# Towards Proof Repair in Cubical Agda

Cosmo Viola
University of Illinois
Urbana-Champaign
USA

Max Fan
University of Illinois
Urbana-Champaign
USA

Talia Ringer
University of Illinois
Urbana-Champaign
USA

## Abstract

Recent work introduced an algorithm and tool in Coq to automatically repair broken proofs in response to changes that correspond to type equivalences. We report on case studies for manual proof repair across type equivalences using an adaptation of this algorithm in Cubical Agda. Crucially, these case studies capture proof repair use cases that were challenging to impossible in prior work in Coq due to type theoretic limitations, highlighting three benefits to working in Cubical Agda: (1) quotient types enrich the space of repairs we can express as type equivalences, (2) dependent path equality makes it possible to internally state and prove correctness of repaired proofs relative to the original proofs, and (3) functional extensionality and transport make it simple to move between slow and fast computations after repair. They also highlight two challenges of working in Cubical Agda, namely those introduced by: (1) lack of tools for automation, and (2) proof relevance, especially as it interacts with definitional equality. We detail these benefits and challenges in hopes to set the stage for later work in proof repair bridging the benefits of both languages.

***Keywords:*** Proof Repair, Cubical Agda

## 1 Introduction

Writing formal proofs in proof assistants like Coq, Agda, Lean, and Isabelle/HOL is a time-intensive task. Even once written, proofs may break in the face of minor changes in the datatypes, programs, and specifications they are about. User study data suggests that this process of writing and rewriting proofs is ubiquitous during proof development [15], and that it can be challenging to deal with even for experts.

*Proof repair* [13] aims to simplify this process by introducing algorithms and tools that fix formal proofs in response to breaking changes. In this paper, we report on manual proof repair case studies in a fragment of Cubical Agda. Our focus is on a particular class of changes: those that can be described by type equivalences, as in the Pumpkin Pi Coq plugin [14]. Using the features of Cubical Agda, however, we are able to construct equivalences between types that cannot naturally be expressed in Coq. We investigate how the proof repair algorithm from Pumpkin Pi carries over to Cubical Agda on two case studies:

1. **We examine two non-isomorphic representations of the integers**: (a) the representation used in the Cubical Agda standard library, and (b) the Grothendieck group completion of the natural numbers. Cubical Agda allows us to formally construct the latter, which would be challenging in Coq, and repair the addition function and several proofs across representations.
2. **We implement two variations of queues**: (a) those backed by lists and (b) those backed by *pairs* of lists. These types are not equivalent, but by leveraging Cubical Agda's quotient types, we can construct an equivalence that describes the change and repair functions and proofs across it.

Cubical Agda [21] extends Agda to give computational meaning to the univalence axiom and higher inductive types by way of cubical type theory [3, 6], enabling for a computational notion of functional extensionality and quotient types. In moving to Cubical Agda, we report on what is gained:

1. **Quotient Equivalences**: We can support equivalences between quotient types, thereby supporting a broader class of changes than those handled by prior work. Recent work in Cubical Agda showed that certain relations describing changes in behavior can be adjusted to equivalences between quotient types [2]. Since Coq lacks quotient types, prior work in Coq could support only a more restricted class of changes: those that can be described by equivalences between sigma types. By taking advantage of cubical's quotient types, we broaden the class of changes we can support.
2. **Internal Correctness**: We can state and prove the correctness of repaired proofs internally in Cubical Agda, using cubical's dependent path equality. Since Coq lacks univalence, prior work in Coq was resigned to stating the correctness theorems on paper inside of a univalent *metatheory*. By taking advantage of cubical's rich notion of path equality, we are able to state and prove previously unproven theorems directly and formally inside of Cubical Agda.



*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*



ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

3. **Repair to Fast Functions**: We can move between fast and slow representations in Cubical Agda much more easily than in Coq. When repairing proofs across equivalences, the repair methodology we adopt from previous work imposes the same inductive structure on the repaired functions and proofs as the original. This in combination with Coq's type theory is why prior work used ad hoc methods to move between, say, inefficient (but easy to reason about) unary addition, and efficient (but difficult to reason about) binary addition, even when one had a principled way to repair from unary to binary representations of the naturals. In Cubical Agda, however, functional extensionality is a theorem, so we can take proof repair one step further, replacing slow repaired functions with fast ones by chaining proof repair with transport across the extensional equality.

We also report on the challenges we experienced:

1. **Automation**: Automation in Cubical Agda is difficult due to both engineering limitations and type theoretic challenges. Because of this, our proof repair case studies in Cubical Agda are still manual. In contrast, Coq's rich plugin system made it possible to build powerful external proof repair automation in prior work.
2. **Proof Relevance**: While Cubical Agda makes it possible to prove internal correctness of certain repaired proofs for the first time, doing so can be challenging thanks to Cubical Agda's *proof relevance*—the fact that it matters not just *what* is proven, but also *how* that is done. Specifically, we struggled to prove that concrete proofs were repaired correctly, *even when we were able to show that all of the components from which they were composed were repaired correctly*. This is because, when it came to *composing* internal correctness proofs, the contents of those individual components had to line up. Aligning proofs about repaired terms with proofs about repaired types in a way that had the right compositional and definitional behavior proved at times to be prohibitively difficult.

All of our code is available.[1] Our goal is to set the stage for proof repair work bridging the two different type theories and tools, in hopes to get the best of both worlds.

## 2 Problem Definition

The problem that we explore is an extension of the proof repair problem from Pumpkin Pi. Given proofs defined over some old type $A$, and an equivalence between $A$ and some new type $B$, Pumpkin Pi repairs proofs that refer to $A$ to instead refer to $B$. It does this by transforming those proofs in such a way that the repaired proofs no longer refer to $A$.

[1] https://github.com/InnovativeInventor/proof-repair-cubical

We move this problem into a fragment of cubical type theory. By working in cubical, we can expand the set of pairs of equivalent types $A$ and $B$ that we can repair proofs over to include an extremely useful set of changes in types not describable in Coq—quotient type equivalences (Section 2.1). Furthermore, we can internally and formally state what it means for any given repaired proof defined over $B$ to correctly correspond to the original proof over $A$—in terms of dependent path equality (Section 2.2). We will ground this in concrete examples with our case studies in Section 4.

### 2.1 Scope: Quotient Type Equivalences

We wish to generalize Pumpkin Pi to Cubical Agda in order to capture broader use cases. In this paper, we perform proof repair across quotient type equivalences. We will describe what makes these *quotient* type equivalences soon. But in general, a *type equivalence* between two types $A$ and $B$ is an isomorphism (a pair of functions that are mutual inverses) satisfying a particular coherence property [20]. It is possible to form an equivalence from any isomorphism, for example by using `isoToEquiv` in Cubical Agda:

```
isoToEquiv : Iso A B → A ≃ B
```

where $\simeq$ in Cubical Agda denotes a type equivalence.

Like prior work [14], we consider changes in datatypes that can be described by these type equivalences. But by working in Cubical, we can support a broader class of changes than type equivalences in Coq can encode. The trick is to encode changes in datatypes as equivalences between *quotient types*—types equipped with an equivalence relation describing what makes two elements of that type "the same."

Quotient types are supported natively in cubical type theory. In Cubical Agda, quotient types (denoted $A/R$ for a type $A$ and relation $R$) are encoded as higher inductive types taking a type and a relation:

```
data _/_ (A : Type) (R : A → A → Type) : Type
```

An element of this quotient is the equivalence class of any $a$ of type $A$ (denoted $[a]$):

```
[_] : (a : A) → A / R
```

Given two elements $[a_1]$ and $[a_2]$, we have that $[a_1] \equiv [a_2]$ whenever $R\ a_1\ a_2$:

```
eq/: (a₁ a₂: A) → (r: R a₁ a₂) → [ a₁ ] ≡ [ a₂ ]
```

even when it is not true that $a_1 \equiv a_2$.

To give a familiar example of a quotient type, we can construct the natural numbers mod 2, $\mathbb{N}/2$. Two naturals are in the same equivalence class if they have the same parity. We can capture this with the following type:

```
~ : ℕ → ℕ → Type
~ n m = (n mod 2) ≡ (m mod 2)

ℕ/2 = ℕ / ~
```

The resulting type has two equivalence classes, and so every element of the type is equal to either `[ 0 ]` or `[ 1 ]`. For another element, like `[ 2 ]`, we have a proof of equality between `[ 0 ]` and `[ 2 ]` via `eq/ [ 0 ] [ 2 ] refl`. This resulting type is then isomorphic to the booleans via the map `[ 0 ]` ↦ `false`, `[ 1 ]` ↦ `true`, and this can be used to construct a type equivalence via `isoToEquiv`. In Section 4, when we discuss our case studies, we will discuss more complicated examples of quotient types and equivalences.

In this paper, we further restrict ourselves to equivalences between *set quotients*, which have one additional constraint:

```
squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

That is, all proofs of equality between elements of the quotient must themselves be equal to each other. In homotopy type theory parlance, the type $A/R$ must be an h-set; in broader dependent type theory parlance, uniqueness of identity proofs must provably hold for the type $A/R$.[2]

Quotient types are commonly used for mathematics—a use case we will highlight in Section 4.1. But they also let us cleanly express changes in the implementation of datatypes—a benefit we will highlight in Sections 4.2 and 5.1.1.

Repairing proofs across changes that can be expressed as quotient type equivalences amounts to instantiating the proof term transformation from Pumpkin Pi inside of a fragment of cubical type theory; we describe this in Section 3.

### 2.2 Goal: Dependent Path Equality

In cubical type theory, we can state what it means for a repaired proof to be correctly related to the original proof. This is due to *univalence* [3, 6] in cubical type theory—equivalence is equivalent to equality [20].[3] In contrast, prior proof repair work in Coq could describe correctness externally in some univalent metatheory [14], but could not generally prove or state correctness internally inside of Coq without the axiom of functional extensionality [18]. This highlights another benefit of working in Cubical, detailed in Section 5.

We define correctness in terms of a generalization of *path equality*, the primitive notion of propositional equality in cubical type theory. In Cubical Agda, if two terms `a` and `b` of the same type `A` are path equal to each other, we can produce a term of type:

```
a ≡ b
```

Our two-element types from before provide an example of path equality. Using univalence, the isomorphism between $\mathbb{N}/2$ and the booleans can be turned into an equality:

```
ℕ/2 ≡ bool
```

Path equality is actually an instantiation of the more powerful notion of *dependent path equality*, for equality between terms with different types (that is, heterogeneous equality). In Cubical Agda, two terms `a` of type `A` and `b` of type `B` are dependently path equal if we can produce a term of type:

```
PathP p a b
```

where `p` is a path between `A` and `B`. Non-dependent path equality ≡ is defined in terms of dependent path equality at the identity path from a type to itself. For example, let `p` be our proof that $\mathbb{N}/2 \equiv$ `bool`. Then the type:

```
PathP p [ 0 ] false
```

is inhabited, because `[ 0 ]` maps to `false` under the isomorphism we used to create `p`.

We can state correctness of a repaired proof relative to the corresponding original version by stating the correct `PathP` type—the one between the original term and the repaired term at the path between their types. Furthermore, we can do this directly inside of Cubical Agda, without any axioms. The type `PathP p [ 0 ] false` we described above gives such an example: if we repair `[ 0 ]` to `false` across $\mathbb{N}/2 \equiv$ `bool`, then this type states that this repair was correct. We describe how to define correctness more generally using `PathP` when we detail our approach in Section 3.

There are two important things to note here. The first is that, while such a `PathP` type can in theory be not just stated in Cubical Agda but also *proven* (inhabited), in practice, finding a term with the correct type can be quite difficult (see Section 5). The second is that correctness of a repaired proof relies not only on the right `PathP` type being inhabited, but *also* on the new version of the proof no longer containing references to the old version of the proof. This we do not state internally to Cubical Agda, but it should still hold metatheoretically for repair to be correct [13, 14]. This is also the primary way in which proof repair in cubical type theory diverges from (dependent) transport: using transport to move proofs across equalities forces both equivalent types to eternally remain in the codebase.

## 3 Approach

We detail our approach to performing proof repair in Cubical Agda. To begin, we recall the algorithm for proof repair across equivalences in Coq that we will adapt (Section 3.1). Then, we dicuss how to adapt the algorithm to Cubical Agda's type system (Section 3.2). Finally, we show how we can leverage Cubical Agda to produce internal proofs of correctness for the individual components of proof repair (Section 3.3)—something that was not possible in Coq, and was not even done metatheoretically for any examples. We will illustrate this approach on our case studies and discuss the tradeoffs relative to Coq in Sections 4 and 5, respectively.

[2]While homotopy type theory makes it possible to define types for which identity proofs are *not* unique, reasoning about equalities between those types can be challenging, so we defer reasoning about them to later work.
[3]In cubical type theory, univalence is not an axiom, but rather follows computationally from more primitive constructs. We take advantage not of univalence directly, but of these more primitive constructive constructs.

$\langle i \rangle \in \mathbb{N},\ \langle v \rangle \in \text{Vars}, \langle s \rangle \in \{ \text{Prop}, \text{Set}, \text{Type}\langle i \rangle \}$

$\langle t \rangle ::= \langle v \rangle \mid \langle s \rangle \mid \Pi\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \lambda\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \langle t \rangle\ \langle t \rangle \mid \text{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle,\ldots,\langle t \rangle\} \mid \text{Constr}\ (\langle i \rangle, \langle t \rangle) \mid \text{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle,\ldots,\langle t \rangle\}$

**Figure 1.** The grammar of CIC$_\omega$ from Pumpkin Pi [14], adapted from Timany and Jacobs [19]. The terms here are, in order: variables, sorts, dependent product types, functions, applications, inductive types, constructors, and eliminators.

### 3.1 Proof Repair in Coq

To understand how our repair transformation in Cubical Agda works, we first briefly recall how Pumpkin Pi's repair transformation operates in Coq. Terms in Coq obey the syntax of its type theory, the Calculus of Inductive Constructions (CIC$_\omega$) [10], an extension of the Calculus of Constructions [9] with inductive types. The grammar for CIC$_\omega$ is in Figure 1.

Pumpkin Pi leverages this structure to do repair from one type to a mathematically equivalent type. The key insight is that, by Lambek's theorem, any equivalence between types `A` and `B` can be decomposed into components that talk only about `A` and only about `B` respectively [13]. Functions and proofs can be unified with applications of these components, making repair a simple proof term transformation [14].

The original Pumpkin Pi work calls each such decomposed equivalence a *configuration*, comprising pairs of the form `((DepConstr, DepElim), (ι, η))` for types on both sides of the equivalence.[4] `DepConstr` and `DepElim` are, respectively, constructors and eliminators for each type. The constructors must generate the elements of the inductive type, and the eliminator must specify how to consume an element produced by the constructors. These constructors and eliminators take the shape of the *original* type `A`, even if the *repaired* type `B` has a different shape. To use an example from the Pumpkin Pi paper, for the type of unary naturals in Figure 2, we could give the following dependent constructors:

```
Definition depConstrNatZero : Nat.
Definition depConstrNatSuc : Nat -> Nat.
```

Then, to do repair to the binary naturals, we need to provide dependent constructors for that type. These dependent constructors *must correspond across the equivalence.* Thus, we arrive at the following constructors:

```
Definition depConstrNZero : N.
Definition depConstrNSuc : N -> N.
```

These dependent constructors do not share the type signatures of the type's constructors, but rather are two user defined functions corresponding to the constructors for the unary naturals. Similarly, the dependent eliminators for both types take the same shape:

```
Definition depElimNat : forall (P : nat -> Type),
  (P depConstrNatZero) ->
  (forall n : nat, P n -> P (depConstrNatSuc n)) ->
  forall n : nat, P n.

Definition depElimN : forall (P : N -> Type),
  (P depConstrNZero) ->
  (forall n : N, P n -> P (depConstrNSuc n)) ->
  forall n : N, P n.
```

[4] For the purposes of this paper, $\eta$, dealing with $\eta$-expansions of constructors applied to eliminators, will always be trivial, and thus we will ignore it.

```
Inductive nat :=
| O : nat
| S : nat -> nat.
```

```
Inductive positive :=
| xO : positive -> positive
| xI : positive -> positive
| xH : positive.

Inductive N :=
| N0 : N
| Npos : positive -> N.
```

**Figure 2.** The natural numbers represented in Coq, taken from the Pumpkin Pi paper [14]. The first representation is unary, and the second is binary. In `positive`, `xH` is one, `xO` is appending a 0 to the right side of the binary representation, and `xI` is appending a 1 to the right side of the binary representation. Then, `N` is either 0 or a positive binary number.

The fact that these two eliminators have the same shape even when the underlying types do not is exactly why we need the remaining element of the configuration: $\iota$. This gives the $\iota$-reduction rules, which specify how to reduce an application of a dependent eliminator to a dependent constructor. Over the original type, this will be definitional—the underlying proof assistant will handle it automatically. But over the repaired type, if the inductive structure has changed—as it has with binary natural numbers—the underlying proof assistant cannot handle it automatically. Instead, this reduction must be made propositional. In the binary example, $\iota$ describing how to reduce the successor case of `depElimN` is a propositional equality that does not hold definitionally.

Once we have defined the components of the configuration, we are ready to do repair. First, the functions we wish to repair are converted to be in terms of dependent constructors, eliminators, and $\iota$-reduction rules, which Pumpkin Pi could do automatically in many cases. Then, we follow the syntactic transformation outlined in Figure 3.

### 3.2 Proof Repair in Cubical Agda

We directly adapt this transformation to a fragment of Cubical Agda. While Cubical Agda is not based on CIC$_\omega$, many of its types can be viewed as using this same syntax.[5] Our transformation operates on this fragment of Cubical Agda

[5] In Cubical Agda, eliminators are usually user defined in terms of pattern matching and recursion. In Coq, they are automatically derived for inductive types, but still backed by pattern matching and recursion. In line with Pumpkin Pi, we do not repair proofs defined using pattern matching and recursion directly, but rather adapt them to use eliminators.

$\boxed{\Gamma \vdash t \Uparrow t'}$

$$\textsc{Dep-Elim}\ \frac{\Gamma \vdash a \Uparrow b \qquad \Gamma \vdash p_a \Uparrow p_b \qquad \Gamma \vdash \vec{f_a} \Uparrow \vec{f_b}}{\Gamma \vdash \mathrm{DepElim}(a,\ p_a)\vec{f_a} \Uparrow \mathrm{DepElim}(b,\ p_b)\vec{f_b}}$$

$$\textsc{Dep-Constr}\ \frac{\Gamma \vdash \vec{t_a} \Uparrow \vec{t_b}}{\Gamma \vdash \mathrm{DepConstr}(j,\ A)\ \vec{t_a} \Uparrow \mathrm{DepConstr}(j,\ B)\ \vec{t_b}} \qquad \textsc{Eta}\ \frac{}{\Gamma \vdash \mathrm{Eta}(A) \Uparrow \mathrm{Eta}(B)}$$

$$\textsc{Iota}\ \frac{\Gamma \vdash q_A \Uparrow q_B \qquad \Gamma \vdash \vec{t_A} \Uparrow \vec{t_B}}{\Gamma \vdash \mathrm{Iota}(j,\ A,\ q_A)\ \vec{t_A} \Uparrow \mathrm{Iota}(j,\ B,\ q_B)\ \vec{t_B}} \qquad \textsc{Equivalence}\ \frac{}{\Gamma \vdash A \Uparrow B}$$

$$\textsc{Constr}\ \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \mathrm{Constr}(j,\ T)\ \vec{t} \Uparrow \mathrm{Constr}(j,\ T')\ \vec{t'}}$$

$$\textsc{Ind}\ \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{C} \Uparrow \vec{C'}}{\Gamma \vdash \mathrm{Ind}(Ty : T)\vec{C} \Uparrow \mathrm{Ind}(Ty : T')\vec{C'}} \qquad \textsc{App}\ \frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'}$$

$$\textsc{Elim}\ \frac{\Gamma \vdash c \Uparrow c' \qquad \Gamma \vdash Q \Uparrow Q' \qquad \Gamma \vdash \vec{f} \Uparrow \vec{f'}}{\Gamma \vdash \mathrm{Elim}(c, Q)\vec{f} \Uparrow \mathrm{Elim}(c', Q')\vec{f'}}$$

$$\textsc{Lam}\ \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \lambda(t : T).b \Uparrow \lambda(t' : T').b'} \qquad \textsc{Prod}\ \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t' : T').b'}$$

$$\textsc{Var}\ \frac{v \in \mathrm{Vars}}{\Gamma \vdash v \Uparrow v}$$

**Figure 3.** Transformation for repair across $A \simeq B$ in our fragment with configuration `((DepConstr, DepElim), (Eta, Iota))`, from previous work [13].

which resembles $\mathrm{CIC}_\omega$ (Figure 1), extended with quotient types. Specifically, our repair sources exclusively uses eliminators instead of pattern matching, and we exclude higher inductive types from consideration except for the specific case of set quotient types. At present, we only consider cases where a quotient type is the target of repair. As a result, we can perform repair in Cubical Agda following the same rules as Pumpkin Pi follows in Coq (Figure 3). However, because we have not yet built out Pumpkin Pi's automation in Cubical Agda, we directly write our functions in terms of the dependent constructors, eliminators, and $\iota$-reduction rules.

We make one simplifying assumption to accommodate these set quotients. For a type `T` defined using set quotients, we require that the motive provided to the eliminator, `(P: T → Set)` satisfies the condition `(x: T) → isSet (P x)`. This condition states that, for any `x : T`, the type `P x` satisfies the uniqueness of identity proofs, or, in the language of Cubical Agda, is an h-set. Because we forbid higher inductive types aside from set quotients, this condition will be satisfied for many applications in software engineering and verification.

However, this restriction does prevent the use of repair for work involving use of higher inductive types, such as the study of homotopy theory. Perhaps the most relevant limitation is that, since `Set`, the type of types, is not an h-set, we cannot do repair on a type family `(P : T → Set)` such that `P x` is `Set` for some `x`. In the course of this work, however, we have come to believe that this simplifying assumption may not be necessary and may be removed in the future.

### 3.3 Correctness

We would like to be able to prove that our function and proofs were repaired correctly. Stating what it means for repair to have occurred correctly is nontrivial. Intuitively, we would like for terms on the new type to behave the same as terms on the old type. However, because our old and new types are different, we cannot simply state this as a homogeneous equality. Instead, we want to know that the terms behave the same up to the equivalence we repair across.

This was done metatheoretically in a univalent type theory for the Pumpkin Pi transformation that we build off of, but it could not be done internally in general, and it was not proven for any particular example at all. In contrast, in Cubical Agda, we have access to heterogeneous equalities in the form of `PathP` types. This gives us the power needed to formalize the correctness theorems about repair from the Pumpkin Pi paper in Cubical Agda, and even to prove them correct on an example type for the first time.

The shape these theorems take depends on the specific case, but the general theme is to construct a `PathP` between terms in the new and old type assuming the existence of such a `PathP` for all of its subterms. Some of these rules are generic across all types. For example, we can internally prove correctness of the Lam rule of the transformation from Figure 3 by way of functional extensionality:

```
lamOK: {T} {F}
  (f: (t: T i0) → F i0 t) (f': (t: T i1)→ F i1 t)
  (b≡b' : ∀ {t : T i0} {t' : T i1}
    (t≡t' : PathP (λ i → T i) t t') →
  PathP (λ i → F i (t≡t' i)) (f t) (f' t')) →
  PathP (λ i → ∀ (t : T i) → F i t) f f'
lamOK {T} {F} f f' b≡b' = funExtDep b≡b'
```

Here, `i`, `i0`, and `i1` are terms of the interval type, which is a primitive construct in cubical type theory from which path equalities are constructed. The rest is analogous to the transformation: `f` is the left function in the transformation, `f'` is the right function, `F i0` is the type of `f`, `F i1` is the type of `f'`, and all other subterms have the same names.

```
elimOK :
  ∀ (a : ℕ) (b : Int / rInt) (a≡b : PathP (λ i → ℕ≡Int/rInt i) a b) →
  ∀ (PA : ℕ → Type) (PB : Int / rInt → Type) (PBSet : ∀ b → isSet (PB b)) →
  ∀ (PA≡PB : ∀ a b (a≡b : PathP (λ i → ℕ≡Int/rInt i) a b) → PathP (λ i → Type) (PA a) (PB b)) →
  ∀ (PA0 : PA zero) (PB0 : PB depConstrInt/rInt0) →
  ∀ (PA0≡PB0 : PathP (λ i → PA≡PB zero depConstrInt/rInt0 depConstr0OK i) PA0 PB0) →
  ∀ (PAS : ∀ a → PA a → PA (suc a)) (PBS : ∀ b → PB b → PB (depConstrInt/rIntS b)) →
  ∀ (PAS≡PBS :
    ∀ a b (IHa : PA a) (IHb : PB b) a≡b (IHa≡IHb : PathP (λ i → PA≡PB a b a≡b i) IHa IHb) →
    PathP (λ i → PA≡PB (suc a) (depConstrInt/rIntS b) (depConstrSOK a b a≡b) i) (PAS a IHa) (PBS b IHb)) →
  PathP (λ i → PA≡PB a b a≡b i) (Nat.elim {A = PA} PA0 PAS a) (depElimSetInt/rInt PB PBSet PB0 PBS b)
```

**Figure 4.** The theorem stating the correctness condition for the repaired dependent eliminator for a simple example type, which has been proven internally in Cubical Agda. This theorem shows that, if all the inputs to the eliminator correspond to each other across the isomorphism, then the output of the eliminator applications also corresponds across that isomorphism. Here, `depConstr0OK` and `depConstrSOK` are the correctness proofs of the repaired constructors, also proven internally.

Other rules are stated specifying the types being repaired. For example, we proved that the repaired eliminator we defined for a simple quotient equivalence was correct. Our source type was $\mathbb{N}$, and our target was `Int / rInt`, where `Int` = $\mathbb{N} \uplus \mathbb{N}$ and `rInt` is the relation placing `inl n` and `inr n` in the same equivalence class. The dependent constructors and eliminators correspond to the usual ones for $\mathbb{N}$. The correctness condition for a repaired eliminator for a given configuration was stated externally in the Pumpkin Pi paper, but it was not proven for any type. We adapted this theorem to Cubical Agda for our example (Figure 4) and, for the first time, proved that it held (see supplementary material).

Even when one can internally prove individual rules of the transformation correct for a particular example change in types, it can be prohibitively challenging to actually *compose* those proofs to get proofs of the correctness of particular repaired proofs. We discuss the reasons for this in Section 5.

## 4 Case Studies

This paper centers two case studies for manual proof repair in Cubical Agda. First, we conduct repair between two representations of the integers (Section 4.1). Second, we study two common implementations of the queue data structure, and how we can repair from one to the other (Section 4.2). Both of these case studies are empowered by the presence of quotient types in Cubical Agda.

### 4.1 Adding, Fast and Slow

Our first case study is mathematically motivated. We consider a change in the type representing the integers. We repair addition and proofs about addition from one representation to the other. Finally, we recover the repaired proofs for a more efficient version of addition over the repaired type.

***Types.*** We start with the default implementation of the integers in Cubical Agda—two copies of $\mathbb{N}$ glued together with one of them reversed, which we call $\mathbb{Z}$.

```
data ℤ : Type₀ where
  pos    : (n : ℕ) → ℤ
  negsuc : (n : ℕ) → ℤ

~ : (ℕ × ℕ) → (ℕ × ℕ) → Type
~ (x1, x2) (y1, y2) = x1 Nat.+ y2 ≡ x2 Nat.+ y1

GZ : Type
GZ = (ℕ × ℕ) / ~
```

**Figure 5.** The types of our integer representations.

We will repair functions and proofs about $\mathbb{Z}$ to use a representation of integers that may be more familiar to set theorists: Instead of two copies of $\mathbb{N}$ glued together at the ends, we consider the integers as elements of $\mathbb{N} \times \mathbb{N}/\sim$, where $(x_1, x_2) \sim (y_1, y_2) \iff x_1 + y_2 = x_2 + y_1$.[6] We would not be able to express this naturally in Coq, but we can represent this in Cubical Agda by defining this relation explicitly and then quotienting $\mathbb{N} \times \mathbb{N}$ by that relation. We refer to the resulting quotient type as `GZ`. The definitions of $\mathbb{Z}$ and `GZ` can be found in Figure 5. In order to do repair between these two types, we need them to be isomorphic. In this case, the isomorphism is the map we expect: `pos n` maps to `[(n , 0)]`, while `negsuc n` maps to `[(0 , n + 1)]`. Verifying that this map is bijective is straightforward.

***Repair.*** To repair functions and proofs across this change, we first must decompose our isomorphism into a configuration, which we do in Figure 6. Here, we can see the dependent constructors, eliminators, and $\iota$-reduction rules for both types. In addition to the listed $\iota$-reduction rules, there are also reversed versions, which we omit for the sake of space.

Notice that the types for the components indeed correspond with each other, and both sides of the configuration share the same inductive structure. The only change in types

[6] We can view this an an instance of constructing the Grothendieck group from the commutative monoid $\mathbb{N}$, and hence the integers arise as the unique group satisfying the universal property that any monoid homomorphism out of $\mathbb{N}$ can be uniquely extended to a group homomorphism out of $\mathbb{Z}$.

```
depConstrℤPos : ℕ → ℤ
depConstrℤNegSuc : ℕ → ℤ

depElimℤ : (P : ℤ → Type) →
  ((n : ℕ) → P (depConstrℤPos n)) →
  ((n : ℕ) → P (depConstrℤNegSuc n)) →
  (x : ℤ) → P x

ιℤPos : (P : ℤ → Set) →
  (posP : (n : ℕ) → P (depConstrℤPos n)) →
  (negSucP : (n : ℕ) → P (depConstrℤNegSuc n)) →
  (n : ℕ) →
  (Q : P (depConstrℤPos → Set) →
  Q (depElimℤ P posP negSucP (depConstrℤPos n)) →
  Q (posP n)

ιℤNegSuc : (P : ℤ → Set) →
  (posP : (n : ℕ) → P (depConstrℤPos n)) →
  (negSucP : (n : ℕ) → P (depConstrℤNegSuc n)) →
  (n : ℕ) →
  (Q : P (depConstrℤNegSuc → Set) →
  Q (depElimℤ P posP negSucP (depConstrℤNegSuc n)) →
  Q (negSucP n)
```

```
depConstrGZPos : ℕ → GZ
depConstrGZNegSuc : ℕ → GZ

depElimGZ : (P : GZ → Type) →
  ((x : GZ) → isSet (P x)) →
  ((n : ℕ) → P (depConstrGZPos n)) →
  ((n : ℕ) → P (depConstrGZNegSuc n)) →
  (x : GZ) → P x

ιGZPos : (P : GZ → Set) →
  ((x : GZ) → isSet (P x)) →
  (posP : (n : ℕ) → P (depConstrGZPos n)) →
  (negSucP : (n : ℕ) → P (depConstrGZNegSuc n)) →
  (n : ℕ) →
  (Q : P (depConstrGZPos → Set) →
  Q (depElimGZ P posP negSucP (depConstrGZPos n)) →
  Q (posP n)

ιGZNegSuc : (P : GZ → Set) →
  ((x : GZ) → isSet (P x)) →
  (posP : (n : ℕ) → P (depConstrGZPos n)) →
  (negSucP : (n : ℕ) → P (depConstrGZNegSuc n)) →
  (n : ℕ) →
  (Q : P (depConstrGZNegSuc → Set) →
  Q (depElimGZ P posP negSucP (depConstrGZNegSuc n)) →
  Q (negSucP n)
```

**Figure 6.** Configuration for our integer representations. Reversed versions of the $\iota$ rules, denoted with a "$^-$", are omitted.

here new to Cubical Agda compared to Pumpkin Pi is that every motive `P : GZ → Set` comes with the requirement that `((x : GZ)→ isSet (P x))`.

Thus, with the configuration defined, we can perform repair according to the procedure discussed in Section 3. To give one concrete example of a term being repaired, we can consider the case of repairing addition from $\mathbb{Z}$ to `GZ`. First, we give the standard definition of addition in $\mathbb{Z}$:

```
_+ℤ_ : ℤ → ℤ → ℤ
m +ℤ n =
  depElimℤ
    (λ _ → ℤ)
    (λ p → m +pos p)
    (λ p → m +negsuc p)
    n
```

Then, we can follow the repair algorithm to get the term:

```
_+GZ_ : GZ → GZ → GZ
m +GZ n =
  depElimGZ
    (λ _ → GZ)
    (λ _ → isSetGZ)
    (λ p → m +posGZ p)
    (λ p → m +negsucGZ p)
    n
```

Notice that the call to `depElimℤ` in the first term is replaced with one to `depElimGZ` in the second term. We also see that, in the course of this repair, we had to repair two other functions: `+pos` and `+negsuc`. These terms are repaired following the same algorithm, and are omitted for space.

We can also repair theorems about our types. In $\mathbb{Z}$, we are able to prove the following theorem, which states that 0 is a left identity for addition:

```
add0Lℤ : (z : ℤ) → z ≡ (depConstrℤPos 0) +ℤ z
```

By manually following the repair algorithm from Section 3, we in turn get a proof of the repaired theorem for `GZ`:

```
add0LGZ : (z : GZ) → z ≡ (depConstrGZPos 0) +GZ z
```

***Fast and Slow.*** Proof repair allows us to repair functions defined on our type, which are then used in proofs. However, the operation of the repaired functions reflects the inductive structure of the old type, rather than that of the new type. Accordingly, the repaired functions are often inefficient.

Here, this manifests as follows: The most general proof that we repair from $\mathbb{Z}$ to `GZ` is the eliminator for $\mathbb{Z}$, which we use to define both addition and the proof about it. This gives us a repaired eliminator for `GZ`, which occurs inside of the repaired addition function and proof over `GZ`.

But this repaired eliminator is slow, and so are our repaired functions and proofs that use it. It is not the eliminator that one would naturally define for `GZ`. Rather, it internally computes a canonical representative of the equivalence class of that element, either of the form $(n, 0)$ for $n \geq 0$ or $(0, n)$ for

```
addHelpFunc' : (ℕ × ℕ) → (ℕ × ℕ) → (ℕ × ℕ)
addHelpFunc' (n₁ , n₂) (m₁ , m₂) = (n₁ + m₁ , n₂ + m₂)

add'Resp : (a a' b b' : ℕ × ℕ) → ~ a a' → ~ b b'
    → ~ (addHelpFunc' a b) (addHelpFunc' a' b')
-- proof of add'Resp omitted

addGZ' : GZ → GZ → GZ
addGZ' =
  setQuotBinOp isReflR isReflR addHelpFunc' add'Resp
```

**Figure 7.** Our fast addition function on the repaired integers.

$n \geq 1$. Because these canonical elements operate like `pos n` and `negsuc n` from the original type, we can compute on them in the same way as we did the original type—in other words, it follows the inductive structure of $\mathbb{Z}$. This means that, for any computation, we must compute canonical elements of equivalence classes, which is wasteful.

Instead, we would much rather use the more natural function $\lambda(n_1, n_2)(m_1, m_2).(n_1{+}m_1, n_2{+}m_2)$. We are able to define this in Cubical Agda by way of the term in Figure 7. The proofs we repaired are for the repaired addition function, as opposed to this more natural and efficient addition function. Thankfully, in Cubical Agda, we are able to use functional extensionality to show that the slow repaired function and the fast addition function are equal, since they have the same output for any inputs:

```
addEqual : addGZ' ≡ _+GZ_
addEqual = funExt
  (λ x → funExt (λ y → addEqualOnInputs x y))
```

Then, to obtain proofs for our fast addition function, we merely need to substitute the fast function for the repaired function in the proofs—something we can do easily thanks to transport. We can then repair our example proof:

```
add'0LGZ: (z: GZ) →
  z ≡ addGZ' (depConstrGZPos 0) z
add'0LGZ = subst
  (λ y → (z : GZ) → z ≡ y (depConstrGZPos 0) z)
  (sym addEqual)
  add0LGZ
```

This points to an advantage over working in Coq, whose type theory has neither functional extensionality nor transport, forcing prior work in proof repair to take a more ad hoc approach to moving between slow and fast repaired functions. We discuss this and other takeaways in detail in Section 5.

### 4.2 Variations on a Theme of Queues

Next, we repair functions and proofs across a change in implementation of a queue data structure. This is motivated by an example from Angiuli et al. [2], which showed that quotient types can be use to adjust certain relations more general than equivalences into equivalences for use with transport in Cubical Agda. That class of changes was cited in the Pumpkin Pi paper as an example that could not be expressed naturally in Coq with the original framework. In Cubical Agda, we can express this use case naturally, highlighting the benefits of working in a type theory with quotient types—something we discuss in detail in Section 5.1.1.

```
OLQ = List A

~ : (List A × List A) → (List A × List A) → Type
~ (l1, l2) (l3, l4) = l1 ++ (rev l2) = l3 ++ (rev l4)

TLQ = (List A × List A) / ~
```

**Figure 8.** One list queues (`OLQ`) and two list queues (`TLQ`).

***Types.*** Our first implementation represents queues using a single list, given by the type `OLQ` in Figure 8. The intention behind this representation is that elements enter the queue at the front of the list and are removed from the queue at the back of the list. The simplicity of this representation of queues comes with a cost: our dequeue operation runs in linear time.

Our second representation is more complicated, but resolves the runtime issues. Instead of one list, we use a two list representation of queues as `List A × List A`. Here, elements enter the queue by being added to the front of the first list, and are removed from the queue by being removed from the front of the second list. If the second list is empty, the first list is reversed onto the second list.

To do repair between these types, we need them to be isomorphic. However, `List A` and `(List A × List A)` are not naturally isomorphic. Based on our description, we would say that a two list queue `(l1 , l2)` corresponds to the queue `l1 ++ (rev l2)`, where `++` is the list append operator. However, this is not an injective map; multiple two list queues correspond to a single one list queue. To resolve this, we will take a quotient of our type of two list queues. We define the equivalence relation `(l1 , l2)~ (l3 , l4)` $\iff$ `l1 ++ ( rev l2) = l3 ++ (rev l4)` and quotient `(List A × List A)` by it, obtaining the type TLQ of two list queues seen in Figure 8. The resulting two types have an isomorphism by the function `[(l1 , l2)]` $\mapsto$ `(l1 ++ (rev l2))`, which is well defined on our quotient type and has as an inverse the function `l` $\mapsto$ `[(l , [])]`.

***Repair.*** We wish to conduct repair from our one list queue type to our two list queue type. We first write the configuration for our isomorphism (Figure 9). We have defined the standard functions enqueue and dequeue defined for one list queues, and we wish to repair them to our newly defined two list queue type. We also have theorems about these functions whose proofs we want to repair. The main theorem states that enqueue and dequeue are related in the way we expect:

```
returnOrEnq : A → Maybe (Q × A) → Q × A
returnOrEnq a = Cubical.Data.Maybe.rec
```

```
  (depConstrEmpty , a)
  (λ p → (enqueue a (proj₁ p) , proj₂ p))

dequeueEnqueue : (a : A) (q : Q) →
  dequeue (enqueue a q)
  ≡ just (returnOrEnq a (dequeue q))
```

We then repair our functions and proofs. To do this, first we convert our functions and proofs to use the dependent constructors, eliminators, and $\iota$-reduction rules. Pumpkin Pi automates this process, but for now we write them using the configuration components directly. Then, we follow the transformation outlined in Section 3 to repair these terms.

As an example, we can see how dequeue is repaired:

```
dequeueOLQ : OLQ → Maybe (OLQ × A)
dequeueOLQ = depElimOLQ (λ _ → Maybe (OLQ × A))
    nothing recCase where
  recCase: (q : OLQ) (outer : A) → Maybe (OLQ × A)
      → Maybe (OLQ × A)
  recCase q outer =
    Cubical.Data.Maybe.rec
      (just (depConstrOLQEmpty , outer))
      (λ p → just (depConstrOLQInsert outer
        (proj₁ p) , (proj₂ p)))

dequeueTLQ : TLQ → Maybe (TLQ × A)
dequeueTLQ = depElimTLQ (λ _ → Maybe (TLQ × A))
  (λ _ → isSetDeqReturnType) nothing recCase where
  recCase: (q : TLQ) (outer : A) → Maybe (TLQ × A)
      → Maybe (TLQ × A)
  recCase q outer =
    Cubical.Data.Maybe.rec
      (just (depConstrTLQEmpty , outer))
      (λ p → just (depConstrTLQInsert outer
        (proj₁ p) , (proj₂ p)))
```

Inspecting the terms, we can see that the dependent constructors and eliminators for our one list queues are replaced with those for two list queues in our dequeue function.

***Fast and Slow.*** While we have obtained a dequeue operation on two list queues, the repaired implementation is inefficient. Our repaired dequeue, as a consequence of the implementation of our dependent constructors and eliminators, always returns the representative of the equivalence class with all elements on the first list. This is undesirable, since we have to do extra computation to move elements from the second list to the first list. We would much prefer dequeue be implemented in the way originally described, simply taking the first element off the second list in the pair. We can implement this dequeue function in Cubical Agda:

```
fastDequeue : TLQ → Maybe (TLQ × A)
fastDequeue = SetQuotients.rec isSetDeqReturnType
    func wellDefined where
  func : TLQ → Maybe (TLQ × A)
  func ([] , []) = nothing
  func ((a :: l1) , []) =
    just (_/_.[ [] , safe-tail (rev (a :: l1)) ] ,
      safe-head a (rev (a :: l1)))
  func ([] , (a :: l2)) = just (_/_.[ [] , l2 ], a)
  func ((b :: l1) , (a :: l2)) =
    just (_/_.[ (b :: l1) , l2 ] , a)
-- proof of well-definedness omitted
```

Now, we want to know that our repaired dequeue is equal to `fastDequeue`. That way, any theorems we prove about dequeue on one list queues can easily be applied to `fastDequeue` by substituting over the equality. With functional extensionality, we obtain a proof of the following equality:

```
deqIsFastDeq : dequeue ≡ fastDequeue
```

With this theorem, we know that anywhere our repaired dequeue appears, we can instead use our more efficient `fastDequeue`. We again discuss this more in Section 5.1.3, alongside other key takeaways in Section 5.

## 5 Case Study Takeaways

We have seen two examples of repair being conducted in Cubical Agda. We now discuss the advantages Cubical Agda provided in making this repair possible and ergonomic (Section 5.1). We also discuss the challenges we currently face in automation and in completing our internal correctness proofs (Section 5.2).

### 5.1 Advantages

We discuss three advantages working in Cubical Agda has provided us: broadening the scope of types we can repair using quotient type equivalences (Section 5.1.1), being able to state correctness of repair internally (Section 5.1.2), and substituting functions which are pointwise equal through the use of functional extensionality (Section 5.1.3).

**5.1.1 Quotient Equivalences.** The original proof repair work, by Ringer et al., operates between types that are strictly equivalent. However, as we have seen, many objects that feel equivalent are not strictly type equivalent. In some cases, that problem could be circumvented through the use of sigma types to make types equivalent. We could try to use that approach for our quotients, but it would come with severe disadvantages. For example, we could represent our two list queues as the sigma type

$$\sum_{x\,:\,\mathrm{List}\,A\times\mathrm{List}\,A} isCanonical\ x$$

where $isCanonical\ x$ is a proposition stating that $x$ is the canonical representative of its equivalence class. This type is equivalent to our type of one list queues. However, this type has only one representative of each equivalence class present within it. Recall that our `fastDequeue` function operates by removing an element from front of the second list in the pair, and enqueueing is fast because it appends to the front of the first list in the pair. With only one representative

```
depConstrOLQEmpty : OLQ
depConstrOLQInsert : A → OLQ → OLQ

depElimOLQ : (P : OLQ → Type) →
  (P depConstrOLQEmpty) →
  (∀ q a → (P q) → P (depConstrOLQInsert a q)) →
  ((x : OLQ) → P x)

ιOLQEmpty : (P : OLQ → Set) →
  (emptyP : P depConstrOLQEmpty) →
  (insertP :
    (q : OLQ) → (a : A) → (P q) →
    P (depConstrOLQInsert a q)) →
  (Q : P depConstrOLQEmpty → Set) →
  Q (depElimOLQ P emptyP insertP depConstrOLQEmpty) →
  Q emptyP

ιOLQInsert : (P : OLQ → Set) →
  (emptyP : P depConstrOLQEmpty) →
  (insertP : (q : OLQ) →
    (a : A) → (P q) →
    P (depConstrOLQInsert a q)) →
  (a : A) → (q : OLQ) →
  (Q : P (depConstrOLQInsert a q) → Set) →
  Q (depElimOLQ P emptyP insertP
      (depConstrOLQInsert a q)) →
  Q (insertP q a (depElimOLQ P emptyP insertP q))
```

```
depConstrTLQEmpty : TLQ
depConstrTLQInsert : A → TlQ → TLQ

depElimTLQ : (P : TLQ → Type) →
  (∀ x → isSet (P x)) →
  (P depConstrTLQEmpty) →
  (∀ q a → (P q) →
  P (depConstrTLQInsert a q)) →
  (x : TLQ) → P x

ιTLQEmpty : (P : TLQ → Set) →
  (pset : (q : TLQ) → isSet (P q))
  (emptyP : P depConstrTLQEmpty) →
  (insertP : (q : TLQ) →
    (a : A) →
    (P q) →
    P (depConstrTLQInsert a q)) →
  (Q : P depConstrTLQEmpty → Set) →
  Q (depElimTLQ P emptyP insertP depConstrTLQEmpty) →
      Q emptyP

ιTLQInsert : (P : TLQ → Set) →
  (pset : (q : TLQ) → isSet (P q))
  (emptyP : P depConstrTLQEmpty) →
  (insertP : (q : TLQ) →
    (a : A) → (P q) →
    P (depConstrTLQInsert a q)) →
  (a : A) → (q : TLQ) →
  (Q : P (depConstrTLQInsert a q) → Set) →
  Q (depElimTLQ P emptyP insertP
      (depConstrTLQInsert a q)) →
  Q (insertP q a (depElimTLQ P emptyP insertP q))
```

**Figure 9.** Configuration for one list and two list queues. Reversed versions of the $\iota$ rules, denoted with a "$-$", are omitted.

of each equivalence class present in the type, this efficient implementation is not possible. For many data structures, including our queue example, we get performance benefits by using more than one element of our equivalence classes.

We could alternatively capture quotients in Coq by using setoids, which are types paired with an equivalence relation representing equality [1]. When comparing elements of a setoid, the equivalence relation is used in place of equality. As a consequence of this, users of setoids need to juggle multiple notions of equality, unlike our quotient types where the same equality is used universally. Setoids also do not enforce that functions defined on them respect the equivalence relation until a user needs to do rewriting under that function in a proof. With our quotient types, well-definedness is checked statically upon writing the function.

**5.1.2 Internal Correctness.** The authors of Pumpkin Pi [14] define what it means for the proof repair transformation to be correct. However, they were unable to internalize those conditions into Coq. This is because the correctness conditions expressed in the paper relied heavily on the heterogeneous notions of equality found in Cubical Agda. As such, without the use of a project like the HoTT library for Coq, stating these theorems within Coq was not possible [4].

In contrast, Cubical Agda has `PathP` types for heterogeneous equality. Thus, we were able to state the correctness conditions. Our original intent was to state these for our case studies, rather than the simple example we gave in Section 3.3. We postponed doing that to focus on our problems with composing proof rules when repairing proofs, seen in Section 5.2.2. However, we were able to show that individual proof rules repaired correctly, such as with `lamOK` and `elimOK`, as seen in Section 3.3. This has been sufficient to see that functions are repaired correctly. For example, we constructed a proof of the below theorem stating that the addition function was repaired across the equivalence correctly:

```
addCorrect : ∀ (a b : ℕ) (a' b' : Int / rInt) →
  ∀ (pa : PathP (λ i → Nat≡Int/rInt i) a a')
    (pb : PathP (λ i → Nat≡Int/rInt i) b b') →
  PathP (λ i → Nat≡Int/rInt i)
    (add' a b) (addInt/rInt' a' b')
```

**5.1.3 Functional Extensionality.** Repair may naturally result in a function which does not take advantage of the properties of the type being repaired to. We saw this in the queue example, where the repaired dequeue function ran in linear time. We would likely question the value of using the repaired two list queue if forced to use this dequeue function. However, by using functinal extensionality to prove that `dequeue ≡ fastDequeue`, we could easily transport our proofs about repaired functions to `fastDequeue`. This is only possible in a constructive way because functional extensionality is a theorem of Cubical Agda, rather than an axiom.

In Coq, a similar effect can be achieved by reasoning pointwise at every place in every theorem where a function is called on an argument. In [14], this is done for the example in Section 3.1 to get a fast binary addition function. However, this requires users to do rewrites at every location where the function is called in every repaired theorem. If the function is called deep within a series of type declarations, this can become inconvenient for the user. Having an equality between the functions themselves allows the rewrite to be done at the top level, simplifying the process.

Swapping function implementations can also make proofs easier. When proving properties of functions, the implementation of the function dictates how we prove those properties. Consider, again, our queue example. By using `depElimTLQ` to prove properties about queues instead of the quotient eliminator, we can avoid doing reasoning about higher inductive types and instead reason on the structure of a list. Even on the result of repair, we may want to prove theorems using the repaired list-based induction principle and then transport those theorems to fast native functions. We see a similar effect with our integer example. Because our repaired constructors take a natural number as input, using our repaired eliminator avoids reasoning about higher inductive types. For this reason, in our proofs that `addGZ' ≡ _+GZ_` and `dequeue ≡ fastDequeue`, we use our dependent eliminators rather than the native quotient eliminators.

## 5.2 Challenges

We discuss two challenges we have faced in performing proof repair in Cubical Agda. First, we describe the challenges in automating the proof repair process (Section 5.2.1). Then, we describe our challenges in dealing with proof relevance (Section 5.2.2).

**5.2.1 Automation.** Ideally, we would be able to automatically perform proof repair in Cubical Agda, similar to prior work in Coq for repair across type equivalences [14]. However, building automation in Cubical Agda is hard due to current engineering and theoretical limitations. The engineering challenges include:

1. **A lack of rich internal tooling and infrastructure.** The automation in the Cubical Agda standard library operates by directly manipulating the AST of the terms, identifiers, and the type checker monad. In contrast, Coq exposes a significant amount of the internals and provides rich quality-of-life tooling for developing practical proof automation, even going as far as to provide custom debugging tooling and a custom memory allocation profiler.
2. **No metalanguage, no side effects.** In prior work like Pumpkin Pi [14], side effects in the metalanguage of Coq were essential for the implementation of persistent caching and other features that were necessary to build practical, performant automation. Cubical Agda does not have a metalanguage.
3. **Lack of documentation and examples.** There are a handful of domain-specific reflective tactics currently available in the standard library. Beyond these narrow examples, there is little documentation on how to safely build more general automation in a manner that does not increase the trusted computing base, especially when it comes to interacting with the type checker monad or any of Cubical Agda's internals.

There are also significant theoretical challenges to resolve:

1. **Higher dimensional equalities.** We do not know how to adapt prior automated proof repair techniques to track and discharge proof obligations unique to Cubical Agda's type theory, such as the interval type and boundary conditions. In particular, we do not know of a technique or decision procedure that is able to reason about general path equalities and higher inductive types.
2. **Proof relevance.** A proof relevant type theory requires one to care about the manner in which a goal was proved, a detail that poses significant challenges to building automation in Cubical Agda. Personal correspondence with Cubical Agda developers indicates hesitance to build automation infrastructure due to theoretical concerns about automation and proof relevance. We describe this challenge in further detail in the following subsection.

**5.2.2 Proof Relevance.** In Cubical Agda, identity proofs need not be unique, and specific proofs may be needed for goals. This can be mitigated by *truncating* types. For instance, our set quotients have the `squash/` constructor, which enforces the uniqueness of identity proofs (i.e., the type is an h-set). While all equality proofs are now equal, we still need to apply `squash/` to dispatch our proof obligations. This application is not automated, so this creates overhead.

However, `Set` itself is not an h-set. As such, we are forced to reason about the exact nature of equality proofs between types. This comes up frequently when working with `PathP` types. Showing that two `PathP`s are equal requires reasoning about the equality of equality proofs between types. If our type equality proof can be written as `λ i → Q (p i)` where `p` is a path in an h-set and `Q` is a dependent product on elements

of that h-set, we can simply use uniqueness of identity proofs to rewrite p to any other path with the same endpoints. It is not always simple, or perhaps even possible, to frame the equality proof in this way. In that case, showing equalities between PathPs has been challenging for us.

One example where this becomes highly relevant is in our statement of the correctness of repair. We were able to prove theorems showing that individual repair transformation rules operated correctly, and we showed that we correctly repaired addition. To show a proof was repaired correctly, however, has proven harder. We need to be able to *compose* those proofs recursively to derive a proof of correctness for that term. Our correctness statement is a PathP, and thus composing these proofs involves producing PathPs from other PathPs. However, as we apply our proof rules, the proof of equality for the types of the endpoints of the PathP grows in complexity. Eventually, our proof rules fail to apply because of mismatches in the type equality proofs.

To give a concrete example, on our simple instance of repair on natural numbers we attempted to prove that a proof of addition being commutative was repaired correctly. Trying to compose our proof rules in the natural way from the top down results in a goal of a PathP along the equality

```
λ i → addCommCorrectType zero depConstrInt/rInt0
        depConstr0Correct (suc b)
        (depConstrInt/rIntS b')
        (depConstrSCorrect b b' b≡b') i
```

However, applying our proof rules from the bottom up to fill that hole produces a PathP along the equality

```
(λ i →
  depConstrSCorrect' i
   (addCorrect zero b depConstrInt/rInt0 b'
     depConstr0Correct b≡b' i)
  ≡ depConstrSCorrect' i
     (addCorrect b zero b'
       depConstrInt/rInt0 b≡b' depConstr0Correct i))
```

To use this PathP to fill the hole, we need to convert it to be along an equal path to the goal. Thus far, we have had limited success doing this in a principled way such that our proof rules are composable. Resolving these issues, so that our proofs rules compose correctly, requires additional work.

## 6 Related Work

**Proof Repair.** This work is based on the proof repair work by Ringer et al. [14]. This work examines how repair can be adapted to Cubical Agda, and compares how Cubical Agda makes various parts of the repair process more or less challenging. Pumpkin Pi's configuration discovery procedures are in turn based on Pumpkin Patch [16]. Sisyphus by Gopinathan et al. [11] is another recent proof repair tool. Their tool repairs proofs of imperative OCaml programs verified in Coq using separation logic, whereas our work focuses on repair of Coq proofs about functional programs.

**Proof Reuse.** Proof repair is an instance of proof reuse, which seeks to use existing proofs in new goals. Other work in proof reuse includes CoqEAL [7] which uses refinement relations to verify properties of efficient functions using proofs on functions that are easy to reason about. In Isabelle/HOL, the Transfer package [12] uses automation to transfer proofs between types. Both approaches require the source and target type to remain in the codebase, unlike proof repair.

**Quotients and Equivalences.** Our work uses quotient types to form equivalences between types. Quotient types exist in other proofs assistants besides Cubical Agda, such as Isabelle/HOL. Bortin and Lüth [5] use quotient types to construct theories in Isabelle, such as multisets and finite sets as quotients of lists. Coq does not have quotient types, but it does have setoids [1], which do not explicitly form equivalence classes like quotients do.

In Angiuli et al. [2], the authors define the structure identity principle. This principle allows them to build data types using a set of predefined combinators, which they call raw structures. Then, users can provide a set of axioms for a raw structure, and can transport those axioms onto any equivalent raw structure. The first example present in that paper is the queue example which we have also studied in our work. Because it uses transport, using the structure identity principle requires the user to keep both versions of the type in their codebase. We avoid that problem, but also have to reason more closely about the inductive structure of our types.

**Univalent Foundations.** This project was conducted in Cubical Agda. Cubical Agda is an implementation of cubical type theory [21]. Cubical type theory [3, 6, 8] was developed to give a constructive account of the univalence axiom. By working in Cubical Agda, we are able to state internal correctness of our repair transformation and have a computational interpretation of functional extensionality. Cubical type theory itself is a derivative of Voevodsky's homotopy type theory [20], which presents the univalence axiom nonconstructively. Homotopy type theory has additionally been implemented in Coq as the HoTT library [4].

Work has been done to approximate univalence in Coq. In Tabareau et al. 2018 [17], type classes are used to define univalent parametricity, which allows the transport of a restricted class of functions and theorems. In subsequent work, they extend the class of terms able to be transported [18].

## 7 Conclusions & Future Work

We have explored manual proof repair in Cubical Agda through the use of two case studies, one on representations of integers and one on queue data structures. In both cases, we were able to repair useful functions and theorems from one type to an equivalent quotient type. By using quotient types, we could then transport those theorems onto more efficient versions of the repaired functions. Additionally, we could state internal correctness theorems about repair, which

would not have been possible in Coq. In the future, we intend to work towards resolving the challenges we have identified, improving automation and producing internal correctness proofs for the repair of arbitrary terms.

## Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

## References

[1] 1999-2023. Coq Reference Manual, Generalized Rewriting. https://coq.inria.fr/refman/addendum/generalized-rewriting.html

[2] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing Representation Independence with Univalence. *Proc. ACM Program. Lang.* 5, POPL, Article 12 (jan 2021), 30 pages. https://doi.org/10.1145/3434293

[3] Carlo Angiuli, Robert Harper, and Todd Wilson. 2017. Computational Higher-Dimensional Type Theory. *SIGPLAN Not.* 52, 1 (jan 2017), 680–693. https://doi.org/10.1145/3093333.3009861

[4] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 164–172. https://doi.org/10.1145/3018610.3018615

[5] Maksym Bortin and Christoph Lüth. 2010. Structured Formal Development with Quotient Types in Isabelle/HOL. In *Proceedings of the 10th ASIC and 9th MKM International Conference, and 17th Calculemus Conference on Intelligent Computer Mathematics* (Paris, France) *(AISC'10/MKM'10/Calculemus'10)*. Springer-Verlag, Berlin, Heidelberg, 34–48.

[6] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

[7] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 147–162.

[8] Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 255–264. https://doi.org/10.1145/3209108.3209197

[9] Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[10] Thierry Coquand and Christine Paulin-Mohring. 1990. Inductively defined types. In *COLOG-88*. Springer, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

[11] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI, Article 107 (jun 2023), 25 pages. https://doi.org/10.1145/3591221

[12] Brian Huffman and Ondřej Kunčar. 2013. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 131–146.

[13] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington.

[14] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 112–127.

[15] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3372885.3373823

[16] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting Proof Automation to Adapt Proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 115–129. https://doi.org/10.1145/3167094

[17] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.* 2, ICFP, Article 92 (jul 2018), 29 pages. https://doi.org/10.1145/3236787

[18] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM* 68, 1, Article 5 (jan 2021), 44 pages. https://doi.org/10.1145/3429979

[19] Amin Timany and Bart Jacobs. 2015. First Steps Towards Cumulative Inductive Types in CIC. In *Theoretical Aspects of Computing - ICTAC 2015*, Martin Leucker, Camilo Rueda, and Frank D. Valencia (Eds.). Springer International Publishing, Cham, 608–617.

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

[21] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (jul 2019), 29 pages. https://doi.org/10.1145/3341691