Proof Repair across Quotient Type Equivalences

COSMO VIOLA, University of Illinois Urbana-Champaign, USA
MAX FAN, Cornell University, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Proofs in proof assistants like Rocq can be brittle, breaking easily in response to changes. To address this, recent work introduced an algorithm and tool in Rocq 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 implementations of data structures.

We extend this algorithm and tool to support quotient type equivalences in Rocq. Notably, since Rocq lacks quotient types entirely, our extensions use Rocq's setoid machinery in place of quotients. 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 demonstrate our extensions on proof repair case studies for previously unsupported changes. We also perform manual proof repair in Cubical Agda, a language with a univalent metatheory, which allows us to construct the first ever internal proofs of correctness for proof repair.

CCS Concepts: • **Software and its engineering** → **Software evolution**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Proof Repair, Cubical Agda, Rocq, Quotient Types

ACM Reference Format:
Cosmo Viola, Max Fan, and Talia Ringer. 2025. Proof Repair across Quotient Type Equivalences. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 386 (October 2025), 26 pages. https://doi.org/10.1145/3763164

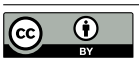
1 Introduction

Writing formal proofs in proof assistants like Rocq[1], 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 [29], and that it can be challenging to deal with even for experts.

Proof repair [26] aims to simplify this process by introducing algorithms and tools that fix formal proofs in response to breaking changes. Given an existing type A and some set of functions and theorems on that type, as well a new type B, proof repair seeks to generate the equivalent functions and theorems defined on the type B. Proof repair further requires that the new functions and proofs

[1]Formerly known as Coq.

Authors' Contact Information: Cosmo Viola, University of Illinois Urbana-Champaign, USA; Max Fan, Cornell University, USA; Talia Ringer, University of Illinois Urbana-Champaign, USA.



ACM 2475-1421/2025/10-ART386
https://doi.org/10.1145/3763164

on B make no reference to the original functions and proofs on A, which distinguishes it from the more general *proof transfer*.

Prior work introduced Pumpkin Pi, a Rocq plugin that performs proof repair across changes in datatypes that can be described by type equivalences [28]. In this work, we extend proof repair to support equivalences between *quotient types* (Section 2). Recent work by Angiuli et al. [3] in Cubical Agda showed that certain relations describing changes in behavior can be adjusted to equivalences between higher inductive types. One specific example presented in that work uses two queue representations. The first, one list queues, enqueue elements to the front of the list and dequeue elements from the back of the list. The second, two list queues, use a pair of lists. Elements are enqueued onto the front of the first list. When dequeuing, if the second list is empty, the first list is reversed onto the second, and then in any case the front element of the second list is removed. These types are not equivalent in any natural way, because multiple two list queues correspond to a single one list queue: for instance, the one list queue `l` corresponds to both `(l, [])` and `([], rev l)`. Angiuli et al. use a higher inductive type to construct the quotient type of two list queues so that, in the resulting quotient type, `(l, [])` and `([], rev l)` are in the same equivalence class, which makes the types equivalent. They are then able to perform proof transfer across that equivalence.

We wish to use this same approach to implement *proof repair* instead of *proof transfer*. Unlike Cubical Agda, Rocq 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 replace quotient types by using Rocq's setoid machinery, and we replace quotient type equivalences with setoid equivalences (Section 2). We then extend the proof transformation to support the newly generated equality proof obligations (Section 3).

By extending Pumpkin Pi to setoids, users gain both expressive power and efficiency. Prior to this work, Pumpkin Pi could capture a similar approach using sigma types instead of setoids. To do this, one would choose a canonical element for each equivalence class in the quotient, and then use the subtype of these elements instead of a quotient or setoid. However, that approach has severe drawbacks. In the two list queue example above, when using both quotient and setoids representations, the multiple representatives of each class allow for an amortized constant time dequeue operation. The subtype representation instead only allows for a linear time dequeue operation.

We implement this algorithm by way of an extension to the implementation of Pumpkin Pi (Section 4). 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 5). Finally, we define correctness for proof repair and construct internal proofs of correct repair in Cubical Agda (Section 6). 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
(5) the first **construction** of internal proofs of correct repair.

Our code is available on Github, and we have packaged the extension, our case studies, and all the tools required to run them into an artifact [41] for this paper.[2]

2 Quotients and Setoids

Our goal is to extend Pumpkin Pi to perform repair between types equivalent up to quotients. To begin this task, we define quotient types [21]:

Definition 2.1. A quotient type `A / eqA` is defined by a type `A` and an equivalence relation `eqA : A → A → Prop`. `A / eqA` has one constructor, `[ · ] : A → A / eqA`, and its elements are called the *equivalence classes* of the elements of `A`. `A / eqA` has the additional property that, for `a, b : A`, `[ a ] = [ b ]` if `eqA a b` is inhabited. Eliminating an element `q : A / eqA` yields the underlying `a : A` from which it was constructed. However, the user must prove that the result of the computation that is being defined is equal for any `a, b : A` with `[ a ] = [ b ]`.

To give a concrete example, consider the quotient of $\mathbb{N}$ by the equivalence relation `eq`$\mathbb{N}$, where `eq`$\mathbb{N}$ `n m` is inhabited if `n` and `m` have the same parity. The resulting type, denoted by mathematicians as $\mathbb{N}/2$, has two elements, `[ 0 ]` and `[ 1 ]`; any other application of `[ · ]` produces a term equal to one of these. For $\mathbb{N}/2$, eliminating `[ 0 ]` and eliminating `[ 2 ]` must provably result in equal terms.

Quotient constructions have historically been challenging to implement directly in constructive logic. Because of the equality property mentioned above, many type systems, including Rocq, do not support quotient types without the use of axioms. Adding quotient types can have nonobvious consequences as well. For instance, extending intensional Martin-Löf type theory with effective quotients and uniqueness of identity proofs is sufficient to prove the law of the excluded middle for types in the first universe if the type theory has at least two universes [24]. A common alternative approach, introduced by Bishop [8] for constructive analysis and popularized in type theory by Hofmann [20, 21], is to use setoids.

Definition 2.2. A setoid is the pair `(A, eqA)`. When the equivalence relation `eqA` is obvious from context, we may also call `A` a setoid as shorthand.

Alternative definitions exist; see Barthe et al. [6] for a survey of these definitions. Setoids are a natural alternative when actually constructing a quotient, and in turn its equivalence classes, is unfeasible. The Rocq standard library provides both definitions for setoids and substantial automation for their use. Quotients, on the other hand, see no such support. Their implementation would require the addition of axioms, which block computation and are generally avoided by Rocq users when possible. Thus, in Rocq, using setoids in place of quotients is the norm [27, 36]. Furthermore, using axioms in a Rocq plugin forces users of that tool to adopt the axiom, so using axioms in plugins is doubly discouraged.

Notice that setoids have no special constructors, eliminators, or equality properties. An element of `A` is said to be an element of the setoid, and users of the setoid should compare elements using `eqA` instead of native equality. Because of this, rewriting is more difficult in setoids than in quotient types. Rewriting by an equality in a quotient type can be done using the equality eliminator, but rewriting cannot generally be done for arbitrary equivalence relations. To compensate for this, we need the notion of a proper function.

Definition 2.3. For setoids `(A, eqA)` and `(B, eqB)`, a function `f : A → B` is *proper* if, for all `a1, a2 : A`, `eqA a1 a2` implies that `eqB (f a1) (f a2)`. [32, 37]

[2]Throughout the paper, links to specific files in the Github source have been provided where relevant. These links are given as circled numbers, like ①.

Using our example, $\mathbb{N}/2$, `f : ℕ → ℕ` is proper if inputs to `f` of the same parity produce outputs of the same parity. The successor function would be one example of such a proper function. When a term is composed of proper functions, it becomes possible to construct proofs for rewriting by equivalence relations in much the same way we can rewrite by equality. Rocq provides automation in its standard library to support constructing these proofs. By using setoids in place of quotients, equivalence relations in place of equality, and proper functions in place of arbitrary functions, we mimic the functionality of quotients using setoids.

Our paper deals with repair across quotient type equivalences. To begin, we recall the definition of a type isomorphisms:

Definition 2.4. An isomorphism from type `A` to type `B` is a function `f : A → B` such that there exists `g : B → A` with

- `∀ (a : A), g (f a) = a`
- `∀ (b : B), f (g b) = b` [39]

Type equivalences are type isomorphisms with an additional adjoint property, which can be derived from any isomorphism. A quotient type equivalence is simply an equivalence between two quotient types. When using setoids to represent quotients, we must instead use a notion of a setoid equivalence:

Definition 2.5. A setoid equivalence between `(A, eqA)` and `(B, eqB)` is a function `f : A → B` such that there exists `g : B → A` satisfying the following properties:

- `f` and `g` are proper.
- `∀ (a : A), eqA (g (f a)) a`
- `∀ (b : B), eqB (f (g b)) b` [8]

The function `g` is said to be inverse to `f`.

Two setoids are equivalent if the quotient types they represent would be isomorphic. One example of a nontrivial setoid equivalence between the setoids `(ℕ, eqℕ)` and `(Bool, =)` is given by the function `isEven : ℕ → Bool`, sending even numbers to `true` and odd numbers to `false`. This function has inverse `g : Bool → ℕ` mapping `true` to 0 and `false` to 1. `f` is proper, since `isEven` respects parity, and `g` is automatically proper because `Bool` uses equality as its equivalence relation. The other conditions, `isEven (g b) = b` and `eqℕ (g (isEven n)) n`, are both satisfied, and thus this indeed forms a setoid equivalence.

3 Approach: Proof Term Transformation

To reiterate, our goal is to perform proof repair across quotient type equivalences. Because we are working in Rocq, which does not have native quotient types, we represent quotients using setoids. Thus, given two types `A` and `B`, each of which the user may choose to represent as either a type using standard equality or a setoid using some equivalence relation, we wish to turn terms defined over `A` into terms defined over `B` which no longer have any references to `A`.

To make the proof repair problem more concrete, we consider a specific example from the Pumpkin Pi paper. In Figure 1, we see two representations of the natural numbers. The first is a unary representation, while the second is a binary representation. Suppose that a user has begun proof development using the unary representation, implementing functions, such as addition, and theorems, such as that 0 is an identity for addition. Then, suppose that the user needs to switch to the binary representation later in development. Proof repair takes the functions and theorems defined over the unary natural numbers and turns them into functions defined over the binary natural numbers with the same behavior. These functions and theorems will make no reference to the unary natural numbers, so that the type can be deleted from the codebase entirely.

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

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

Fig. 1. The natural numbers, as represented in Rocq's standard library, in unary (left) and binary (right). The type `positive` represents all positive natural numbers. `xH` is one, `xO n` is appending a `0` to the right side of the binary representation of `n`, and `xI n` is appending a `1` to the right side of the binary representation of `n`. Then, `N` is either `0`, as `N0`, or a positive binary number.

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

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

Fig. 2. The grammar of CIC_ω from Pumpkin Pi [28], adapted from Timany and Jacobs [38]. The terms here are, in order: variables, sorts, dependent product types, functions, applications, inductive types, constructors, and eliminators.

We will review how Pumpkin Pi repairs proofs across type equivalences by directly transforming proof terms across those equivalences (Section 3.1). Then, we extend Pumpkin Pi's transformation to support setoid equivalences (Section 3.2).

3.1 Pumpkin Pi's Transformation

The Pumpkin Pi transformation that we extend operates over terms in the type theory of Rocq, the Calculus of Inductive Constructions (CIC_ω) [17]. CIC_ω extends the Calculus of Constructions [16] with inductive types. The grammar for CIC_ω is in Figure 2; the type rules are standard and omitted.

Pumpkin Pi implements proof repair over terms in CIC_ω 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 any equivalence between types `A` and `B` can be decomposed into separate components that talk only about `A` and only about `B` [26]. 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` [28].

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. `DepConstr` and `DepElim` are, respectively, constructors and eliminators for each type, termed *dependent constructors* and *dependent eliminators* internally by Pumpkin Pi (even in cases where those constructors and eliminators might be non-dependently typed). 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. Usually, then, for at least one of the types `A` or `B` `DepConstr` and `DepElim` will be distinct from the usual constructors and eliminators for that type.

```
Definition addNat (a b : nat) :=
  depElimNat
    (fun _ => nat -> nat)
    (fun b => b)
    (fun a IH b => depConstrNatSuc (IH b))
    a
    b.
```

```
Definition addN (a b : N) :=
  depElimN
    (fun _ => N -> N)
    (fun b => b)
    (fun a IH b => depConstrNSuc (IH b))
    a
    b.
```

Fig. 3. Repairing the addition function from unary (left) to binary (right) natural numbers.

Like the Pumpkin Pi paper [28], we will use repair between unary and binary natural numbers as an example. For the type of unary naturals in Figure 1, we can 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 for `N`. These constructors *must correspond across the equivalence*, so we must provide 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. Specifically, the successor constructor for `N` instead has type `positive -> N`. Likewise, we can choose the eliminator for `nat`, `nat_rect`, as the `depElim` for `nat`, and the dependent eliminator for `N` then must take the shape of `nat`'s eliminator:

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

We give a concrete example of repairing addition in Figure 3 ①.

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. However, if the inductive structure is different, as it is for `N`, this will be a propositional equality. As an example, the ι-reduction rule for `depConstrNSuc` has the following type:

```
Definition iotaNSuc (P : N -> Type) (PO : P 0)
  (PS : forall x : N, P x -> P (depConstrNSuc x))
  (n : N) (Q : P (depConstrNSuc n) -> Type) :
  Q (PS n (depElimN P PO PS n)) -> Q (depElimN P PO PS (depConstrNSuc n))
```

We may also use a version of this rule in proofs which runs in the reverse direction:

```
Definition iotaNSucRev (P : N -> Type) (PO : P 0)
  (PS : forall x : N, P x -> P (depConstrNSuc x))
  (n : N) (Q : P (depConstrNSuc n) -> Type) :
  Q (depElimN P PO PS (depConstrNSuc n)) -> Q (PS n (depElimN P PO PS n))
```

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

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

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

Eta
$$\frac{}{\Gamma \vdash \mathrm{Eta}(A) \Uparrow \mathrm{Eta}(B)}$$

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

Equivalence
$$\frac{}{\Gamma \vdash A \Uparrow B}$$

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

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

App
$$\frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'}$$

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

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

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

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

Fig. 4. Transformation for repair across $A \simeq B$ with configuration ((DepConstr, DepElim), (Eta, Iota)), from previous work [26]. Our work adapts and extends this transformation.

The last term, η, represents reducing the η-expansion of constructors applied to eliminators. However, we have yet to find an example where a nontrivial η is actually necessary. Thus, for the purposes of this paper, η will always be trivial and we will ignore it. [3]

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 4.2). Then, we follow the syntactic transformation outlined in Figure 4. We will discuss what it means for the repaired term to be *correct* in Section 6.

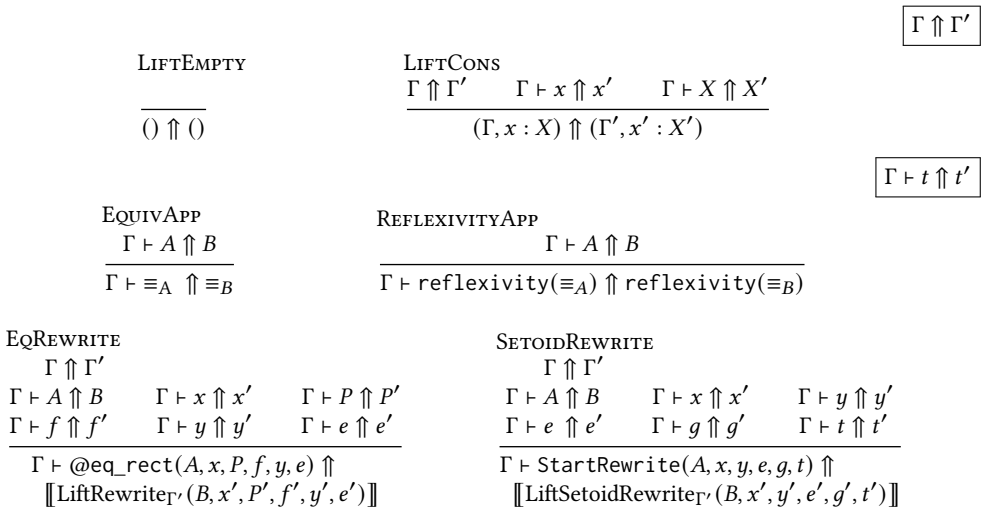


Fig. 5. The additional rules needed for repairing across setoid equivalences. There are two mutually defined judgments: one to repair environments (top) and one to repair terms (bottom).

3.2 Extended Transformation

We now 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 5). We handle equivalence and its proofs in three cases: its type, its construction by reflexivity, and its elimination by rewriting.

Types. To define an instance of a repair transformation using setoids, the user must provide custom equivalence relations for any type they 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, ReflexivityApp, 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 any case, each `reflexivity(`$\equiv_C$`)` repairs to `reflexivity(`$\equiv_D$`)` (by ReflexivityApp).

Elimination by Rewriting. For rewriting, we split into two cases depending on if $\equiv_C$ is `@eq C` or some other equivalence relation. When $\equiv_C$ is `@eq C`, then the eliminators (like `@eq_rect` in Rocq) define term rewrites, with `P` defining where in the term rewrites take place. Our equivalence relations do not support arbitrary term rewrites, however, so we cannot directly translate this

[3]The example of nontrivial η given in the Pumpkin Pi paper can be rewritten to instead have nontrivial ι and trivial η ②, and we are not aware of any cases where this cannot be done. Further, while the type of ι is always clearly defined, what η should be is unclear for many examples. For this reason, it's possible that η should be dropped entirely, and a configuration should simply include `DepConstr`, `DepElim`, and ι.

```
Inductive unit :=
  | tt.

Theorem rewrite_example :
  forall (x : unit), eq x tt -> eq x tt.
Proof.
  intros x H.
  rewrite H.
  reflexivity.
Qed.

fun (x : unit) (H : @eq unit x tt) =>
  @eq_ind_r unit tt
    (fun x0 : unit => @eq unit x0 tt)
    (reflexivity(@eq unit) tt) x H
```

```
Inductive unit_two :=
  | one
  | two.

Definition eq_unit_two (u1 u2 : unit_two) :
     Prop := True.

fun (x : unit_two)
  (H : eq_unit_two x one) =>
  [[ LiftRewrite_Γ'
    (unit_two, one,
    (fun x0 : unit_two =>
      eq_unit_two x0 one),
    (reflexivity(eq_unit_two) one), x, H)
  ]] : forall x : unit_two,
         eq_unit_two x one ->
         eq_unit_two x one
```

Fig. 6. A demonstration of repairing a rewrite in a proof. The left side is over the source type, which is simply the type with one element. We provide a sample Rocq tactic script which performs a rewrite, and below it is the term produced. The right side shows the repaired term over the target setoid, which is the type with two elements which are equal under the equivalence relation.

term. Instead, we assume we have an oracle $[\![-]\!]$ which can prove that a given rewrite, denoted $\text{LiftRewrite}_\Gamma(\texttt{D}, \texttt{x}, \texttt{P}, \texttt{px}, \texttt{y}, \texttt{H})$, can be performed. In Section 4, we will discuss how we implement this oracle without any additional axioms using Rocq's built in setoid rewriting automation. 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 the proof produced by the oracle (by EqRewrite). We show an example of repairing such a rewrite in Figure 6 ③.

If $\equiv_C$ is some other equivalence relation, the user can still perform rewrites, 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 $\text{LiftSetoidRewrite}_\Gamma(\texttt{D}, \texttt{x}, \texttt{y}, \texttt{e}, \texttt{g t})$. Then, instances of `StartRewrite(C, x, y, e, g, t)` repair to $[\![\text{LiftSetoidRewrite}_\Gamma(\texttt{D}, \texttt{x}, \texttt{y}, \texttt{e}, \texttt{g t})]\!]$ (by SetoidRewrite).

Importantly, this extension to Pumpkin Pi can only affect the trusted computing base of the proof system if the rewrite oracle has trust assumptions. Thus, by enforcing that the implementation of the oracle introduces no such assumptions, using this extension poses no risk of a soundness failure beyond that of the base type theory without any extensions.

4 Implementation

We implement this extension to the transformation by extending the Pumpkin Pi Rocq plugin (Section 4.1). Consistently with the original Pumpkin Pi, our implementation relies on some user annotations (Section 4.2), and places some other restrictions on the format terms can take (Section 4.3). Our implementation includes custom automation to dispatch properness proofs specific to setoids (Section 4.4).

4.1 Extending Pumpkin Pi

We extend the Pumpkin Pi Rocq plugin directly. Plugins are a method of adding functionality to Rocq. They are written in OCaml and can interact directly with the Rocq internal codebase. Plugins can directly transform and produce terms; all terms that plugins produce are checked by Rocq'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 5. Once the transformation rules were determined, the implementation was made straightforward thanks to Pumpkin Pi's extensibility; in all, our extension adds 1659 lines of code, 510 of which are dedicated to properness proof generation implemented in `propergen.ml`. The core extensions to the transformation can be found in `lift.ml`, ④ `liftconfig.ml`, ⑤ and `liftrules.ml` ⑥ in the artifact.

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 Rocq's setoid automation to implement the oracle that produces proofs of rewrites described in Section 3.2 with no axioms or other extensions to the trusted computing base. Rocq has a tactic, called `setoid_rewrite`, which attempts to perform rewriting by an equivalence relation. However, because rewriting arbitrary terms between equivalent elements is not possible, we must *prove* for each function we define that the function is proper, as defined in Section 2, if we wish to rewrite under applications of that function. In Rocq, we prove this by creating an instance of 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.

4.2 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 requirement for our extension, and thus both the original configuration and to our extension to equivalences must be annotated. However, we have implemented automation into Pumpkin Pi that generates many of the needed annotations for equivalence relations automatically.

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 Rocq's equality eliminators ⑤. 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 considered improperly annotated.

When an equivalence relation is provided for a type, we provide a custom tactic `rewrite_annotate` which the user can use in place of `rewrite` in proofs, which automatically performs annotation while rewriting. This inserts a custom annotation term `START_REWRITE` before applying the rewrite, which our extension looks for to identify and repair rewrites. This constant takes two non-implicit arguments: the proof of equivalence by which the user is rewriting, and the type they are rewriting ⑦.

```
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.
```

Fig. 7. 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).

```
Definition respectful_hetero
  (A B : Type)
  (C : A -> Type) (D : B -> Type)
  (R : A -> B -> Prop)
  (R' : forall (x : A) (y : B), C x -> D y -> Prop) :
    (forall x : A, C x) -> (forall x : B, D x) -> Prop :=
    fun f g => forall x y, R x y -> R' x y (f x) (g y).
```

Fig. 8. An equivalence relation from the Rocq standard library on dependently typed functions stating that the functions are respectful. R relates elements of their domains, and R' relates elements of their codomains. Notice that because the functions are dependently typed, R' is a family of equivalence relations, one for each pair of elements x : A, y : B. Specializing respectful_hetero to non-dependently typed functions is used to define the notion of a proper function we use in this work.

4.3 Restrictions

Pumpkin Pi does not directly repair terms involving pattern matching and recursion, instead requiring that proof terms it repairs are written using induction principles. As a result, our extension to Pumpkin Pi also has this limitation. However, Pumpkin Pi includes some automation to transform pattern matching and recursion in proof terms into induction, 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 depElim term on each side of the configuration, 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 7. We do this because Rocq's setoid automation does not work for a dependently typed notion of a proper function. Specifically, the term respectful_hetero, given in Figure 8, exists in the standard library [37] and could form the basis for such automation, but no such automation has been implemented in the standard library. 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 Rocq, Prop is frequently treated as effectively proof irrelevant, so this loss is more acceptable.

In addition, our use of Rocq's setoid automation is facilitated through the use of Rocq'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, the setoid rewrite

```
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.
```

Fig. 9. The types of our integer representations. We provide an instance of `Equivalence eq_GZ` for the case study in the artifact.

automation does not allow specifying a motive `P`. To perform setoid rewrites with a motive, 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 perform a `setoid_rewrite` to prove `H : ∀ (w : B), Q w x -> Q w y`, and recover the desired rewrite proof as `H y`. Our extension uses this methodology when repairing rewrites with a motive to setoids ④.

4.4 Automating Properness Proofs

To perform rewrites on a term using Rocq'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 properness proofs for every previously repaired function. We implement automation that helps prove many functions to be 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 constructs properness 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 Rocq's `solve_proper` tactic, though we modify it slightly. Rocq'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 Rocq'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, we will consider an example. Let (`C`, `eqC`) be a setoid, 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 P po ps n)
```

by induction on `n`. The base case holds by assumption via the definition of `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 ⑧. In our case studies, this is necessary for automatically solving some of the generated proper goals ⑨.

```
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.
```

Fig. 10. 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.

5 Case Studies

We use our extended version of Pumpkin Pi to automatically repair proofs on three case studies that use quotient type equivalences. First, we conduct repair between two representations of the integers (Section 5.1). Second, we study two common implementations of the queue data structure and how we can repair from one to the other (Section 5.2). Third, we repair between dense and sparse representations of polynomials with natural number coefficients (Section 5.3). In each of these case studies, the type to which we repair functions and theorems has a structure which enables efficient implementations of key operations. Our extension makes it possible to take advantage of this structure while using the repaired theorems, something impossible with prior versions of Pumpkin Pi. All of the case study examples can be found in more detail in the artifact.

5.1 Adding, Fast and Slow

Our first case study is a usage of quotients that is foundational in mathematics. We consider a change in the type representing integers from the inductive type found in many standard libraries to the standard quotient based representation. 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 Rocq implementation of this case study can be found in the artifact ⑨.

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 frequently used as a definition in mathematics: 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`[4], and the equivalence relation `eq_GZ`. The definitions are in Figure 9. `Z` and `GZ` are setoid equivalent using the definition from Section 2, 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 7 from Section 4.3): `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 artifact.

[4]In reference to Grothendieck, as this is the Grothendieck group of the natural numbers

```
Theorem add0LZ : ∀ z : Z,
  z = addZ (depConstrZPos 0) z.
```

```
Theorem add0LGZ : ∀ z : GZ,
  eq_GZ z (addGZ (depConstrGZPos 0) z).
```

Fig. 11. An addition identity whose proof we repaired automatically.

Function Repair. Next, we repair functions automatically using our extension of Pumpkin Pi. For example, we repair addition from `Z` (Figure 10, left) to `GZ` (Figure 10, 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 properness 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 11 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 artifact.

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 properness 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 properness 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 12. Consistently with prior work in Pumpkin Pi, to move between slow and fast implementations

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

Fig. 12. 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.
```

Fig. 13. One list queues and two list queues. We provide an instance of `Equivalence eq_queue` in the artifact.

we take an ad hoc approach. First, we prove that both implementations of addition are pointwise equivalent in our setoid, producing a term `addEqualFastAdd`. 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`.

5.2 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. [3], which showed that quotient types can be used 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 Rocq with the original framework. With our extensions to Pumpkin Pi, we can express this using setoids. Our Rocq implementation of this case study can be found in the artifact (10).

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)` $\iff$ `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 13 for the types, and the artifact for the repair configuration.

```
Definition enqueueOLQ (a : A) (q : OLQ) :
    OLQ :=
  depConstrOLQInsert a q.

Definition dequeueHelpOLQ (outer : A)
  (q : OLQ) (m : option (OLQ * A)) :
  option (OLQ * A) :=
@option_rect
  (OLQ * A)
  (fun _ => option (OLQ * A))
  (fun (p : (OLQ * A)) => Some
    (depConstrOLQInsert outer (fst p) ,
      (snd p)))
  (Some (depConstrOLQEmpty, outer))
  m.

Definition dequeueOLQ :
  OLQ -> option (OLQ * A) :=
  depRecOLQ (option (OLQ * A)) None
      dequeueHelpOLQ.
```

```
Definition enqueueTLQ (a : A) (q : TLQ) :
    TLQ :=
  depConstrTLQInsert a q.

Definition dequeueHelpTLQ (outer : A)
  (q : TLQ) (m : option (TLQ * A)) :
  option (TLQ * A) :=
@option_rect
  (TLQ * A)
  (fun _ => option (TLQ * A))
  (fun (p : (TLQ * A)) => Some
    (depConstrTLQInsert outer (fst p) ,
      (snd p)))
  (Some (depConstrTLQEmpty, outer))
  m.

Definition dequeueTLQ :
  TLQ -> option (TLQ * A) :=
  depRecTLQ (option (TLQ * A)) None
      dequeueHelpTLQ.
```

Fig. 14. Definitions for enqueue and dequeue over OLQ on the left, and their repaired versions over TLQ on the right.

```
Definition returnOrEnqOLQ (a : A)
  (m : option (OLQ * A)) : (OLQ * A) :=
  @option_rect
    (OLQ * A)
    (fun _ => prod OLQ A)
    (fun (p : (OLQ * A)) =>
      (enqueueOLQ a (fst p), snd p))
    (depConstrOLQEmpty, a)
    m.

Theorem dequeueEnqueue (a : A) (q : OLQ) :
  dequeueOLQ (enqueueOLQ a q)
  = Some (returnOrEnqOLQ a (dequeueOLQ q)).
```

```
Definition returnOrEnqTLQ : (a : A)
  (m : option (TLQ * A)) : (TLQ * A) :=
  @option_rect
    (TLQ * A)
    (fun _ => prod TLQ A)
    (fun (p : (TLQ * A)) =>
      (enqueueTLQ a (fst p), snd p))
    (depConstrTLQEmpty, a)
    m.

Theorem dequeueEnqueue (a : A) (q : TLQ) :
  eq_deq_ret
    (dequeueTLQ (enqueueTLQ a q))
    (Some
      (returnOrEnqTLQ a (dequeueTLQ q))).
```

Fig. 15. Main theorem relating dequeue and enqueue, stated for OLQ on the left and TLQ on the right. We repair this theorem's proof from OLQ to TLQ. Here, eq_deq_ret is the equivalence relation eq_queue lifted to the return type of dequeueTLQ.

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. The first three can be found in Figure 14, and the last in Figure 15.

```
Definition fastDequeueTLQ (q : TLQ) :=
  let (l1, l2) := q in match l1, l2 with
  | [] , [] => None
  | h1 :: t1 , [] =>
      Some (([] , tl (rev l1)), hd h1 (rev l1))
  | _ , h2 :: t2 => Some ((l1, t2), h2)
  end.
```

Fig. 16. Fast dequeue function for two list queues.

```
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.
```

Fig. 17. Definitions and equivalence relations for `CLPoly` and `CEPPoly`. `removeLeadingZeros l` removes leading zeros from `l`, while `coeff p exp` is the nth degree coefficient of `p`. We provide instances of `Equivalence eq_CLPoly` and `Equivalence eq_CEPPoly` in the artifact.

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

```
Definition depConstrCEPPoly
  (l : list nat)
  (p : noLeadingZeros l) : CEPPoly.
Definition depRecCEPPoly (C : Type)
  (X : forall (l : list nat)
    (p : noLeadingZeros l), C)
  (p : CEPPoly) : C.
```

Fig. 18. The types of `depConstr` and `depRec` for `CLPoly` on the left and `CEPPoly` on the right, where `opaque_list` is an alias for `list nat`.

In the previous case study, our automation succeeded in generating every function's properness proof. This time, however, while the proofs that `enqueueTLQ` and `dequeueTLQ` are proper are generated automatically, the properness proofs for `dequeueHelpTLQ` and `returnOrEnqTLQ` failed 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.

Proof Repair. We prove a theorem `dequeueEnqueueOLQ`, found in Figure 15, 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 described earlier, found in Figure 16, and can follow the same methodology as in the previous case study to

```
Definition addCLPoly (p1 p2 : CLPoly) :=
  depRecCLPoly CLPoly
    (fun (l : opaque_list)
      (p : noLeadingZeros l) =>
       depRecCLPoly CLPoly
         (fun (l0 : opaque_list)
           (p0 : noLeadingZeros l0) =>
           depConstrCLPoly (addLists l l0)
             (addListsNoLeadingZeros l
               l0 p p0))
        p2)
    p1.

Definition evalCLPoly (p : CLPoly)
  (n : nat) :=
  depRecCLPoly nat
    (fun (l : opaque_list)
      (proof : noLeadingZeros l) =>
       evalList l n)
    p.
```

```
Definition addCEPPoly (p1 p2 : CEPPoly) :=
  depRecCEPPoly CEPPoly
    (fun (l : opaque_list)
      (p : noLeadingZeros l) =>
       depRecCEPPoly CEPPoly
         (fun (l0 : opaque_list)
           (p0 : noLeadingZeros l0) =>
           depConstrCEPPoly (addLists l l0)
             (addListsNoLeadingZeros l
               l0 p p0))
        p2)
    p1.

Definition evalCEPPoly (p : CEPPoly)
  (n : nat) :=
  depRecCEPPoly nat
    (fun (l : opaque_list)
      (proof : noLeadingZeros l) =>
       evalList l n)
    p.
```

Fig. 19. The definitions of addition and evaluation for polynomials. The original, over `CLPoly`, is on the left, and the repaired version over `CEPPoly` is on the right.

```
Theorem addComm :
  forall (p1 p2 : CLPoly),
    eq_CLPoly (add p1 p2) (add p2 p1).
```

```
Theorem evalRespectsAdd :
  forall (p1 p2 : CLPoly) (n : nat),
    eval (add p1 p2) n =
      (eval p1 n) + (eval p2 n).
```

Fig. 20. The types of `addComm` and `evalRespectsAdd` for `CLPoly`. These theorems were repaired to `CEPPoly`.

port 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.

5.3 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. The former representation uses the simplest possible data type to represent a polynomial, while the latter representation will mimic the way polynomials are commonly written, which will utilize far less storage for polynomials of high degree with many terms having a coefficient of 0. 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 the artifact (11).

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. For example, $x^2 + 3$ is represented as `[1; 0; 3]`. 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. Using the same example, $x^2 + 3$ would be represented as `[(1, 2); (3, 0)]`. Two members of `CEPPoly` are equivalent if the polynomial they represent has the same coefficients. Both representations are found in Figure 17.

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` and `CEPPoly` in Figure 18.

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. The original definitions and the repaired versions can be found in Figure 19. 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 that evaluation distributes over addition. The statements of these theorems can be found in Figure 20. 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 which specifically use setoid rewriting.

Further Steps. The constructor for `CLPoly`, seen in Figure 18, accepts 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`, as well as any other functions and theorems that should not be repaired. This behavior and our strategy for dealing with it are inherited from Pumpkin Pi.

6 Correctness

Up to now, we have been imprecise with what it means for repair to be conducted correctly. The original Pumpkin Pi paper [28] outlines a definition for correctness of repair using univalent type theory. We review that definition here, giving the needed background in univalent type theory. Then, we implement this definition in a proof assistant which has the necessary univalent type theory. This allows us to, for the first time, construct proofs that functions and theorems were correctly repaired.

To state and prove correctness of repair, we need a notion of heterogeneous equality. The Pumpkin Pi paper uses *dependent path equality* in a univalent type system for this purpose. Univalence states that equivalence is equivalent to equality [4, 11, 39]: any equivalence of types corresponds to a

unique proof of equality between those two types. We term these equalities *path equalities*. Given a path equality `p : A ≡ B` between types and given elements `a : A`, `b : B` of those types, we can construct the type of dependent path equalities between those elements, notated `PathP p a b`. This type is inhabited if and only if the equivalence of types corresponding to `p` by univalence maps `a` to `b`.

The repair transformation is parametrized by an equivalence `f : A → B` between the old and new types. By univalence, this equivalence gives an equality between those types. This equality of types in turn extends to equalities of types built using those types: for instance, if `A ≡ B`, then `A → C ≡ B → C`. For any repaired term, then, there is an equality between the type of the original term and the type of the repaired term. We obtain this equality inductively in much the same way we do repair inductively. Whenever an inductive type appears in the old term, if that type is `A`, we use the equality proof corresponding to `f`. Whenever another inductive type appears, we use `refl` as the equality proof. When the other rules would apply, we have theorems constructing an equality proof from the proofs for the constituent terms. Thus, we can construct the type of dependent path equalities between these terms. If that type is inhabited, we say that the term was repaired correctly.

Thus, to use this approach to construct proofs of correct repair, we need to work in a type system that supports this notion of dependent path equality. For this, we choose Cubical Agda. It is true that we could have just as well used Rocq's HoTT library [7], which treats univalence as an axiom. However, Rocq's HoTT library also fundamentally changes how the `Prop` universe works. Switching to such a library for the sake of constructing correctness proofs would have imposed as much overhead for us as directly using a different proof assistant that has univalence, and may have been extra confusing to readers as it would have looked the same as Rocq while relying on fundamentally different dependencies that change the type theory. Therefore, we instead switch to Cubical Agda. We also like the bonus that terms in Cubical Agda compute, since Cubical Agda has univalence as a theorem rather than an axiom.

As a technical detail, we restrict ourselves to working with types which are *h-sets*: that is, types where uniqueness of identity proofs holds. As a bonus, Cubical Agda allows a direct implementation of quotient types via the following higher inductive type:

```
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
```

The first constructor is the constructor for an element of a quotient type as described in Section 2, and the second encodes the equality property for quotient types. The third constructor enforces that quotient types are h-sets. We will use this type instead of setoids when specifying correctness of repair in Cubical Agda. As a result, the additional rules to repair equivalence relations from Figure 5 in Section 3 are not needed when conducting repair in Cubical Agda. The transformation from Figure 4 in Section 3 works, assuming the same annotations as in Rocq, noting that we cannot reuse any of Pumpkin Pi's automation in Cubical Agda. There is only one additional restriction: for a quotient type `Q`, every motive `P : Q → Set` comes with the requirement that `((x : Q) → isSet (P x))`, to ensure that we actually do stay in the h-set fragment of Cubical Agda. To demonstrate that our repair methodology still works under this paradigm, we manually followed the transformation to repair the functions and proofs from the first two case studies in Section 5, which can be found in the artifact (12) (13).

Then, we go about proving theorems which state that each repair rule repairs terms correctly, given that the inputs to the rule are themselves correctly repaired. Some of these theorems are

```
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'
```

Fig. 21. 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 4: `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)
```

Fig. 22. 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.

generic across all types. For example, we internally prove correctness of the Lam rule of the transformation from Figure 4 in Figure 21, which is generic across any repair instance.

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 = ℕ ⊎ 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. The type of the rule showing that the dependent eliminator repairs correctly can be found in Figure 22.

Using these rules, we were able to compose the correctness proofs to show the correctness of repaired *functions*, like addition:

```
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')
```

We also were able to prove theorems about these functions. For instance, below is the type of a term proving that our proof that addition is commutative on $\mathbb{N}$, `addCommNat`, was correctly repaired to a proof that addition is commutative on our new type:

```
addCommCorrect :
  (a : ℕ) (a' : Int / rInt) (pa : PathP (λ i → Nat≡Int/rInt i) a a') →
  (b : ℕ) (b' : Int / rInt) (pb : PathP (λ i → Nat≡Int/rInt i) b b') →
  PathP (λ i → addCorrect a b a' b' pa pb i ≡ addCorrect b a b' a' pb pa i)
    (addCommNat a b) (addCommInt/rInt a' b')
```

When constructing proofs of correct repair for theorems, because of difficulties when composing different `PathP`s, our rules for proving correct repair do not always compose directly in their current formulation. As a result, if Cubical Agda had the facilities for building automation, we would not currently have a complete procedure for constructing proofs of correct repair fully automatically. However, we are able to prove these theorems directly in such cases, demonstrating that our repair procedure produces the desired output. All of the rules we prove, as well as the proofs of correct repair in our example, can be found in the artifact (14).

7 Related Work

Proof Repair. This work extends the Pumpkin Pi [28] proof repair transformation and Rocq 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. [30] and Robert [31], with strong influence from program repair [25]. Sisyphus [19] 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 Rocq using an embedded separation logic, whereas our work repairs proofs that are written in Rocq 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 [40]. Cubical type theory [4, 11, 15] 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 [39], which presents the univalence axiom non-constructively. Homotopy type theory has additionally been implemented in Rocq as the HoTT library [7]. Univalence leads to a related idea, the Structure Identity Principle, which states that the type of equalities of structures is equivalent to the type of isomorphisms of those structures [1, 14, 39].

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 [13] 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 [22] 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 [10, 18, 43]. More work on proof reuse can be found in the QED at Large [27] survey of proof engineering.

Work has been done to implement transfer tools in Rocq that approximate or externally implement automation corresponding to univalent transport. Tabareau et al. [34] 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 [35] similar to the repair transformation from Pumpkin Pi, which Ringer [26] 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 [12] implements external transfer for Rocq 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 implement repair.

Quotients and Equivalences. Our work uses quotient types [21] to expand the scope of proof repair. Maietti [24] shows that, in intensional Martin-Löf type theory with uniqueness of identity proofs and at least two universes, the existence of effective quotients implies the law of the excluded middle for types in the first universe. Quotient types exist in other proofs assistants besides Cubical Agda, like Isabelle/HOL [23, 42], as well as Lean [5] by way of axioms. Bortin and Lüth [9] use quotient types to construct theories in Isabelle, like multisets and finite sets as quotients of lists. Rocq does not have quotient types, but it does have setoids [32], 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 [2]. XTT uses Bishop sets, analogous to setoids, to construct a type theory where all types have definitional uniqueness of identity proofs [33]. We draw on quotient types for our work in Cubical Agda, and we draw on setoids for our work in Rocq.

Our idea for extending proof repair using quotient type equivalences to begin with comes from Angiuli et al. [3], 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 [28].

8 Conclusions & Future Work

We extended Pumpkin Pi to support changes represented by quotient type equivalences, enabling repair in situations previously untenable. The key challenge we overcame was supporting quotient types in a proof repair algorithm built for a type theory that does not 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. We also constructed the first internal correctness proofs for repair. We wish 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 also hope to tackle automation for proof repair directly in Cubical Agda, which is challenging because Cubical Agda lacks tools for building automation (except by reflection). We hope this will open the door to proof repair for more sophisticated classes of changes.

Data-Availabilty Statement

This paper comes with an artifact hosted on Zenodo containing the extension to Pumpkin Pi we wrote, as well as all the case studies described within the paper [41]. Instructions for reproducing the results of the artifact can be found in `Overview.md` in the artifact. The source code for our extension to Pumpkin Pi and our case studies can be found on Github. Links to specific files in the Github source have been provided where relevant throughout the paper. These links are given as circled numbers, like ①.

Acknowledgements

We thank Amélia Liao, Reed Mullanix, Tom Jack, and the Cubical Agda Discord server for their help in using Cubical Agda. We thank also Carlo Angiuli for their early thoughts on this project. This material is based upon work supported by the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), and the Naval Information Warfare Center Pacific (NIWC Pacific) under Contracts No. N66001-21-C-4023 and FA8750-24-C-B044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL, DARPA, and NIWC Pacific.

References

[1] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. 2022. The Univalence Principle. arXiv:2102.06275 [math.CT] https://arxiv.org/abs/2102.06275

[2] 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. https://doi.org/10.1007/978-3-030-33636-3_7

[3] 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

[4] 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

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

[6] Gilles Barthe, Venanzio Capretta, and Olivier Pons. 2003. Setoids in type theory. *J. Funct. Program.* 13, 2 (March 2003), 261–293. https://doi.org/10.1017/S0956796802004501

[7] 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

[8] Errett Bishop. 1967. *Foundations of Constructive Analysis.* Mcgraw-Hill, New York, NY, USA.

[9] 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. https://doi.org/10.1007/978-3-642-14128-7_5

[10] 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

[11] 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

[12] 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. https://doi.org/10.1007/978-3-031-57262-3_10

[13] 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. https://doi.org/10.1007/978-3-319-03545-1_10

[14] Thierry Coquand and Nils Anders Danielsson. 2013. Isomorphism is equality. *Indagationes Mathematicae* 24, 4 (2013), 1105–1120. https://doi.org/10.1016/j.indag.2013.09.002 In memory of N.G. (Dick) de Bruijn (1918–2012).

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] Martin Hofmann. 1994. Elimination of extensionality in Martin-Löf type theory. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–190. https://doi.org/10.1007/3-540-58085-9_76

[21] Martin Hofmann. 1995. Extensional concepts in intensional type theory. (1995).

[22] 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. https://doi.org/10.1007/978-3-319-03545-1_9

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

[24] Maria Emilia Maietti. 1999. About Effective Quotients in Constructive Type Theory. In *Types for Proofs and Programs*, Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–178. https://doi.org/10.1007/3-540-48167-2_12

[25] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

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

[27] 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

[28] 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. https://doi.org/10.1145/3453483.3454033

[29] 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

[30] 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

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

[32] Matthieu Sozeau. 1999-2023. Rocq Reference Manual, Generalized Rewriting. https://rocq-prover.org/doc/V9.0.0/refman/addendum/generalized-rewriting.html

[33] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2022. A Cubical Language for Bishop Sets. *Logical Methods in Computer Science* Volume 18, Issue 1, Article 43 (Mar 2022). https://doi.org/10.46298/lmcs-18(1:43)2022

[34] 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

[35] 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

[36] The Rocq Development Team. 1999-2023. Rocq Reference Manual, Reasoning with equalities. https://rocq-prover.org/doc/V8.13.2/refman/proofs/writing-proofs/rewriting.html

[37] The Rocq Development Team. 2019. *The Rocq Prover Standard Library, Coq.Classes.Morphisms.* https://rocq-prover.org/doc/v8.9/stdlib/Coq.Classes.Morphisms.html

[38] 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. https://doi.org/10.1007/978-3-319-25150-9_36

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

[40] 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

[41] Cosmo Viola, Max Fan, and Talia Ringer. 2025. *Proof Repair across Quotient Type Equivalences - Artifact.* https://doi.org/10.5281/zenodo.16921959

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

[43] 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

Received 2025-03-26; accepted 2025-08-12