Proof Repair across Quotient Type Equivalences

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

Proofs in proof assistants like Coq can be brittle, breaking easily in response to changes. To address this, recent work introduced an algorithm and tool in Coq to automatically repair broken proofs in response to changes that correspond to type equivalences. However, many changes remained out of the scope of this algorithm and tool—especially changes in underlying *behavior*. We extend this proof repair algorithm so that it can express certain changes in behavior that were previously out of scope. We focus in particular on equivalences between *quotient types*—types equipped with a relation that describes what it means for any two elements of that type to be equal. Quotient type equivalences can be used to express interesting changes in representations of mathematical structures, as well as changes in the underlying implementations of data structures.

We extend this algorithm and tool to support quotient type equivalences in Coq. Notably, since Coq lacks quotient types entirely, our extensions use Coq's setoid machinery to represent quotients externally. Specifically, (1) our extension to the algorithm supports new changes corresponding to setoids, and (2) our extension to the tool supports this new class of changes and further automates away some of the new proof obligations. We ground our setoid extensions by way of a discussion of a corresponding manual proof repair approach in Cubical Agda, which supports quotient types and allows for some internalization of the correctness criteria for proof repair. We demonstrate our extensions on proof repair case studies for previously unsupported changes.

Keywords: Proof Repair, Cubical Agda, Coq, Quotient Types

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 [22], and that it can be challenging to deal with even for experts.

Proof repair [19] aims to simplify this process by introducing algorithms and tools that fix formal proofs in response to breaking changes. Prior work introduced a Coq plugin called Pumpkin Pi for proof repair across changes in datatypes that can be described by type equivalences [21]. Pumpkin Pi could handle only a limited class of changes corresponding to non-equivalent types by way of expressing those changes as equivalences between sigma types. The power of this was limited, however, and could not represent fundamental changes in *behavior* in a practical way.

In this work, we extend proof repair to support equivalences between *quotient types* (Section 2). Recent work in Cubical Agda showed that certain relations describing changes in behavior can be adjusted to equivalences between quotient types [2]. However, unlike Cubical Agda, Coq lacks quotient types entirely, so one cannot use the original Pumpkin Pi transformation as-is to support this class of changes. To handle this, we represent quotient types externally using Coq's setoid machinery, and we represent quotient type equivalences as setoid equivalences (Section 3). We then extend the proof transformation to support the newly generated equality proof obligations (Section 4).

We implement the extended algorithm by way of an extension to the implementation of Pumpkin Pi (Section 5). Our implementation includes new automation, both to support repair across this class of changes, and to automate away some of the newly generated proof obligations corresponding to this class of changes. We demonstrate our extended implementation on three case studies that cannot be handled by prior proof repair work—two that are mathematical in nature and one that deals with changes in behavior (Section 6). We ground the meaning of our extended transformation by way of a discussion comparing to a manual approach in Cubical Agda (Section 7). Our contributions are:

1. an **extension** to the algorithm for proof repair across type equivalences that supports quotient type equivalences represented via setoids,
2. an **implementation** of this extension in Pumpkin Pi,
3. new **automation** in this implementation to minimize the proof burden of setoid-specific proof obligations,
4. a **demonstration** of the above supporting new use cases by way of three case studies, and

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

5. a **discussion** grounding the meaning of the above by comparing to a manual approach in Cubical Agda.

Our code is available in supplementary material.

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. In this way, it operates a lot like an automated transfer or transport method (see Section 8), but it also goes one step further: since A represents an old version of the type, and B a new version, repair further removes all references to A in the repaired functions and proofs, so that they refer only to B. In contrast, transport and transfer methods typically produce terms that refer to both A and B, for example by applying functions mapping between them.

We expand this problem to allow for the set of pairs of equivalent types A and B that we can repair proofs over to include a useful class of changes in types not directly describable in Coq—those that can be described by quotient type equivalences (Section 2.1). For simplicity, we start by considering this problem in a fragment of cubical type theory which supports quotient types directly. In such a context, 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 later explain how we can approximate this externally in Coq using setoids (Section 3), and we will return to cubical for a discussion at the end (Section 7).

2.1 Scope: Quotient Type Equivalences

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 [29]. It is possible to form an equivalence from any isomorphism.

Like Pumpkin Pi in Coq, we consider changes in datatypes that can be described by these type equivalences, but we also consider a broader class of changes than type equivalences encode in Coq. 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." Since Coq cannot directly express these types, we will later approximate them externally.

For now, let us work in cubical, which has internal quotient types. We denote quotient types as A/R for a type A and a relation R. An element of this quotient is the equivalence class of any a of type A. Given elements $[a_1]$ and $[a_2]$, we have that $[a_1] \equiv [a_2]$ whenever $R\ a_1\ a_2$, even when it is false that $a_1 \equiv a_2$. For example, consider the natural numbers mod 2, $\mathbb{N}/2$. Two naturals are in the same equivalence class if they have the same parity. This 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 ]`, since both are even.

Given two isomorphic types, at least one of which is a quotient type, we can form type equivalences between them in cubical just as we normally could with other isomorphic types. For example, $\mathbb{N}/2$ is isomorphic to the booleans via the map `[ 0 ]` $\mapsto$ `false`, `[ 1 ]` $\mapsto$ `true`, and this can be used to construct a type equivalence. In this paper, we further restrict ourselves to equivalences between *set quotients*, which have the additional constraint that all proofs of equality between elements of the quotient must themselves be equal to each other. In homotopy type theory parlance, A/R must be an h-set; in broader dependent type theory parlance, uniqueness of identity proofs must provably hold for A/R.

Quotient type equivalences are useful for mathematics as well as for describing changes in implementation of datatypes. For example, in Section 6.2 we will show how to use quotient type equivalences to repair functions and proofs between two non-isomorphic implementations of a queue: one backed by a single list, and one backed by a pair of lists. For the latter, we will also implement an efficient dequeueing function that operates by removing an element from the front of the second list of the pair, and an efficient enqueueing function that works by appending to the front of the first list in the pair. We will then use a straightforward methodology to obtain repaired proofs over the efficient functions as well.

Supporting quotient type equivalences makes repair across this change possible for the first time. In Pumpkin Pi, some changes more interesting than vanilla type equivalences could be expressed as equivalences between sigma types; we could try that approach for quotients, but it would come with severe disadvantages. For example, we could represent queues backed by pairs of lists as the sigma type:

$$\sum_{x : \text{List } A \times \text{List } A} \mathit{isCanonical}\ x$$

where *isCanonical* x is a proposition stating that x is the canonical representative of its equivalence class. This type is equivalent to one list queues, but has only one representative of each equivalence class. This makes an efficient implementation of both enqueue and dequeue impossible. Quotient types empower us to obtain repaired functions and proofs that operate over the efficient implementations in the end.

2.2 Goal: Dependent Path Equality

In cubical type theory, we can in general state what it means for a repaired proof to be correctly related to the original

proof. This is because cubical type theory is *univalent* [3, 8]—equivalence is equivalent to equality [29].[1]

We define correctness in terms of a generalization of *path equality*, the primitive notion of propositional equality in cubical. If two terms `a` and `b` of the same type `A` are path equal, we can produce a term of type `a ≡ b`. Using univalence, we can obtain an isomorphism from any path equality. For example, from our isomorphism between $\mathbb{N}/2$ and the booleans, we can obtain a path equality of type $\mathbb{N}/2 \equiv$ `bool`.

Path equality is actually an instantiation of the more powerful heterogeneous notion of *dependent path equality*. In this paper, we use Cubical Agda syntax to denote dependent path equality. Terms `a : A` and `b : B` are dependently path equal if there is a term of type `PathP p a b` where `p` is a path between `A` and `B`. For example, let `p` be our proof that $\mathbb{N}/2 \equiv$ `bool`. Then there is a term of type `PathP p [ 0 ] false` because `[ 0 ]` maps to `false` under the isomorphism we used to create `p`. Non-dependent path equality ≡ is dependent path equality at a single type, where `p` is the identity path.

We can state correctness of a repaired proof relative to the corresponding original version by defining the correct `PathP` type—the one between the original term and the repaired term at the path between their types. For example, the type `PathP p [ 0 ] false` can be viewed as stating that it is correct to repair `[ 0 ]` to `false` across $\mathbb{N}/2 \equiv$ `bool`.

We will describe how to use this to define correctness more generally in Section 7. We will also formalize and prove some of these correctness theorems for the very first time, for manual proof repair in of Cubical Agda. Consistently with prior work, though, our extension to Pumpkin Pi for Coq will simply aspire to satisfy the correctness criteria from Pumpkin Pi, which are stated in an unspecified univalent metatheory, without any correctness proofs.

We will further require that repaired functions and proofs no longer contain references to the old version of those functions and proofs, defined over the old version of the type. Also consistently with prior work, we will not state this formally. But this is the primary way in which proof repair diverges from (dependent) transport: using transport to move proofs across equalities forces both equivalent types to eternally remain in the codebase. Repair lets us move on and remove the old code.

3 Trick: Setoid Equivalences

We now return from cubical to Coq, the proof assistant that our extension to Pumpkin Pi targets. In contrast with cubical, Coq's type theory does not support quotient types at all. So how can we support proof repair across quotient type equivalences when quotient types do not even exist?

[1] 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.

Our answer is to work with setoids—an external notion of quotients. Setoids in Coq are types paired with an equivalence relation representing equality [25]. For example, we can represent our $\mathbb{N}/2$ type from Section 2.1 in Coq as the setoid (`nat`, `mod_two`), where `mod_two` is again equivalence modulo 2. In this case, we call `nat` the carrier of the setoid. We define functions and theorems on elements of the carrier.

Coq has a `Setoid` type class, and any setoid defines an instance of this type class. However, our extension will not use this type class. Instead, we will understand metatheoretically that any pair of a type and an equivalence relation on that type forms a setoid, and we will use the machinery that derives from instances of the `Equivalence` and `Proper` type classes. We can view any type as a setoid this way by letting the equivalence relation be propositional equality, but more interesting setoids will use equivalence relations that do not correspond to propositional equality in Coq.

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 with native quotient types where the same equality is used universally. For example, in cubical we can compare equality of elements of $\mathbb{N}/2$ using the native equality $a \equiv b$, but in Coq, when considering (`nat`, `mod_two`) as a setoid, it is too strong to claim that `a = b`. Instead, we compare equality of elements of the setoid (`nat`, `mod_two`) by using the equivalence relation `mod_two a b`.

Coq's setoids do not enforce that functions defined on them respect the equivalence relation until a user needs to rewrite under that function in a proof, unlike with native quotient types, where well-definedness is checked statically upon writing the function. For example, we can define:

```
f (x : nat) : bool := eqb x 3
```

as a valid function for our (`nat`, `mod_two`) setoid in Coq. However, unlike with propositional equality, it is *not* the case that `f x1 = f x2` whenever `mod_two x1 x2`, and we *cannot* rewrite `f x1` to `f x2` using a proof that `mod_two x1 x2`.

In other words, this particular choice of `f` is not *proper*. When it is the case that, for setoids (`A`, `eqA`) and (`B`, `eqB`) and a function `f : A -> B`, the theorem:

```
∀ (a1 a2 : A), (eqA a1 a2) -> eqB (f a1) (f a2)
```

holds, we say that `f` is proper. A function satisfying this property defines an instance of the `Proper` type class in Coq. For example, consider `isEven : nat -> bool`, which sends even numbers to `true` and odd numbers to `false`. This is proper considering the domain as the setoid (`nat`, `mod_two`) and the codomain as the setoid (`bool`, `=`).

To represent quotient type equivalences in Coq, we define a notion of a *setoid equivalence*. Two setoids (`A`, `eqA`) and (`B`, `eqB`) are equivalent if there is a pair of functions `f : A -> B` and `g : B -> A` satisfying the following properties:

- `f` and `g` are proper.

$\langle i \rangle \in \mathbb{N}$, $\langle v \rangle \in$ Vars, $\langle s \rangle \in \{$ Prop, Set, 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 [21], adapted from Timany and Jacobs [28]. The terms here are, in order: variables, sorts, dependent product types, functions, applications, inductive types, constructors, and eliminators.

- `∀ (a : A), eqA (g (f a)) a`
- `∀ (b : B), eqB (f (g b)) b`

As an example, the `isEven` function is one half of an equivalence between (`nat`, `mod_two`) and (`bool`, `=`). It has as an inverse that sends `true` to 0 and `false` to 1. It is easy to verify that this pair of functions satisfies the above properties.

4 Approach: Proof Term Transformation

The way that Pumpkin Pi repairs proofs across type equivalences is by directly transforming proof terms across those equivalences (Section 4.1). We extend Pumpkin Pi's transformation to support setoid equivalences (Section 4.2).

4.1 Pumpkin Pi's Transformation

The Pumpkin Pi transformation that we extend operates over terms in the the type theory of Coq, the Calculus of Inductive Constructions (CIC$_\omega$) [13]. CIC$_\omega$ extends the Calculus of Constructions [12] with inductive types. The grammar for CIC$_\omega$ is in Figure 1; the type rules are standard and omitted.

Pumpkin Pi implements proof repair over terms in CIC$_\omega$ by transforming proof terms implemented over an old type `A` to instead be implemented over a new version of that type `B`. The key insight behind this transformation is that, by Lambek's theorem, any equivalence between types `A` and `B` can be decomposed into separate components that talk only about `A` and only about `B` [19]. Functions and proofs can be unified with applications of these components, reducing repair to a simple proof term transformation replacing components that talk about `A` with their counterparts over `B` [21].

Pumpkin Pi calls each such decomposed equivalence a *configuration*, comprising pairs of the form:

`(DepConstr, DepElim) , (ι, η)`

for types on both sides of the equivalence.[2] `DepConstr` and `DepElim` are, respectively, constructors and eliminators for each type. The constructors must generate the elements of the type, and the eliminator must specify how to consume an element produced by the constructors. These constructors and eliminators must take the *same shape*, even if `A` and `B` themselves have different shapes.

[2]For the purposes of this paper, η, dealing with η-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 naturals represented in Coq, taken from the Pumpkin Pi paper [21], in unary (left) and binary (right). In `positive`, `xH` is one, `xO` is appending a 0 to the right side of the binary representation, and `xI` is appending a 1 instead. Then, `N` is either 0 or a positive binary number.

To use an example from the Pumpkin Pi paper, for the type of unary naturals in Figure 2, we could choose `depConstr` for `nat` to be `O` and `S`, the constructors for `nat`. However, to repair to the binary naturals, we need to provide dependent constructors `N`. These constructors *must correspond across the equivalence*, so we arrive at constructors with these types:

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

These dependent constructors *do not* share the type signatures of `N`'s constructors, but rather are user-defined functions corresponding to `nat`'s constructors. Likewise, the dependent eliminator for `N` takes the shape of `nat`'s eliminator:

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

The fact that both dependent eliminators must have the same shape even when the underlying types do not is exactly why we need the remaining element of the configuration: ι. This gives the ι-reduction rules, which specify how to reduce an application of a dependent eliminator to a dependent constructor. When the shape is of the underlying type is the same as the shape of the configuration components, as is true for for `nat`, this will be definitional—the proof assistant will handle it automatically. But if the inductive structure has changed, as it has `N`, this will be a propositional equality.

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 explicitly refer to the configuration terms. Sometimes this happens via manual user annotation, and sometimes this happens via custom unification machinery inside of Pumpkin Pi that does this automatically for some classes of changes (see Section 5.2). Then, we follow the syntactic transformation outlined in Figure 3. We will discuss what it means for this to be correct in Section 7.2.

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

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

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

$$\textsc{Iota}\ \frac{\Gamma \vdash q_A \Uparrow q_B \qquad \Gamma \vdash \vec{t_A} \Uparrow \vec{t_B}}{\Gamma \vdash \mathrm{Iota}(j,\ A,\ q_A)\ \vec{t_A} \Uparrow \mathrm{Iota}(j,\ B,\ q_B)\ \vec{t_B}} \qquad \textsc{Equivalence}\ \frac{}{\Gamma \vdash A \Uparrow B} \qquad \textsc{Constr}\ \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \mathrm{Constr}(j,\ T)\ \vec{t} \Uparrow \mathrm{Constr}(j,\ T')\ \vec{t'}}$$

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

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

Figure 3. Transformation for repair across $A \simeq B$ with configuration `((DepConstr, DepElim), (Eta, Iota))`, from previous work [19]. Our work adapts and extends this transformation.

$\boxed{\Gamma \Uparrow \Gamma'}$

$$\textsc{LiftEmpty}\ \frac{}{() \Uparrow ()} \qquad \textsc{LiftCons}\ \frac{\Gamma \Uparrow \Gamma' \qquad \Gamma \vdash x \Uparrow x' \qquad \Gamma \vdash X \Uparrow X'}{(\Gamma, x : X) \Uparrow (\Gamma', x' : X')}$$

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

$$\textsc{EquivApp}\ \frac{\Gamma \vdash A \Uparrow B}{\Gamma \vdash \equiv_A \Uparrow \equiv_B} \qquad \textsc{ReflexivityApp}\ \frac{\Gamma \vdash A \Uparrow B}{\Gamma \vdash \mathrm{reflexivity}(\equiv_A) \Uparrow \mathrm{reflexivity}(\equiv_B)}$$

$$\textsc{EqRewrite}\ \frac{\Gamma \Uparrow \Gamma' \qquad \Gamma \vdash A \Uparrow B \qquad \Gamma \vdash x \Uparrow x' \qquad \Gamma \vdash P \Uparrow P' \qquad \Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash y \Uparrow y' \qquad \Gamma \vdash e \Uparrow e'}{\Gamma \vdash \mathrm{@eq\_rect}(A, x, P, f, y, e) \Uparrow [\![\mathrm{LiftRewrite}_{\Gamma'}(B, x', P', f', y', e')]\!]}$$

$$\textsc{SetoidRewrite}\ \frac{\Gamma \Uparrow \Gamma' \qquad \Gamma \vdash A \Uparrow B \qquad \Gamma \vdash x \Uparrow x' \qquad \Gamma \vdash y \Uparrow y' \qquad \Gamma \vdash e \Uparrow e' \qquad \Gamma \vdash g \Uparrow g' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash \mathrm{StartRewrite}(A, x, y, e, g, t)) \Uparrow [\![\mathrm{LiftSetoidRewrite}_{\Gamma'}(B, x', y', e', g', t')]\!]}$$

Figure 4. The additional rules needed for repairing across setoid equivalences. There are two mutually defined judgements: one to repair environments (top) and one to repair terms (bottom).

4.2 Extended Transformation

We extend this transformation to work for setoid equivalences as well. To do this, we adapt our transformation to handle the changes in how equality works (Figure 4). We handle equivalence and its proofs in three cases: its type, its construction by reflexivity, and its elimination by rewriting.

Types. To define repair, we provide custom equivalence relations for any type we wish to consider as a setoid. This is done by providing three terms:

1. the type `C : Type`
2. a binary relation $\equiv_C$ `: C -> C -> Prop`
3. a proof that $\equiv_C$ is an equivalence relation

If the user does not supply these terms for some type, then $\equiv_C$ is be assumed to be equality. Then, if a type `C` repairs to `D`, all occurrences of $\equiv_C$ repair to $\equiv_D$ (by EquivApp).

Construction by Reflexivity. The next rule deals with constructions of equivalence relations by reflexivity. The proof supplied by the user that $\equiv_C$ is an equivalence relation contains a term for reflexivity, with type `∀ c : C,` $\equiv_C$ `c c`.

We denote this term `reflexivity(`$\equiv_C$`)`. If $\equiv_C$ is `@eq C`, this is simply `eq_refl C`. In either case, each `reflexivity(`$\equiv_C$`)` repairs to `reflexivity(`$\equiv_D$`)` (by ReflexivityApp).

Elimination by Rewriting. For rewriting, we split into two cases. When $\equiv_C$ is `@eq C`, then equality is Leibniz equality. That is, for any `P : C -> Type`, if `c1 = c2` then `P c1 -> P c2`. Thus, the eliminators (like `@eq_rect` in Coq) define term rewrites, with `P` defining where in the term rewrites take place. Our equivalence relations are not Leibniz, however, so we cannot directly translate this term. Instead, we assume we have an oracle $[\![-]\!]$ which can prove that a given rewrite,

denoted LiftRewrite$_\Gamma$(D, x, P, px, y, H), can be performed. We discuss how this oracle is implemented in Section 5. This oracle requires access to the environment Γ so that the oracle can refer to the repaired terms when discovering the rewrite proof. The rules LiftEmpty and LiftCons describe how the environment is transformed. Then, applications of `@eq_rect` are replaced with that proof (by EqRewrite).

If $\equiv_C$ is some other equivalence relation, the user can still rewrite, except those rewrites behave differently. We denote such a rewrite using `StartRewrite(C, x, y, e, g, t)`. Here, `x y : C`, `e : ` $\equiv_C$ `x y`, `g` is the type the user is rewriting, and `t` is the term the rewrite is applied to. This rewrite replaces every instance of `x` in the type of `t` with `y`. The SetoidRewrite rule assumes that our oracle $[\![-]\!]$ can perform a rewrite using the repaired data on the repaired term, which we denote by LiftSetoidRewrite$_\Gamma$(D, x, y, e, g t).

5 Implementation

We implement this extension to the transformation by extending the Pumpkin Pi Coq plugin (Section 5.1). Consistently with the original Pumpkin Pi, our implementation relies on some user annotations (Section 5.2), and places some other restrictions on the format terms can take (Section 5.3). Our implementation includes novel automation to dispatch proper proofs specific to setoids (Section 5.4).

5.1 Extending Pumpkin Pi

We extend the Pumpkin Pi Coq plugin directly. Plugins are a method of adding functionality to Coq. They are written in OCaml and can interact directly with the Coq internal codebase. Plugins can directly transform and produce terms; all terms that plugins produce are checked by Coq's type checker, and so cannot be ill typed.

Pumpkin Pi has various classes of proof repair transformations across type equivalences for which it has specialized automation. We add an additional class, termed setoid lifting, to support our extended transformation. This class mostly reuses the existing transformation, but implements the new rules from Figure 4. In all, our extension adds 1659 lines of code, 510 of which are dedicated to proper proof generation implemented in `propergen.ml`. The core extensions to the transformation can be found in `lift.ml`, `liftconfig.ml`, and `liftrules.ml` in the supplementary material.

In our extended configuration, the proof that $\equiv_C$ is an equivalence relation takes the form of instances of type classes `Equivalence` $\equiv_C$. This makes it possible to use Coq's setoid automation to implement the oracle that produces proofs of rewrites described in Section 4.1. Coq has a tactic, called `setoid_rewrite`, which attempts to perform rewriting by an equivalence relation. However, because equivalence relations are not generally Leibniz, we must *prove* for each function we define that the function is proper, as defined in Section 3, if we wish to rewrite under applications of that function. When we prove this, we instantiate the `Proper` type class. The `setoid_rewrite` tactic uses the `Proper` and `Equivalence` type class instances to search for proofs of rewrites, and thus we can use it as our oracle.

5.2 User Annotations

To perform repair, Pumpkin Pi requires that users annotate their proofs explicitly with components of the configuration. These annotations are required to identify parts of the configuration, thereby decoupling the undecidable part of proof repair (configuration inference) from the decidable part (the proof term transformation itself). We inherit this design decision for our extension, for which this applies to both the original configuration and to our extension to equivalences.

For the original configuration, we do not yet implement any new automation for the new class of changes we support to unify arbitrary subterms with applications of `DepConstr`, `DepElim`, η, and ι. Users of repair across setoid equivalences must apply these terms from the configuration directly in order for repair to work. In contrast, for some specific classes of changes implemented in the original Pumpkin Pi work, these annotations can sometimes be automatically inferred, lessening the burden of user annotation.

For the new configuration components corresponding to equivalence relations, we have some inference in simple cases. Specifically, if no relation is provided for a given type, our extension defaults to strict equality. Then, we can automatically infer annotations corresponding to applications of equality `@eq C`, as well as applications of reflexivity `@eq_refl C`. The same holds for rewrites that fully apply any of Coq's equality eliminators. However, this annotation inference cannot deal with unapplied instances of `@eq` and `@eq_refl` that may later be specialized to `C`, or equality eliminators that are not fully applied. Such terms are improperly annotated.

We do not yet have annotation inference for rewrites when the equivalence relation is user-specified and does *not* correspond to equality, as these setoid rewrites can manifest as long and complex subterms that are difficult to automatically identify. Instead, we supply a custom user annotation `START_REWRITE`, which the user must apply to their term before applying the rewrite. This constant takes two non-implicit arguments: the proof of equivalence by which the user is rewriting, and the type they are rewriting. We provide a custom tactic `rewrite_annotate` which the user can use in place of `rewrite` in proofs, which automatically performs annotation while rewriting.

5.3 Restrictions

Like the original Pumpkin Pi, we repair proof terms directly, and we do not directly repair terms involving pattern matching and recursion. Pumpkin Pi includes some automation to transform pattern matching and recursion to induction,

```
Theorem depRec (C : Type)
  (posP : ∀ (n : nat), C)
  (negSucP : ∀ (n : nat), C)
  (z : GZ) : C.
```

```
Theorem depElimProp (P : GZ -> Prop)
  `(p : Proper (GZ -> Prop) (eq_GZ ==> iff) P)
  (posP : ∀ (n : nat), P (depConstrPos n))
  (negSucP : ∀ (n : nat), P (depConstrNegSuc n))
  (z : GZ) : P z.
```

Figure 5. The types of the two eliminators we use in one of our case studies. The left has non-dependently typed output, but can eliminate into `Type`, while the right has dependently typed output but only eliminates into `Prop`. The right eliminator also requires a proof that the motive is proper as a function from the setoid (`GZ`, `eq_GZ`) to the setoid (`Prop`, `iff`).

which is likewise bundled in our extension. We inherit Pumpkin Pi's proof term decompiler, which makes it possible to get simple tactic proofs from repaired proof terms.

While the original proof repair work had a single dependent eliminator, we have multiple eliminators. One eliminates into the sort `Type`, but is purely nondependent. The other is dependent, but only eliminates into the sort `Prop`, and requires that the motive of the eliminator be proven to be proper, considering `Prop` as a setoid with `iff`. The types of two of these eliminators for one of our case studies are in Figure 5. We do this because Coq's setoid automation does not work for a dependently typed notion of an equivalence relation. Thus, we cannot use the setoid automation to perform rewrites on applications of functions we define with a dependently typed eliminator. For our `Prop`-sorted eliminator, this loss means that users cannot automatically perform rewrites on the *proofs* of propositions. In Coq, `Prop` is frequently treated as effectively proof irrelevant, so this loss is more acceptable.

In addition, our use of Coq's setoid automation is facilitated through the use of Coq's rewrite tactics. However, these tactics do not allow a rewrite which does not change the goal type, and so neither does our automation for repairing setoid rewrites. Furthermore, setoid rewrites do not allow specifying a motive `P`. To get around this, a user trying to prove `P x -> P y` can perform a substitution `P[z/y]`, where `z` is free in `P`, and then define `Q := fun z => P[z/y]`. Then, for another fresh variable `w`, the user can use `setoid_rewrite` to prove `H : ∀ (w : B), Q w x -> Q w y`, and recover the desired rewrite proof as `H y`.

5.4 Automating Proper Proofs

To perform rewrites on a term using Coq's setoid automation, it is necessary to prove that the functions in that term are proper. Thus, when repairing terms, it is potentially necessary to have proper proofs for every previously repaired function. We implement novel automation that helps prove many functions proper automatically. Our automation must fail in some cases, though, since proving a general function is proper is undecidable. To prove this, for any proposition `P : Prop`, define `f : bool -> Prop, f b = if b then True else P`. Generating a proof that `f` is proper, using `bool` as a setoid relating `true` and `false`, and `Prop` as a setoid with the relation `iff`, is equivalent to proving `iff P True`, which is undecidable.

Presently, our automation proves proper proofs automatically in two practical cases. First, when `f` is a composition of proper functions, we introduce hypotheses stating that all arguments are equivalent, and then rewrite by these hypotheses. This is the approach taken by Coq's `solve_proper` tactic, though we modify it slightly. Coq's `solve_proper` fails if any of its inputs do not appear in the body of the function because the rewrite for that argument will fail. To avoid this, we use the `try` tactical when rewriting. We also try both Coq's `rewrite` and `setoid_rewrite` tactics, while `solve_proper` only runs `setoid_rewrite`. While `setoid_rewrite` supports rewriting under binders in some instances, `rewrite` succeeds in some instances where `setoid_rewrite` fails.

Second, suppose that `f` is an application of the eliminator of some inductive type with a constant motive. Then, if all of the inductive case arguments provided are proper functions, we can prove that the eliminator is a proper function from its base cases to its output. To make this concrete, let `C` be a setoid with `eqC` the equivalence relation, fix some `P = fun _ => C`, and consider:

```
nat_rect P : ∀ (po : C)
  (ps : forall (n : nat) (pn : C), C) (n : nat), C
```

Then, we can automatically prove that:

```
Proper (eq ==> eqC ==> eqC) ps -> ∀ (n : nat),
  Proper (eqC ==> eqC) (fun po => nat_rect po ps n)
```

by induction on `n`. The base case holds by assumption via proper, and the inductive case holds by `ps` being proper.

This can be done in general for inductive types. Thus, our automation checks if `f` is an eliminator at the top level, and if so generates and tries to prove such a proper goal. If it succeeds, the automation attempts to show that the inductive cases (in the above example, `ps`) are proper using the rewriting strategy described above. In our case studies, this solves the proper goals for `add_posGZ` and `add_negsucGZ`.

6 Case Studies

We use our extended version of Pumpkin Pi on three case studies that use quotient type equivalences. First, we conduct repair between two representations of the integers (Section 6.1). Second, we study two common implementations of

```
Inductive Z : Set :=
| pos : nat -> Z
| negsuc : nat -> Z.
Definition GZ := nat * nat.
Definition eq_GZ (z1 z2 : GZ) := match z1, z2 with
| (a1, a2), (b1, b2) => a1 + b2 = a2 + b1
end.
```

Figure 6. The types of our integer representations. We provide an instance of `Equivalence eq_GZ` in the case study.

the queue data structure, and how we can repair from one to the other (Section 6.2). Third, we repair between dense and sparse representations of polynomials with natural number coefficients (Section 6.3). All of the case study examples can be found in more detail in the supplementary material.

6.1 Adding, Fast and Slow

Our first case study is mathematically motivated. We consider a change in the type representing 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. Our Coq proofs for this case study can be found in `grothendieck_int_equivalence_repair_tool.v`.

Types & Configuration. Our first representation, `Z`, is based on the default implementation of the integers in Cubical Agda: two copies of $\mathbb{N}$ glued back-to-back. We will repair functions and proofs about `Z` to use a representation that may be more familiar to set theorists: viewing the integers as elements of $\mathbb{N} \times \mathbb{N}/\sim$, where $(x_1, x_2) \sim (y_1, y_2) \iff x_1 + y_2 = x_2 + y_1$. We call the resulting type `GZ`, and the equivalence relation `eq_GZ`. The definitions are in Figure 6. `Z` and `GZ` are setoid equivalent using the definition from Section 3, by mapping `pos n` to `(n, 0)` and `negsuc n` to `(0, S n)`.

We next decompose our isomorphism into a repair configuration consisting of dependent constructors, dependent eliminators, and ι-reduction rules for both types. The configuration differs from those found in the original Pumpkin Pi examples in that there are two eliminators (Figure 5 from Section 5.3): `depRec` for eliminating into nondependent types, and `depElimProp` for eliminating into dependent types that reside in `Prop`. Furthermore, `depElimProp` on `GZ` has an extra proof obligation: the motive `P : GZ -> Prop` must be a proper function, where the sort `Prop` is viewed as the setoid (`Prop`, `iff`). While in theory each of these eliminators need their own set of ι-reduction theorems, we provide them solely for `depRec`, since needing ι for `depElimProp` is not common and does not show up in our case study. The full configuration can be found in the supplementary material.

Function Repair. Next, we repair functions automatically using our extension of `Pumpkin Pi`. For example, we repair addition from `Z` (Figure 7, left) to `GZ` (Figure 7, right) by running the following command:

```
Lift Z GZ in addZ as addGZ.
```

Note that the call to `depRecZ` is directly replaced with one to `depRecGZ`, and the functions `add_posZ` and `add_negsucZ` are replaced with their repaired analogues. We also repair the successor and predecessor functions.

Our extension of Pumpkin Pi also automatically generates proofs that the repaired functions are proper. First, the user must prove that `depRec` is proper. Our extension uses this to generate proper proofs for functions applying `depRec`. For example, it generates the proof that addition is proper:

```
addGZ_proper : Proper (eq_GZ ==> eq_GZ ==> eq_GZ) addGZ.
```

Proof Repair. We automatically repair the proof `add0LZ`, which shows that `0` is a left identity for addition. Figure 8 shows the old and new theorem types. Note that, in the repaired theorem type, equality has been automatically replaced with an equivalence relation on the type, reflecting that the repaired theorem is about setoid equality instead of `eq`. In addition, we repair a proof that `0` is a right identity for addition. The original and repaired proofs can be found in the supplementary material.

For now, there is a bit of extra work related to proper proof generation when our proofs apply `depElimPropGZ` (as `add0LZ` does). In particular, while our automation generates proper proofs for some of the motives passed to `depElimPropGZ`, there is not yet a way to automatically supply those proofs to Pumpkin Pi so that it uses them when repairing applications of `depElimPropZ`. For now, we define a constant corresponding to the motive, which we then separately repair:

```
Lift Z GZ in add0LMotiveZ as add0LMotiveGZ.
```

The proof that this motive is proper is automatically generated. We then reconfigure Pumpkin Pi to use these applications of `depElimPropZ` and `depElimPropGZ` for its eliminators:

```
Definition appliedDepElimPropZ :=
  depElimPropZ add0LMotiveZ.
Definition appliedDepElimPropGZ :=
  depElimPropGZ add0LMotiveGZ add0LMotive_proper.
```

We use `appliedDepElimPropZ` in our proof of `add0LZ`, and can repair the term. Presently, an implementation bug only surfacing in this case study forces us to repair the arguments to `appliedDepElimPropZ` before reconfiguring. Future versions can avoid the need for this workaround by automatically supplying the necessary proper proofs to `depElimProp` terms.

Further Steps. As is, our repaired addition function is inefficient. It uses the repaired eliminator for `GZ`, which inherits the inductive structure of the eliminator for `Z`. This repaired eliminator is slow, as it internally computes a canonical representative of the equivalence class of the given element.

We adapt our repaired proofs to use the more efficient addition function defined in Figure 9. Consistently with prior

```
Definition addZ (z1 z2 : Z) : Z :=
  depRecZ Z
    (fun (p : nat) => add_posZ z1 p)
    (fun (p : nat) => add_negsucZ z1 p)
    z2.
```

```
Definition addGZ (z1 z2 : GZ) : GZ :=
  depRecGZ GZ
    (fun p : nat => add_posGZ z1 p)
    (fun p : nat => add_negsucGZ z1 p)
    z2.
```

Figure 7. The annotated definition of addition on `Z` (left, adapted from the Cubical Agda standard library), and the repaired function defined over `GZ` (right). We omit the definitions of `add_posZ` and `add_negsecZ`, which are also automatically repaired.

```
Theorem add0LZ : ∀ z : Z,
  z = addZ (depConstrZPos 0) z.
Theorem add0LGZ : ∀ z : GZ,
  eq_GZ z (addGZ (depConstrGZPos 0) z).
```

Figure 8. An addition identity whose proof we repaired.

```
Definition fastAddGZ (a b : GZ) := match b with
  | (b1, b2) => match a with
    | (a1, a2) => (a1 + b1, a2 + b2)
  end
end.
```

Figure 9. Our fast addition function on the repaired integers. The direct use of pattern matching is acceptable because this function is neither to be repaired nor a product of repair.

```
Definition OLQ := list A.
Definition TLQ := list A * list A.
Definition insOrder (q : TLQ) := match q with
| (l1, l2) => l1 ++ rev l2
end.
Definition eq_queue (q1 q2 : TLQ) :=
  insOrder q1 = insOrder q2.
```

Figure 10. One list queues and two list queues. We provide an instance of `Equivalence eq_queue` in the case study.

work in Pumpkin Pi, to move between slow and fast implementations we take an ad hoc approach. First, we prove that both implementations of addition are pointwise equivalent in our setoid. Then, we rewrite across this equality to get from proofs of repaired theorems defined over slow addition to proofs of corresponding theorems defined over fast addition. For example, we use this methodology to translate the proof of `add0LGZ` into the proof of theorem:

```
Theorem fastAdd0LGZ : ∀ (z : GZ),
  eq_GZ z (fastAddGZ (depConstrGZPos 0) z).
```

using only one rewrite by `addEqualFastAdd`.

6.2 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. With our extensions to Pumpkin Pi, we can express this using setoids. Our proofs for this case study can be found in `two_list_queue_equivalence_repair_tool.v`.

Types & Configuration. Our first implementation `OLQ` represents queues using a single list. Elements enqueue at the front of the list and dequeue from the back of the list. This is simple, but the dequeue operation runs in linear time. Our second implementation `TLQ` uses a two list representation of queues. Elements enqueue at the front of the first list, and dequeue from the front of the second list, reversing the first list onto the second when the second is empty. This defines an amortized constant time dequeue operation.

Each two list queue `(l1, l2)` corresponds to the queue `l1 ++ (rev l2)`, but multiple two list queues correspond to a single one list queue. Thus, we use the equivalence relation `(l1, l2)` ∼ `(l3, l4)` ⟺ `l1 ++ (rev l2) = l3 ++ (rev l4)` and consider `TLQ` as a setoid. These two types are setoid equivalent along the expected correspondence, which lets us define the repair configuration. See Figure 10 for the types, and the supplementary material for the repair configuration.

Function Repair. We now use our extension to repair functions across this change. We provide the standard queue API by repairing `enqueueOLQ` and `dequeueOLQ`, as well as the helper functions `dequeueHelpOLQ` and `returnOrEnqTLQ`.

In the previous case study, our automation succeeded in generating every function's proper proof. This time, however, while the proofs that `enqueueTLQ` and `dequeueTLQ` are proper are generated automatically, the proofs that `dequeueHelpTLQ` and `returnOrEnqTLQ` are proper fail to generate and needed to be supplied manually. Also, we need to define multiple equivalence relations, since the return type of `dequeueTLQ` is `option (TLQ * A)` and our automation does not yet automatically lift the equivalence over `TLQ` to types including `TLQ`. The user must provide these equivalences to Pumpkin Pi.

```
Definition CLPoly := list nat.
Definition eq_CLPoly (l1 l2 : CLPoly) :=
  removeLeadingZeros l1 = removeLeadingZeros l2.
Definition CEPPoly := list (nat * nat).
Definition eq_CEPPoly (p1 p2 : CEPPoly) :=
  forall (exp : nat), coeff p1 exp = coeff p2 exp.
```

Figure 11. Definitions and equivalence relations for `CLPoly` and `CEPPoly`. `removeLeadingZeros l` removes leading zeros from `l`, while `coeff p exp` is the `n`th degree coefficient of `p`.

Proof Repair. We prove a theorem `dequeueEnqueueOLQ` stating that enqueue and dequeue commute in the expected way, using `returnOrEnq` as a helper function. We repair this automatically, with none of the workarounds from the previous case study, since the proof of `dequeueEnqueueOLQ` is defined using the `option_rect` rather than `depElimProp`. We also repair the proof of `dequeueEmptyOLQ`, providing an algebraic specification for the repaired datatype.

Further Steps. As in the previous case study, we have finished repairing proofs, but our repaired implementation of dequeue is inefficient. Thus, we implement the fast version of dequeue descirbed earlier, and follow the same methodology as in the previous case study to port our repaired proofs to use this fast version. We first show pointwise equality of the functions, and then rewrite inside of our proofs, giving us repaired proofs about the fast dequeue function.

6.3 Polynomial Polynomials

For our third case study, we provide sparse and dense representations of univariate polynomials with natural number coefficients, both of which are represented using setoids. We implement and repair addition of polynomials and evaluation of polynomials on a natural number, as well as proofs that addition is commutative and that evaluation respects addition. Our proofs can be found in `polynomial.v`.

Types & Configuration. Our first representation of polynomials, `CLPoly` (short for coefficient list polynomial), is as lists of natural numbers. The members of the list are the coefficients of the polynomial in order of decreasing degree. Two `CLPoly`s are equivalent if they are equal after removing leading zeros. Our second representation, `CEPPoly` (short for coefficient-exponent pair polynomial), is lists of pairs of natural numbers. Each pair `(c, exp)` represents a monomial in a sum, with `c` the coefficient and `exp` the exponent of the monomial, with duplicate exponents allowed. Two members of `CEPPoly` are equivalent if the polynomial they represent has the same coefficients. Both representations are in Figure 11.

Notice that every equivalence class in either of the setoids uniquely defines exactly one polynomial. Thus, the isomorphism between these setoids maps each member of the class representing polynomial p in one setoid to a member of the class representing polynomial p in the other setoid.

Next, we define a configuration for repair. Pumpkin Pi's configurations are based on decomposing an equivalence between inductive types, but here, both types are setoids. We choose an inductive type equivalent to both of our setoids, and the configuration components for our setoids have the structure of that type. For this, we choose the type of canonical representatives of `CLPoly`, the sigma type:

```
{l : list nat | noLeadingZeros l}
```

The types of `depConstr` for both `CLPoly` and `CEPPoly` take the shape of the constructor of this type, as do the types of the eliminators and ι-reduction rules. We show this for part of our configuration for `CLPoly` in Figure 12.

Function Repair. We use `CLPoly` as our source setoid and `CEPPoly` as our target setoid. We define `add` and `eval`, representing addition and evaluation, explicitly annotated with the components of the configuration. We automatically repair these functions using our extension to Pumpkin Pi. Our extension automatically proves that `eval` is proper, but fails to show `add` is proper, so we do so manually.

Proof Repair. We repair proofs of `addComm`, which states that `add` is commutative, and `evalRespectsAdd`, which states `eval (add p1 p2)n = (eval p1 n)+ (eval p2 n)`. Both use `depElimProp`, so we reconfigure Pumpkin Pi to use specialized versions of `depElimProp`, as we did in the first case study. From there, both proofs repair automatically. Neither of these proofs use rewriting. Thus, to demonstrate lifting setoid rewrites to setoid rewrites, we also repair three simple proofs about polynomials specifically using setoid rewriting.

Further Steps. The constructor for `CLPoly`, seen in Figure 12, takes an argument of type `opaque_list`. Because `CLPoly` is internally `list nat`, Pumpkin Pi will attempt to repair *all* instances of `list nat`. However, we define functions over `list nat` which we do not want to repair. Thus, we define an alias `opaque_list` for `list nat` and tell Pumpkin Pi not to repair `opaque_list` or any functions and theorems that should not be repaired. This behavior and our strategy for dealing with it are inherited from Pumpkin Pi.

7 Discussion: Grounding in Cubical

We extended Pumpkin Pi with additional automation for setoids in order to support a new class of changes corresponding to quotient type equivalences. This is in spirit with Pumpkin Pi's existing philosophy, as Pumpkin Pi is a tool that is external to Coq. But, to make sense of Pumpkin Pi's behavior, we need a univalent metatheory. This applies for both the existing behavior and for our extensions.

This discussion goes further than prior work on Pumpkin Pi, grounding the meaning not just in an uninstantiated univalent metatheory, but also directly in Cubical Agda. In

```
Definition depConstr (l : opaque_list) (p : noLeadingZeros l) : CLPoly.
Definition depRec (C : Type) (X : forall (l : opaque_list) (p : noLeadingZeros l), C) (p : CLPoly) : C.
```

Figure 12. The types for `depConstr` and `depRec` for `CLPoly`, where `opaque_list` is an alias for `list nat`.

particular, we demonstrate how setoids approximate quotient types (Section 7.1) as well the tradeoffs between those representations. For the original Pumpkin Pi work, this lets us formalize, for the first time ever, some of the correctness criteria for repair (Section 7.2). Finally, it reveals that the ad hoc treatment of replacing slow implementations with fast implementations in Coq could become much more grounded as well (Section 7.3). To concretize this discussion, we manually reimplement two case studies from Section 6 in Cubical Agda, which we use throughout this section as examples.

7.1 Grounding Setoids

Because Coq lacks quotient types, our implementation uses setoids to represent quotients. This incurred additional implementation overhead. To understand this overhead and where it manifests, we revisit the same proof repair problem manually in Cubical Agda, which has native quotient types.

In Cubical Agda, set quotient types are encoded internally as higher inductive types:

```
data _/_ (A : Type) (R : A → A → Type) : Type
  [_] : (a : A) → A / R
  eq/: (a₁ a₂: A) → (r: R a₁ a₂) → [ a₁ ] ≡ [ a₂ ]
  squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

For example, consider the equivalence between the two integer types from Figure 6 in Section 6.1. In Cubical Agda, the type `GZ` is defined as $(\mathbb{N} \times \mathbb{N})/\sim$, where $\sim$ represents our equivalence relation. The resultant type is the quotient of $\mathbb{N}$ by $\sim$. Since quotient types are internal, a quotient type equivalence in Cubical Agda is just a normal type equivalence. See `grothendieck_int_equivalence.agda`.

In Agda, we must prove that our `depElims` are proper functions upon definition; then, all functions defined using them are automatically proper. We may skip this proof when defining `depRec` when using setoids in Coq, but the burden appears when we repair proofs. When using setoids, we can only rewrite under functions we have proven proper. By proving that `depRec` is proper, our automation handles many of these proof obligations, but not every case as in Agda.

More consequentially, the additional rules to repair equivalence relations from Figure 4 in Section 4 are not needed at all in a restricted set fragment of Cubical Agda. The transformation from Figure 3 in Section 4 works, assuming the same annotations as in Coq, noting that we cannot reuse any of Pumpkin Pi's automation in Cubical Agda). There is only one additional restriction: every motive `P : GZ → Set` comes with the requirement that `((x : GZ)→ isSet (P x))`, to ensure that we actually do stay in the set fragment.

To demonstrate this, we manually followed the transformation to repair the functions and proofs from the first two case studies, which can be found in the supplementary materials in files `grothendieck_int_equivalence.agda` and `two_list_queue_equivalence.agda` respectively.

7.2 Grounding Correctness

We would like to prove that our functions and proofs were repaired correctly. As noted in previous work [19, 21], stating this relies on a univalent notion of equality: we want to know that terms behave the same up to the equivalence we repair across. Without assuming univalence as an axiom in Coq (which goes against the very philosophy of Pumpkin Pi), we cannot state these correctness criteria internally.

This was done metatheoretically in a univalent type theory for the Pumpkin Pi transformation that we build off of, but it was not proven for any particular example at all. In Cubical Agda, heterogeneous path equalities in the form of `PathP` types give 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 equivalence for the first time.

Some of these theorems are generic across all types. For example, we internally prove correctness of the Lam rule of the transformation from Figure 3. Other rules are stated specifying the types being repaired. For example, we proved 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` relates `inl n` and `inr n`. 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 and, for the first time, proved that it held. For details, see `equivalence_int_abs.agda` in the supplementary material, or Figure 13 and Figure 14 in the appendix.

Even when we can internally prove individual rules of the transformation correct, it can be challenging to actually *compose* those proofs to get proofs of the correctness of particular repaired proofs. Our correctness theorem is stated in terms of `PathP`s, but showing that two `PathP`s are equal requires reasoning about the equality of equality proofs between types. This means entering the fragment of cubical that is *proof relevant*: identity proofs need not be unique, and specific proofs may be needed for goals. Concretely, we were able to compose the correctness proofs to show the correctness of repaired *functions*, like addition. But we could not

always do the same for many more interesting *proofs*, like commutativity of addition, even after proving full correctness of the eliminator. This is why we say that we are able to show just *some* correctness proofs internally in Cubical Agda, even though in theory, it should be possible to show *all* of them. Fully reckoning with proof relevance so that correctness proofs compose correctly requires additional work. For this reason, we have yet to successfully implement internal correctness proofs in the Cubical Agda versions of the case studies we discussed in Section 6.

7.3 Grounding Fast and Slow

Recall that two of the case studies in Coq used an ad hoc process for moving between fast and slow implementations of repaired functions. This is due to the lack of functional extensionality and internal transport in Coq. In Cubical Agda, functional extensionality and transport are both theorems. This makes the process of moving between slow and fast implementations of repaired functions straightforward, grounding it in computational notions of functional extensionality and transport. It also shows that even in a world with external proof repair, internal transport can still be quite useful.

Consider the addition example from Section 6.1. In Cubical Agda, we call our fast addition function `addGZ'`, and our slow repaired addition function `_+GZ_`. Using functional extensionality, we can show that these two functions are path-equal to each other. Then, to obtain proofs for our fast addition function, we substitute the fast function for the repaired function in the repaired proofs using transport. The proofs for this can be found in `grothendieck_int_equivalence.agda` in the supplementary material, and similar proofs for the two list queue example are in `two_list_queue_equivalence.agda`.

8 Related Work

Proof Repair. This work extends the Pumpkin Pi [21] proof repair transformation and Coq plugin to support quotient type equivalences, a class of changes previously not supported. Pumpkin Pi has some more mature automation for other classes of changes, like automatic search for configurations, that we do not yet extend to work for quotient type equivalences. Proof repair was first introduced in parallel by Ringer et al. [23] and Robert [24], with strong influence from program repair [18]. Sisyphus [15] is a recent proof repair tool that, like our work, can handle changes in behavior (using a mix of dynamic and static techniques). However, Sisyphus repairs proofs of imperative OCaml programs verified in Coq using an embedded separation logic, whereas our work repairs proofs that are written in Coq directly.

Univalent Foundations. Parts of this project are grounded in Cubical Agda, and parts assume a univalent metatheory. Cubical Agda is an implementation of cubical type theory [30]. Cubical type theory [3, 8, 11] was developed to give a constructive account of the univalence axiom. When working in Cubical Agda, we are able to state and prove internal correctness of parts 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 [29], which presents the univalence axiom non-constructively. Homotopy type theory has additionally been implemented in Coq as the HoTT library [5].

Proof Reuse and Transfer. Proof repair is an instance of proof reuse, which seeks to use existing proofs in new goals. Other work in proof reuse includes CoqEAL [10] which uses refinement relations to verify properties of efficient functions using proofs on functions that are easy to reason about. CoqEAL can handle relations more general than equivalences, but does not include support for porting proofs across those changes. In Isabelle/HOL, the Transfer package [16] uses automation to transfer proofs between types. Both approaches require the source and target type to remain in the codebase, unlike proof repair. A complementary approach is to design proofs to be more reusable or more robust to changes from the start [7, 14, 32]. More work on proof reuse can be found in the QED at Large [20] survey of proof engineering.

Work has been done to implement transfer tools in Coq that approximate or externally implement automation corresponding to univalent transport. Tabareau et al. [26] defines univalent parametricity, which allows transport of a restricted class of functions and theorems. Univalent parametricity implements an ad hoc form of transport that only sometimes requires functional extensionality, and in many cases is axiom-free. It also includes a form of type-directed search to transport terms by way of type classes, something that proof repair tools like Pumpkin Pi and our extension still lack. Subsequent work introduces a white-box transformation [27] similar to the repair transformation from Pumpkin Pi, which Ringer [19] describes as developed in parallel with mutual influence. None of these support quotient type equivalences like our work does, though it is possible that by leaning further on functional extensionality, one could use these tools with quotient type equivalences.

More recent work called Trocq [9] implements an external form of transfer for Coq that directly supports relations more general than equivalences, like CoqEAL, but also supports proofs. Like Pumpkin Pi, Trocq goes out of its way to avoid depending on axioms like univalence and functional extensionality. Trocq's motivation of supporting transfer of proofs across relations more general than equivalences is similar to the motivation of our extensions to Pumpkin Pi, with two differences: (1) our work supports a more limited class of relations that can be described as equivalences between quotient types, and (2) for that class of changes, by extending Pumpkin Pi's proof term transformation, our work makes it possible to remove the old version of a type after applying repair. The major benefit of our tool relative to Trocq comes from (2)—while all proof repair tools implement a

kind of transfer, not all transfer methods implement repair, and Trocq does not appear to implement repair.

Quotients and Equivalences. Our work uses quotient types to expand the scope of proof repair. Quotient types exist in other proofs assistants besides Cubical Agda, like Isabelle/HOL [17, 31], as well as Lean [4] by way of axioms. Bortin and Lüth [6] use quotient types to construct theories in Isabelle, like multisets and finite sets as quotients of lists. Coq does not have quotient types, but it does have setoids [25], which do not explicitly form equivalence classes like quotients do. Setoid type theory uses a setoid model to justify the axioms needed to represent quotient types [1]. We draw on internal quotient types for our grounding in Cubical Agda, and we draw on setoids for our work in Coq.

Our idea for extending proof repair using quotient type equivalences to begin with comes from Angiuli et al. [2], which shows that certain relations more general than equivalences can be represented this way. The first example present in that paper is the queue example which we have also studied in our work. Because that work uses transport, it 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. In doing so, we extend proof repair to support a new class of changes described as missing from the original Pumpkin Pi work [21].

9 Conclusions & Future Work

We extended Pumpkin Pi to support changes represented by quotient type equivalences, adding more expressive power. The key challenge we overcame was supporting quotient types in a proof repair algorithm built for a type theory that does not even have quotient types to begin with. We addressed this by representing quotient types using setoids, extending the Pumpkin Pi algorithm and implementation to repair proofs about equivalence relations, and adding new automation to dispatch newly generated proof obligations. Our extension demonstrated success on three case studies not supported by the original Pumpkin Pi.

Going forward, we would like to reconcile our grounding in Cubical Agda with our implementation in Coq, ideally getting the best of both worlds—something we could do easily if only we knew of a univalent language with both native quotient types and a strong metalanguage for building automation. We also hope to continue to improve our extension's automation and usability. And we hope to look at other kinds of types and relations that can be expressed even when the type theory lacks them, as quotient types can be by way of setoids. We hope all of this will open the door to proof repair for more sophisticated classes of changes.

References

[1] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. 2019. Setoid Type Theory—A Syntactic Translation. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Springer International Publishing, Cham, 155–196.

[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] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2017. Theorem Proving in Lean. https://leanprover.github.io/theorem_proving_in_lean/index.html

[5] 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

[6] 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.

[7] Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

[8] 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

[9] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Trocq: proof transfer for free, with or without univalence. In *European Symposium on Programming*. Springer, 239–268.

[10] 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.

[11] 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

[12] 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

[13] 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

[14] Benjamin Delaware, William Cook, and Don Batory. 2011. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2048066.2048113

[15] 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

[16] 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.

[17] Isabelle Development Team. 1994-2024. Isabelle. http://isabelle.in.tum.de

[18] Martin Monperrus. 2017. Automatic Software Repair: a Bibliography. *ACM Computing Surveys* (2017). https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf

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

[20] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

[21] 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.

[22] 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

[23] 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

[24] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs*. Ph. D. Dissertation. UC San Diego.

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

[26] 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

[27] 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

[28] 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.

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

[30] 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

[31] Makarius Wenzel et al. 2004. The Isabelle/Isar reference manual.

[32] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

A Types of theorems that repair of eliminators is correct

In Section 7.2, we discuss proving theorem stating the correctness condition from the Pumpkin Pi paper. In Figure 13, we show that the Lam rule is correct. In Figure 14, the type of our proof that repair of a dependent eliminator is correct is presented. 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. The term witnessing this type can be found in `equivalence_int_abs.agda` in the supplementary material.

```
  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'
```

Figure 13. A theorem showing that the Lam rule is correct. Here, `i`, `i0`, and `i1` are terms of the interval type, which is a primitive construct in cubical used to define path equalities. The rest is analogous to Figure 3: `f` is the left function in the rule, `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 14. The theorem stating the correctness condition for the repaired dependent eliminator for a simple example type, which has been proven internally in Cubical Agda. Here, `depConstr0OK` and `depConstrSOK` are the correctness proofs of the repaired constructors, also proven internally.