. (last accessed date 25/08/2016)

Jacobson, J. (2009) *A formalization of Darcs patch theory using inverse semigroups.* Available at: `ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf`. (last accessed date 25/08/2016)

Kapulkin, C., Lumsdaine, P. L. & Voevodsky, V. (2012) *The Simplicial Model of Univalent Foundations*. arXiv:1211.2851.

Kleene, S. C. (1945) On the interpretation of intuitionistic number theory. *J. Symbol. Log.* **10**(4), 109–124.

Kreisel, G. (1959) Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in Mathematics*, Heyting, A. (ed), Amsterdam: North-Holland Pub. Co, pp. 101–128.

Lawvere, F. W. (1963) *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories.* PhD Thesis, Columbia University.

Licata, D. R., & Brunerie, G. (2013) $\pi_n(S^n)$ in homotopy type theory. In *Proceedings of the Third International Conference on Certified Programs and Proofs*. New York, NY, USA: Springer-Verlag New York, Inc., pp. 1–16.

Licata, D. R., & Brunerie, G. (2015) A Cubical Approach to Synthetic Homotopy Theory. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '15)*. Washington, DC, USA: IEEE Computer Society, pp. 92–103.

Licata, D. R. & Finster, E. (2014) Eilenberg–MacLane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '14)*. New York, NY, USA: ACM, pp. 66:1–66:9.

Licata, D. R., & Harper, R. (2011) 2-dimensional directed type theory. In *Electron. Notes Theor. Comput. Sci.* **276**, 263–289.

Licata, D. R. & Harper, R. (2012) Canonicity for 2-dimensional type theory. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. New York, NY, USA: ACM, pp. 337–348.

Licata, D. R. & Shulman, M. (2013) Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*. Washington, DC, USA: IEEE Computer Society, pp. 223–232.

Löh, A., Swierstra, W. & Leijen, D. (2007) *A Principled Approach to Version Control.* Preprint. Available at: `http://www.andres-loeh.de/VersionControl.html`. (last accessed date 25/08/2016)

Lumsdaine, P. L. (2009) Weak ω-categories from intensional type theory. In *Typed Lambda Calculi and Applications: 9th International Conference, TLCA 2009, Brasilia, Brazil, July*

1–3, 2009. Proceedings. Curien, P.-L. (ed), Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 172–187.

Lumsdaine, P. L. (2011 April) *Higher Inductive Types: A Tour of the Menagerie*. Available at: http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/. (last accessed date 25/08/2016)

Lumsdaine, P. L. & Shulman, M. (2013) *Higher Inductive Types*. In preparation.

Lynagh, I. (2012 January) *Camp Patch Theory*. Available at: http://projects.haskell.org/camp/files/theory.pdf. (last accessed date 25/08/2016)

McBride, C. (2000) *Dependently Typed Functional Programs and Their Proofs*. PhD Thesis, University of Edinburgh.

Mimram, S. & Di Giusto, C. (2013) A categorical theory of patches. *Electron. Notes Theor. Comput. Sci.* **298**, 283–307.

Nordström, B., Peterson, K. & Smith, J. M. (1990) *Programming in Martin-Löf's Type Theory, an Introduction*. Clarendon Press.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD Thesis, Chalmers University of Technology.

Pijul Project. (2015) Available at: https://pijul.org/. (last accessed date 25/08/2016)

Polonsky, A. (2015) *Internalization of Extensional Equality*. arXiv.1401.1148.

Reynolds, J. C. (2002) Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. Washington, DC, USA: IEEE Computer Society, pp. 55–74.

Roundy, D. (2005) Darcs: Distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. New York, NY, USA: ACM, pp. 1–4.

Roundy, D. (2009 April) *Theory of Patches*. Available at: http://www.cs.tufts.edu/comp/150GIT/archive/david-roundy/theory-patches-2009.pdf. (last accessed date 25/08/2016)

Shulman, M. (2011 April) *Homotopy Type Theory VI: Higher Inductive Types*. Available at: http://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html. (last accessed date 25/08/2016)

Shulman, M. (2015) Univalence for inverse diagrams and homotopy canonicity. *Math. Structures Comput. Sci.* **25**, 1203–1277.

Sittampalam, G. *et al*. (2005) *Some Properties of Darcs Patch Theory*. Available at: http://urchin.earth.li/darcs/ganesh/darcs-patch-theory/theory/formal.pdf. (last accessed date 25/08/2016)

Swierstra, W. & Löh, A. (2014) The semantics of version control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. New York, NY, USA: ACM, pp. 43–54.

Univalent Foundations Program. (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*. Available at: http://homotopytypetheory.org/book. (last accessed date 25/08/2016)

van den Berg, B. & Garner, R. (2011) Types are weak ω-groupoids. *Proc. London Math. Soc.* **102**(2), 370–394.

Voevodsky, V. (2006) A very short note on homotopy λ-calculus. *Unpublished*, September, 1–7.

Warren, M. A. (2008) *Homotopy Theoretic Aspects of Constructive Type Theory*. PhD Thesis, Carnegie Mellon University.