Proof Repair and (Co)algebra Equivalences

E.G. PLANTE III, University of Illinois at Urbana-Champaign, USA
ERIC PAUL, University of Illinois at Urbana-Champaign, USA
COSMO VIOLA, University of Illinois at Urbana-Champaign, USA
G.A. KAVVOS, University of Bristol, United Kingdom
TALIA RINGER, University of Illinois at Urbana-Champaign, USA

Maintaining formal proofs in response to breaking changes in their dependencies is a major challenge of proof engineering. *Proof repair* is a kind of proof automation that lessens the burden of this challenge by automatically adapting proofs in response to those changes. While proof repair is well-understood from an algorithmic lens, it has, until now, been poorly understood from a theoretical lens. We address this gap by giving the first foundational treatment of proof repair, focusing in particular on type-theoretic proof repair in response to changes in datatypes. Our key insight is that the core component of this proof repair algorithm—a "configuration"—is exactly an algebra isomorphism. We formalize and prove this insight, answering open questions from prior work. We then take this one step further and show that, by taking the dual of this representation, we obtain proof repair over coinductive types. All of our results are formalized in Agda.

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

Additional Key Words and Phrases: Proof Repair, Type Theory, Category Theory

1 Introduction

Formal proof developments in proof assistants like Agda, Rocq, and Lean [3, 34, 36] have grown larger as the technology has grown more accessible. One timeless challenge in large formal proof developments is the burden of change [43]. Proof repair has emerged as a promising tool to cope with change by automatically fixing formal proofs in response to breaking changes.

This paper is about a class of tools for coping with changes in datatypes in proof developments that we refer to as *type-theoretic proof repair* [42, 44, 54]: given an old and a new version of a datatype, these tools automatically rewrite the programs and proofs stated over the old version into programs and proofs stated over the new one. Type-theoretic proof repair leverages the underlying type theory of a proof assistant to establish relationships between datatypes across which programs and proofs can be repaired. This is, of course, not the only kind of proof repair—other repair tools exist that, for example, repair proofs concerning higher-order imperative code [23], or leverage non-type-theoretic methods like large language models [31] (see Section 5 for more).

The objective of this paper is to provide a mathematical foundation for type-theoretic proof repair tools that clarifies the scope of changes contemporary repair tools can handle and the extensions proof repair tools can support. To this end, we formalize the prior work on proof repair within type theory, establish its connection to the categorical notion of algebras on endofunctors, and finally dualize our connection to develop proof repair on coinductive types. Our treatment characterizes the objects at the heart of repair tools rather than the syntactic transformations the tools perform over terms, which is beyond our present scope.

The principal theme undergirding our work is that **proof repair is an algebra isomorphism**. Our intention is to bring this conclusion to light by carefully relating prior proof repair work to the type-theoretic understanding of datatypes. This perspective explains why existing repair tools work, settles questions left open in prior work, and relaxes assumptions repair was thought to

Authors' Contact Information: E.G. Plante III, University of Illinois at Urbana-Champaign, USA; Eric Paul, University of Illinois at Urbana-Champaign, USA; Cosmo Viola, University of Illinois at Urbana-Champaign, USA; G.A. Kavvos, University of Bristol, United Kingdom; Talia Ringer, University of Illinois at Urbana-Champaign, USA.

require. It also points beyond current tools, to repair for types no existing tool can handle. All of our results and examples are marked with a blue checkmark linking to the corresponding Agda formalization.

Contributions. Concretely, our contributions are:

- We establish results about the fundamental constructs of proof repair tools: configurations and equivalences (Section 2). Our development formalizes results stated informally in previous work and develops new results that give a broader understanding of repair.
- We demonstrate that proof repair can be understood through the notion of initial algebras on a suitable functor (Section 3). This new viewpoint clarifies the class of relationships between types that configuration-based repair captures, characterizing them exactly in terms of universal properties. We recast a case study from the original work on proof repair in terms of algebras, and formalize the connection to proof repair algorithms. Our main result asserts that configurations are algebra isomorphisms.
- We find that repair is suitable for type theories that are inconsistent with univalence, which was previously thought to be a requirement. This opens the door for new implementations of repair in proof assistants like Lean (Section 3.4).
- With our new categorical perspective on proof repair, we develop the foundations for repair algorithms that handle a new class of types: coinductive types (Section 4).

Motivation

We introduce proof repair using a classic case study ✓. Suppose that a proof engineer may have previously developed a verified library using the `Nat` datatype. They want to change this library to use `Bin`, which uses a more efficient, binary representation:

```
data Nat : Type where
    zero : Nat
    succ : Nat -> Nat
```

```
data Pos : Type where
 one : Pos
 pb0 : Pos -> Pos
 pb1 : Pos -> Pos

data Bin : Type where
 bzero : Bin
 bpos  : Pos -> Bin
```

Proof repair is the study of algorithms that replace one of these representations with the other. To understand what such an algorithm must accomplish, consider the following definitions:

```
isZeroN : Nat -> Bool
isZeroN zero   = true
isZeroN (succ _) = false
```

```
zUnit : (n : Nat) -> n + zero = n
zUnit zero   = refl
zUnit (succ m) = ap succ (zUnit m)
```

While making a datatype change in a verified library, a proof engineer may encounter many such programs and proofs, requiring broad changes to their development. A simple way of achieving that is to compose each old term with a conversion function `toNat : Bin -> Nat`:

```
isZeroB : Bin -> Bool
isZeroB b = isZeroN (toNat b)
```

```
zUnitB : (b : Bin) -> toNat b + zero = toNat b
zUnitB b = zUnit (toNat b)
```

However, this output is unsatisfying because the transferred terms still route every operation back through `Nat` via `toNat`, so the old representation remains a dependency of the library.

Repair algorithms aim to do better than that. When given a *configuration*, i.e. a decomposition of a function that is an equivalence of types into a mapping between the constructors and eliminators of `Nat` and `Bin`, the repair algorithm specializes the old code to the new datatype, producing native definitions that recurse directly on `Bin`'s constructors, without mentioning `Nat`:

```
isZeroB : Bin -> Bool
isZeroB bzero  = true
isZeroB (bpos _) = false
```

```
zUnitB : (b : Bin) -> b + bzero = b
zUnitB bzero  = refl
zUnitB (bpos p) = ap bpos (pUnit p)
```

This algorithmic view of proof repair is well understood [44]. Before developing the algebra perspective, we give a formal account of proof repair, both to connect it to our abstract development and to establish results about the mechanics of repair.

Type Theory, Notation, and Conventions

We work within an intensional Martin-Löf type theory with dependent products, dependent sums, and intensional identity types, using Agda as a proxy for this type theory. Our development uses the homotopy-theoretic reading of the identity type [52], but assumes no univalence principles. The only axioms we assume on the inductive side are function extensionality together with its computation rule on happly and the corresponding uniqueness rule (every path of functions is the extensionality of its happly). For the coinductive development (Section 4) we work in Cubical Agda [53], whose path type supports the copattern-style reasoning that coinductive types demand; there, function extensionality is a theorem and, again, no use of univalence (or other cubical-specific axioms beyond the path type itself) is made.

We write $=$ for the identity (path) type and $=_{\mathrm{def}}$ for definitions. We write $(a : A) \times B\,a$ for dependent sums and freely use $\times$ for its non-dependent instance. We write $* : \mathbb{1}$ for the unit type and its inhabitant. A *type equivalence* $f : A \simeq B$ is a bi-invertible map: a function $f : A \to B$ together with a left inverse and a right inverse. A type is *contractible* when it has a term to which every term is equal; it is a *proposition* when any two of its terms are equal, and a *set* when its identity types are all propositions. Finally, when one equation depends on another we display it as a plain equality rather than writing the transport or dependent path explicitly; the formal Agda statements carry the dependency in full.

2 Proof Repair via Configurations

In this section, we formally define configurations and establish the correspondence between configurations and type equivalences.

The prior work on tools for type-theoretic repair, such as PUMPKIN Pi [44, 54], develop the notion of a *configuration* to capture a decomposed version of a type-theoretic equivalence $f : A \simeq B$ at the level of the metalanguage. The repair tool then operates on the syntax of the type theory, replacing terms of type A with terms of type B according to this specification. A configuration for the example in Section 1 names each type's constructors by `DepConstr` and its dependent eliminator by `DepElim`, whose computation rules are `Iota`.[1]

```
-- DepConstr: constructors, indexed by type and constructor number
DepConstr Nat 0 = zero   DepConstr Bin 0 = bzero
DepConstr Nat 1 = succ   DepConstr Bin 1 = binSucc

-- DepElim: a dependent eliminator for each type
DepElim Nat : (P : Nat -> Type) -> P (DepConstr Nat 0) -> ((n : Nat) -> P n
            -> P (DepConstr Nat 1 n)) -> (n : Nat) -> P n
DepElim Bin : (P : Bin -> Type) -> P (DepConstr Bin 0) -> ((b : Bin) -> P b
            -> P (DepConstr Bin 1 b)) -> (b : Bin) -> P b

-- Iota: the eliminator computes on each constructor
Iota Nat 0 : DepElim Nat P pz ps (DepConstr Nat 0) = pz
Iota Nat 1 : DepElim Nat P pz ps (DepConstr Nat 1 n) = ps n (DepElim Nat P pz ps n)
Iota Bin 0 : DepElim Bin P pz ps (DepConstr Bin 0) = pz
Iota Bin 1 : DepElim Bin P pz ps (DepConstr Bin 1 b) = ps b (DepElim Bin P pz ps b)
```

[1] We omit the `Eta` data that prior work included because our formalization shows it is redundant.

Proof repair tools then use the decomposition of the source and target types' inductive structure to automatically repair code, by component-wise replacing the original constructors and eliminators with the new ones. The repair algorithm replaces every `DepElim Nat` by `DepElim Bin` and `DepConstr Nat` by `DepConstr Bin`, while the case data handed to the eliminator is carried over unchanged. For example, the proof `zUnit` given below would be repaired into `zUnitB`:

```
zUnit : (n : Nat) -> n + zero = n
zUnit = DepElim Nat (λn -> n + zero = n)
  refl
  (λn ih -> ap succ ih)
```

```
zUnitB : (b : Bin) -> b + bzero = b
zUnitB = DepElim Bin (λb -> b + bzero = b)
  refl
  (λb ih -> ap binSucc ih)
```

The basis of a configuration lies in the constructors of a type, which may take recursive and non-recursive arguments. We call a specification of these constructors a *constructor algebra*. Our definitions here are inspired by the work of Dybjer [20] and Chapman et al. [14], both of which proved crucial in formalizing the previously informal notion of configurations.

Definition 2.1 (Constructor Algebra ✓)**.** To describe the argument list of a single constructor as a telescope, we define a *constructor arity* type generated by:

$$\begin{aligned}\mathsf{Done} &: \mathsf{ConstrArity}\\ \mathsf{Nonrec} &: (A : \mathsf{Type}) \to (A \to \mathsf{ConstrArity}) \to \mathsf{ConstrArity}\\ \mathsf{Rec} &: (D : \mathsf{Type}) \to \mathsf{ConstrArity} \to \mathsf{ConstrArity}.\end{aligned}$$

Here Done closes the telescope, Nonrec $A\,k$ records a non-recursive argument of type A on which the remaining arity k may depend, and Rec $D\,cs$ records a recursive argument with positions indexed by D—that is, a D-indexed family of recursive subterms, interpreted as a map $D \to X$ into the carrier. Non-recursive and recursive arguments may be freely interleaved.

We define the type of *arguments* to a constructor, parametrized by a carrier X standing for the inductive type itself:

$$\begin{aligned}&\mathsf{Args} : \mathsf{ConstrArity} \to \mathsf{Type} \to \mathsf{Type}\\ &\mathsf{Args}\ \mathsf{Done}\ X =_{\mathrm{def}} \mathbb{1}\\ &\mathsf{Args}\ (\mathsf{Nonrec}\ A\,k)\ X =_{\mathrm{def}} (a : A) \times \mathsf{Args}\ (k\ a)\ X\\ &\mathsf{Args}\ (\mathsf{Rec}\ D\ cs)\ X =_{\mathrm{def}} (D \to X) \times \mathsf{Args}\ cs\ X.\end{aligned}$$

A *signature* is a type of constructor labels together with an assignment of an arity to each label:

$$\mathsf{Signature} =_{\mathrm{def}} (\mathsf{Op} : \mathsf{Type}) \times (\mathsf{Op} \to \mathsf{ConstrArity})$$

Finally, a *constructor algebra* for a signature on a carrier X interprets every label as a constructor taking its arguments into the carrier:

$$\begin{aligned}&\mathsf{ConstrAlgebra} : \mathsf{Signature} \to \mathsf{Type} \to \mathsf{Type}\\ &\mathsf{ConstrAlgebra}\ (\mathsf{Op},\ \mathsf{arity})\ X =_{\mathrm{def}} \big((c : \mathsf{Op}) \times \mathsf{Args}\ (\mathsf{arity}\ c)\ X\big) \to X.\end{aligned}$$

For a map $g : X \to Y$ and $cs : \mathsf{ConstrArity}$, define $\mathsf{mapArgs}\ cs\ g : \mathsf{Args}\ cs\ X \to \mathsf{Args}\ cs\ Y$ by postcomposing every recursive position with g and leaving non-recursive data untouched:

$$\begin{aligned}&\mathsf{mapArgs}\ \mathsf{Done}\ g\ as =_{\mathrm{def}} as\\ &\mathsf{mapArgs}\ (\mathsf{Nonrec}\ A\,k)\ g\ (a, as) =_{\mathrm{def}} (a,\ \mathsf{mapArgs}\ (k\ a)\ g\ as)\\ &\mathsf{mapArgs}\ (\mathsf{Rec}\ D\ cs)\ g\ (r, as) =_{\mathrm{def}} (g \circ r,\ \mathsf{mapArgs}\ cs\ g\ as).\end{aligned}$$

This action is functorial: it preserves identities and composition, and it carries an equivalence $X \simeq Y$ on carriers to an equivalence $\mathsf{Args}\ cs\ X \simeq \mathsf{Args}\ cs\ Y$ on arguments.

A configuration captures the shared inductive structure of two types by taking a constructor algebra and a proof that the type has an inductive eliminator for that constructor algebra, as well as a β law for that induction principle.

Definition 2.2 (Configuration ✓)**.** Fix a signature $sig = (\mathsf{Op}, \mathsf{arity})$ and a constructor algebra $\alpha : \mathsf{ConstrAlgebra}\ sig\ C$. Relative to a *motive* $P : C \to \mathsf{Type}$, the *inductive hypotheses* at an argument list provide a proof of P at every recursive child. Formally,

$$\mathsf{IH}\ \mathsf{Done}\ P\ as =_{\mathrm{def}} \mathbb{1}$$

$$\mathsf{IH}\ (\mathsf{Nonrec}\ A\ k)\ P\ (a, as) =_{\mathrm{def}} \mathsf{IH}\ (k\ a)\ P\ as$$

$$\mathsf{IH}\ (\mathsf{Rec}\ D\ cs)\ P\ (r, as) =_{\mathrm{def}} \big((d : D) \to P\ (r\ d)\big) \times \mathsf{IH}\ cs\ P\ as.$$

An *induction principle* for α takes a motive P together with, for each label c, a case sending arguments as and their inductive hypotheses to a value of $P\ (\alpha\ c\ as)$, and produces a dependent section:

$$\begin{aligned}\mathsf{Ind}\ sig\ \alpha =_{\mathrm{def}} (P : C \to \mathsf{Type}) \to\ & \Big((c : \mathsf{Op})\ (as : \mathsf{Args}\ (\mathsf{arity}\ c)\ C) \to \mathsf{IH}\ (\mathsf{arity}\ c)\ P\ as \to P\ (\alpha\ c\ as)\Big) \to \\ & (x : C) \to P\ x.\end{aligned}$$

Such an eliminator ind satisfies the *β law* when it computes on constructors: writing $\overline{ih}$ for the canonical inductive hypotheses obtained by applying the eliminator itself to each recursive child, for all P, cases, label c, and arguments as,

$$\mathsf{ind}\ P\ cases\ (\alpha\ c\ as) \;=\; cases\ c\ as\ \overline{ih}.$$

An *inductive algebra* for sig on C packages a constructor algebra with an induction principle and its β law (referred to as ι in prior proof repair work):

$$\mathsf{IndAlg}\ sig\ C =_{\mathrm{def}} (\alpha : \mathsf{ConstrAlgebra}\ sig\ C) \times (\mathsf{ind} : \mathsf{Ind}\ sig\ \alpha) \times \mathsf{Beta}\ sig\ \mathsf{ind}.$$

Finally, a *configuration* for sig between carriers C and D is a pair of inductive algebras for sig, one on each carrier:

$$\mathsf{Config}\ sig\ C\ D =_{\mathrm{def}} \mathsf{IndAlg}\ sig\ C \times \mathsf{IndAlg}\ sig\ D.$$

This definition internalizes the `DepConstr/DepElim/Iota` data that PUMPKIN Pi externally maintains; the results that follow are theorems about this reconstruction, and we support its faithfulness by recovering the tool's correctness criteria (Proposition 2.4) and case studies from it.

Though our signatures are non-indexed, our formalization handles the repair of inductive families in a way that is consistent with the proof repair literature. Our formalization includes the case study from Ringer et al. [44] of repairing between $\mathtt{Vec}\ A\ n$ and $\mathtt{Fin}\ n \to A$ where the per-index signature $sig\ n$ has a single, non-recursive operation installing the n values ✓. Programs and proofs over vectors repair index-by-index, with the coherence conditions of Proposition 2.4 instantiated at each n. So the restriction to non-indexed signatures reflects the class of changes contemporary repair tools perform, which doesn't include repairing types with differing index types.

Configurations were developed as an algorithm-friendly decomposition of type equivalences, which in a tool setting are externalized in some automation language. Equipped with our formal definitions, we are able to verify that configurations do indeed correspond to type equivalences.

Proposition 2.3 (Configurations and Inductive Equivalences are Equivalent ✓)**.** *Fix a signature sig and C, D : Type. In the presence of an inductive algebra on C, the type equivalences and configurations are equivalent:*

$$\mathsf{IndAlg}\ sig\ C \;\times\; (C \simeq D) \;\simeq\; \mathsf{Config}\ sig\ C\ D.$$

Proof sketch. An inductive algebra on C appears on both sides, so fixing one it suffices to give an equivalence $(C \simeq D) \simeq \mathsf{IndAlg}\ sig\ D$. The inductive algebra transports directly across the equivalence. Conversely, a pair of inductive algebras determines an equivalence through their folds. Each inductive algebra sends a constructor algebra γ on a type Y to a map $\mathsf{fold}_C\ \gamma : C \to Y$, the eliminator at the constant motive Y, characterized by its β rule

$$\mathsf{fold}_C\ \gamma\ (\alpha_C\ c\ as) \;=\; \gamma\ c\ \big(\mathsf{mapArgs}\ (\mathsf{arity}\ c)\ (\mathsf{fold}_C\ \gamma)\ as\big).$$

Writing $\varphi = \mathsf{fold}_C\, \alpha_D$ and $\psi = \mathsf{fold}_D\, \alpha_C$, the composite $\psi \circ \varphi$ is a fold of α_C into itself; so is the identity, and the fold into a fixed algebra is unique, so $\psi \circ \varphi = \mathsf{id}_C$, and symmetrically $\varphi \circ \psi = \mathsf{id}_D$. These two constructions are mutually inverse, giving the equivalence. □

The stipulation that an inductive algebra on C must be present reveals that equivalences do not capture all of the information contained in a configuration. We offer a better perspective on this issue in Section 3 via P-algebras.

In the typical case of repair, a proof engineer has a natural choice of configuration in mind, but, in general, unnatural and useless configurations exist. For example, a configuration could reuse the same type or trivially specify a single constructor and eliminator. It is up to the proof engineer and repair tooling to ensure that configurations execute a useful form of repair. We return to this point in Section 3, where the algebraic reformulation makes this precise.

A configuration can also satisfy correctness criteria [44] that ensure that the configurations behave coherently with the equivalence the configuration is intended to capture. Our formal definition satisfies these criteria.

Proposition 2.4 (Correctness Conditions ✓)**.** *Fix* $(\mathcal{A}_C, \mathcal{A}_D) : \mathsf{Config}\ sig\ C\ D$ *with constructor algebras* α_C, α_D*, eliminators* $\mathsf{ind}_C, \mathsf{ind}_D$*, and* β *laws* β_C, β_D*. Let* $\varphi : C \to D$ *be the forward map of the equivalence induced by the configuration (Proposition 2.3). The configuration satisfies the three coherence conditions of PUMPKIN Pi [44].*

(1) constr_ok*. Constructors commute with* φ*: for every label* c *and arguments* $as : \mathsf{Args}\ (\mathsf{arity}\ c)\ C$*,*

$$\varphi\ (\alpha_C\ c\ as)\ =\ \alpha_D\ c\ \big(\mathsf{mapArgs}\ (\mathsf{arity}\ c)\ \varphi\ as\big).$$

(The formal statement is a heterogeneous strengthening, relating C-side and D-side arguments across the lifted argument equivalence; the form displayed here is the instance at φ*-mapped arguments.)*

(2) iota_ok*. Each eliminator computes on its own constructors: this is exactly the* β *law,* β_C *for* ind_C *and* β_D *for* ind_D*.*

(3) elim_ok*. The eliminators commute with* φ*: for every motive* $P : D \to \mathcal{U}$ *and cases* cs_D*, there are transported C-side cases* $\varphi^* cs_D$ *(delegating to* cs_D *on the* φ*-mapped arguments and transporting back along* constr_ok*) such that, for every* $x : C$*,*

$$\mathsf{ind}_C\ (P \circ \varphi)\ (\varphi^* cs_D)\ x\ =\ \mathsf{ind}_D\ P\ cs_D\ (\varphi\ x).$$

Proof sketch. All three are straightforwardly derivable from the structure carried by the configuration and function extensionality. □

We apply the equivalence explicitly rather than using univalence and transport as in Ringer et al. [44]. The first condition, constr_ok, states that φ *respects the constructors*, sending a constructed value to the corresponding construction on the other side. The remaining two ask nothing of φ in isolation; iota_ok is just each type computing on its own constructors, and elim_ok is recovered from constr_ok once each type is known to be generated by its constructors in the strong sense the eliminators capture. Thus, a single condition does the work, and the rest follows from source and target being canonically built from their constructors. We will see in Section 3 that this is no accident: a structure-respecting map between two such canonically-generated types must be an isomorphism, and naming this phenomenon precisely is what the algebraic reformulation buys. This is the first hint of our central thesis, that proof repair is an algebra isomorphism.

Additionally, our formalization enables us to prove a conjecture about configurations stated in Viola et al. [54]. The original proof repair work contained extra data for an η principle in both the configuration and correctness conditions that we confirm is not needed because it is derivable from the existing data ✓.

Configurations are a useful framework for building proof repair tools that operate on the syntax of a dependent type theory. However, the syntactic detail that makes configurations useful to tools becomes a burden for foundations. Two questions remain that we give answers (Section 3) to in terms of well-understood type theory:

(1) Any two countably infinite types are equivalent, but theorems and programs do not meaningfully translate across arbitrary bijections. Can we isolate when repair is "natural"?
(2) What are the changes that configurations can handle?

3 Algebraic Foundations for Proof Repair

To answer these questions, we formulate repair in terms of the language of algebras on polynomial endofunctors [7, 11], which we shortly introduce from scratch for the unfamiliar reader. We first introduce P-algebras and the algebraic view of inductive datatypes, providing the necessary background. We then establish that configurations are a representation of algebra isomorphisms. Finally, we showcase a purely algebraic form of proof repair.

3.1 Warming up to Algebras

We begin with a tour of P-algebras. Note that we are working within type theory, and so the definitions we use have a type-theoretic flavor and are not the precise definitions one may see in a category theory textbook. We nonetheless refer to these type-theoretic notions by their usual category-theoretic names.

Definition 3.1 (Polynomial Endofunctor ✓)**.** Fix a type A of *shapes* and a family $B : A \to \mathsf{Type}$ assigning to each shape its type of *positions*. The associated polynomial endofunctor is defined as

$$\mathrm{P} : \mathsf{Type} \to \mathsf{Type}$$
$$\mathrm{P}\,X \;=_{\mathrm{def}}\; (a : A) \times (B\,a \to X),$$

a shape a together with a $B\,a$-indexed family of children drawn from X. For a fixed A and B, P's action on a function $f : X \to Y$ is post-composition on the children, which makes P a functor.

$$\text{P-map} : (X \to Y) \to \mathrm{P}\,X \to \mathrm{P}\,Y$$
$$\text{P-map}\,f\,(a, u) \;=_{\mathrm{def}}\; (a, f \circ u)$$

We call a pair $(A : \mathsf{Type},\ B : A \to \mathsf{Type})$ a *polynomial*.

Definition 3.2 (P-algebra ✓)**.** For a polynomial endofunctor P, a *P-algebra* is a pair (C, σ) of a carrier C and a structure map $\sigma : P\,C \to C$. Fix $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$. We define a P-algebra formally as follows

$$\text{P-Alg} : \mathsf{Type}$$
$$\text{P-Alg} =_{\mathrm{def}} (C : \mathsf{Type}) \times (\mathrm{P}\,C \to C)$$

Example 3.3 (Nat Algebra ✓)**.** Consider the functor

$$P : \mathsf{Type} \to \mathsf{Type} \qquad P\,X =_{\mathrm{def}} \mathbb{1} + X.$$

A P-algebra is a pair (C, σ) with a structure map $\sigma : \mathbb{1} + C \to C$. By the universal property of the coproduct, such a map is equivalently a point $z : C$ together with an endomap $s : C \to C$, which we bundle as $\sigma = [\,z\,,\,s\,]$. Thus, a P-algebra is exactly a type equipped with an interpretation of a `zero` constructor and a `succ` constructor. The type `Nat` carries the algebra $(\texttt{Nat}, [\,\texttt{zero}, \texttt{succ}\,])$, reading the nullary shape as `zero` and the unary shape as `succ`. The payoff of the algebraic view is that the same functor has many carriers: `Bin` carries $(\texttt{Bin}, [\,\texttt{bzero}, \texttt{binSucc}\,])$, a different interpretation of the same two shapes.

Definition 3.4 (Algebra Homomorphism ✓)**.** Given two algebras (C, σ) and (D, τ) for a polynomial endofunctor $P : \mathsf{Type} \to \mathsf{Type}$, an algebra homomorphism from (C, σ) to (D, τ) is a function $h : C \to D$ such that the following diagram commutes:

$$\begin{array}{ccc} PC & \xrightarrow{Ph} & PD \\ \downarrow{\scriptstyle \sigma} & & \downarrow{\scriptstyle \tau} \\ C & \xrightarrow{h} & D \end{array}$$

Formally, we define the type of algebra homomorphisms as the function space between the carriers paired with the proof of the above commutative diagram:

$$\begin{aligned} &\mathsf{isAlgHom} : (X\ Y\ : \text{P-Alg}\ A\ B) \to (\mathsf{Carrier}\ X \to \mathsf{Carrier}\ Y) \to \mathsf{Type} \\ &\mathsf{isAlgHom}\ (C,\ \sigma)\ (D,\ \tau)\ h =_{\text{def}} h \circ \sigma = \tau \circ \text{P-map}\ h \\ \\ &\mathsf{AlgHom} : \text{P-Alg}\ A\ B \to \text{P-Alg}\ A\ B \to \mathsf{Type} \\ &\mathsf{AlgHom}\ X\ Y =_{\text{def}} (h\ :\ \mathsf{Carrier}\ X \to \mathsf{Carrier}\ Y) \times (\mathsf{isAlgHom}\ X\ Y\ h) \end{aligned}$$

Algebra homomorphisms compose (AlgHom−∘), and every algebra carries the identity homomorphism id-hom, whose carrier map is the identity function.

Example 3.5 (Nat Homomorphism ✓)**.** Homomorphisms between P-algebras are maps that preserve the constructors of the source algebra. The map `fromNat : Nat → Bin` is a homomorphism:

$$\begin{array}{ccc} \mathbb{1} + \mathtt{Nat} & \xrightarrow{\mathbb{1} + \mathtt{fromNat}} & \mathbb{1} + \mathtt{Bin} \\ {\scriptstyle [\,\mathtt{zero},\,\mathtt{succ}\,]}\downarrow & & \downarrow{\scriptstyle [\,\mathtt{bzero},\,\mathtt{binSucc}\,]} \\ \mathtt{Nat} & \xrightarrow[\mathtt{fromNat}]{} & \mathtt{Bin} \end{array}$$

This commutation is precisely the constr_ok condition of Proposition 2.4: a homomorphism is a map that respects the constructors.

Since the central claim of this paper is that proof repair is an algebra isomorphism, we next fix precisely what an isomorphism of algebras is.

Definition 3.6 (Algebra Equivalence ✓)**.** Fix algebras $X, Y : \text{P-Alg}\ A\ B$. An algebra homomorphism $f : \mathsf{AlgHom}\ X\ Y$ is an *algebra equivalence* when it has a right inverse and a left inverse:

$$\begin{aligned} &\mathsf{isAlgEquiv} : (X\ Y : \text{P-Alg}\ A\ B) \to \mathsf{AlgHom}\ X\ Y \to \mathsf{Type} \\ &\mathsf{isAlgEquiv}\ X\ Y\ f =_{\text{def}} \big((g : \mathsf{AlgHom}\ Y\ X) \times (\text{AlgHom-}\circ\ f\ g = \text{id-hom}_X)\big) \\ &\qquad\qquad \times \big((h : \mathsf{AlgHom}\ Y\ X) \times (\text{AlgHom-}\circ\ h\ f = \text{id-hom}_Y)\big) \\ \\ &\mathsf{AlgEquiv} : \text{P-Alg}\ A\ B \to \text{P-Alg}\ A\ B \to \mathsf{Type} \\ &\mathsf{AlgEquiv}\ X\ Y =_{\text{def}} (f : \mathsf{AlgHom}\ X\ Y) \times \mathsf{isAlgEquiv}\ X\ Y\ f \end{aligned}$$

Forgetting the homomorphism witnesses, the carrier maps of an algebra equivalence form a type equivalence $\mathsf{Carrier}\ X \simeq \mathsf{Carrier}\ Y$; an algebra equivalence is exactly a type equivalence whose two maps respect the algebra structure. We say *algebra isomorphism* interchangeably.

Definition 3.7 (Homotopy-initial Algebra ✓)**.** A P-algebra X is *homotopy-initial* if for every P-algebra Y the type of homomorphisms $X \to Y$ is contractible (isContr). Formally, for some $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$,

$$\begin{aligned} &\mathsf{isInitAlg} : \text{P-Alg}\ A\ B \to \mathsf{Type} \\ &\mathsf{isInitAlg}\ X =_{\text{def}} (Y : \text{P-Alg}\ A\ B) \to \mathsf{isContr}(\mathsf{AlgHom}\ X\ Y) \end{aligned}$$

For brevity, we may refer to homotopy-initial algebras as initial algebras.

Equipped with our P-algebra toolbox, we turn back to our motivating example and phrase it in terms of algebras. In this example we are implicitly relying on the correspondence between homotopy-initial algebras and W-types established by Awodey, Gambino, and Sojakova [12]. Of course, we are utilizing Agda's `data` declaration instead of W-types, but we are still able to put an algebra structure on the Agda datatypes as they capture the same inductive structure.

Example 3.8 (Motivating Example with Initial Algebras ✓)**.** First, we obtain that `Nat` and `Bin` are initial algebras for the functor $PX = \mathbb{1} + X$.

```
NatInit : isInitAlg (Nat , [ zero , succ ])
BinInit : isInitAlg (Bin , [ bzero , binSucc ])
```

The unique homomorphism out of the initial algebra is the recursor, and its dependent strengthening is the induction principle. Both are derived from `isInitAlg` alone — the recursor as the carrier of the unique map, the eliminator from the section it determines on the total space.

```
recNat : {D : Type} -> D -> (D -> D) -> Nat -> D
recNat-zero : recNat z s zero  = z
recNat-succ : recNat z s (succ n) = s (recNat z s n)

indNat : (Q : Nat -> Type)  -> Q zero  -> ((n : Nat) -> Q n -> Q (succ n))  -> (n : Nat)  -> Q n
indNat-zero : indNat Q qz qs zero = qz
indNat-succ : indNat Q qz qs (succ n) = qs n (indNat Q qz qs n)
```

With this, we can write the programs from before entirely in terms of universal properties:

```
isZeroN : Nat -> Bool
isZeroN = recNat true (λ _ -> false)

zUnit : (n : Nat) -> n + zero = n
zUnit = indNat (λ n -> n + zero = n)  recNat-zero  (λ n ih -> recNat-succ • ap succ ih)
```

Because `Bin` is initial for the same functor, it carries the same universal property, the same recursor and eliminator, now out of `Bin`. Repair is then a simple instantiation of the universal property at the new initial algebra: `recNat`/`indNat` become `recBin`/`indBin` and `zero`/`succ` become `bzero`/`binSucc`, while the programs' defining data is untouched.

```
recBin : {D : Type} -> D -> (D -> D) -> Bin -> D
indBin : (Q : Bin -> Type)  -> Q bzero  -> ((b : Bin) -> Q b -> Q (binSucc b))  -> (b : Bin) -> Q b

isZeroB : Bin -> Bool
isZeroB = recBin true (λ _ -> false)

zUnitB : (b : Bin) -> b + bzero = b
zUnitB = indBin  (λ b -> b + bzero = b)  recBin-zero (λ b ih -> recBin-succ • ap binSucc ih)
```

The algebraic perspective on proof repair is epitomized by the preceding example; the two very different representations of natural numbers are initial for the same functor and hence must have a unique isomorphism between them. This is our thesis emerging: `Nat` and `Bin` are repaired precisely through this algebra isomorphism. We spend the rest of this section cementing this idea, and establishing that our slogan (proof repair is an algebra isomorphism) is general and concrete.

3.2 Connecting Configurations and Algebras

Type equivalences are an unsatisfactory answer to the question of what configurations capture. Indeed, because type equivalences don't capture the inductive structure of configurations, our proof of Proposition 2.3 supposed an inductive algebra to generate a configuration from an equivalence. We utilize P-algebras to put our understanding of configurations on firmer ground. The main idea of this section is that configurations are a syntax-sensitive method for capturing a shared inductive

structure on types, which can be represented by algebra isomorphisms. We begin to develop this idea by coaxing configurations into something less burdened by syntax.

Every configuration has a canonical form that we call *W-type configurations*. These configurations emulate W-types by capturing the inductive structure of types with a single constructor that takes a non-recursive and recursive argument.

Definition 3.9 (W-type Configurations ✓)**.** For every polynomial $(A : \mathsf{Type},\ B : A \to \mathsf{Type})$, we define a one-constructor *W-type signature*: a single operation whose arity supplies a shape $a : A$ and then a $B\,a$-indexed family of recursive children.

$$\mathsf{WTypeArity} : \mathbb{1} \to \mathsf{ConstrArity}$$
$$\mathsf{WTypeArity}\ A\ B\ \_ =_{\mathrm{def}} \mathsf{Nonrec}\ A\ (\lambda a \to \mathsf{Rec}\ (B\ a)\ \mathsf{Done})$$

$$\mathsf{WTypeSignature} : \mathsf{Signature}$$
$$\mathsf{WTypeSignature}\ \_\ \_\ .\mathsf{Signature.Op} =_{\mathrm{def}} \mathbb{1}$$
$$\mathsf{WTypeSignature}\ A\ B\ .\mathsf{Signature.arity} =_{\mathrm{def}} \mathsf{WTypeArity}\ A\ B$$

$$\mathsf{WTypeIndAlg} : (C : \mathsf{Type}) \to \mathsf{Type}$$
$$\mathsf{WTypeIndAlg}\ C =_{\mathrm{def}} \mathsf{IndAlg}\ (\mathsf{WTypeSignature}\ A\ B)\ C$$

A *W-type configuration* is the special configuration where the signature is a W-type signature, collapsing all per-constructor data into one constructor with one non-recursive argument and one recursive argument. For some carriers $C,\ D : \mathsf{Type}$,

$$\mathsf{WTypeConfig} =_{\mathrm{def}} \mathsf{Config}\ (\mathsf{WTypeSignature}\ A\ B)\ C\ D$$

Whereas configurations are useful in capturing the syntactic detail of particular representations of inductive types, W-type configurations are a representation that collapses that syntactic detail which is useful for reasoning about the underlying structure configurations capture. Indeed, we prove every configuration is represented by a corresponding W-type configuration.

Lemma 3.10 (Configurations Faithfully Embed into W-type Configurations ✓)**.** *Every configuration over an arbitrary signature sig determines a W-type configuration on the same carriers. We read off the polynomial whose shapes pair a constructor label with its non-recursive data and whose positions are the recursive arguments occurring in that shape:*

$$A =_{def} (c : \mathrm{Op}) \times \mathrm{Args}(\mathrm{arity}\ c)\ \mathbb{1}, \qquad B\,(c, s) =_{def} \mathrm{recPos}(\mathrm{arity}\ c)\ s,$$

and the configuration transports to a W-type configuration for this (A, B):

$$\mathsf{Config}\ \mathrm{sig}\ C\ D \to (A : \mathsf{Type}) \times ((B : (A \to \mathsf{Type})) \times (\mathsf{WTypeConfig}\ C\ D))$$

The embedding is faithful: writing $\varphi, \psi : C \leftrightarrows D$ for the forward and backward maps of the repair equivalence induced by the configuration (Proposition 2.3), and φ_W, ψ_W for those induced by its W-type configuration, the maps are equal:

$$\varphi_W = \varphi \qquad \textit{and} \qquad \psi_W = \psi.$$

Consequently, nothing about repair is lost by passing to the canonical form, and we may state our main theorem over W-type configurations.

Theorem 3.11 (W-type Configurations are Unique Algebra Equivalences ✓)**.** *For any polynomial $(A : \mathsf{Type},\ B : A \to \mathsf{Type})$ and carriers $C,\ D : \mathsf{Type}$, a W-type configuration is exactly an algebra equivalence between algebras on C and D whose source is initial:*

$$\begin{aligned}\mathsf{WTypeConfig}\ A\ B\ C\ D \ \simeq\ & (\sigma_C : P\ C \to C)\\ & \times (\sigma_D : P\ D \to D)\\ & \times \mathsf{AlgEquiv}\ (C, \sigma_C)\ (D, \sigma_D)\\ & \times \mathsf{isInitAlg}\ (C, \sigma_C)\end{aligned}$$

Proof sketch. The carrier-level WTypeIndAlg $A\,B\,X \simeq$ InitAlgOn $A\,B\,X$ is given by the homotopy-initiality and W-type correspondence of Awodey et al. [12], which we then use to get an equivalence between the corresponding pairs. We construct an equivalence with a pair of initial algebras and a single initial algebra together with an algebra equivalence between the two. Composing these equivalences yields the desired conclusion. That the equivalence is unique ✓ is given by the universal property and shown by Awodey et al. [12]. □

This result gives precise type-theoretic answers to the two questions raised in Section 2. To the first — when is repair "natural"? — it shows that a configuration is strictly more than a type equivalence: it witnesses that its carriers are initial for a common functor, each via a chosen structure map. Repair is therefore natural precisely when the proof engineer equips the source and target with natural structure maps and the correspondence used for repair is then not an arbitrary equivalence but the algebra isomorphism that initiality forces to exist uniquely between them. There are also unnatural structure maps one could impose on a carrier. What the algebraic view does is localize the naturalness question to a universal construction, a structure map that presents an initial algebra. A configuration is a collection of separately-specified data each of which can be independently unnatural, while the algebraic reformulation collapses all of it to a well-understood object, the structure map. With this in mind, we can phrase the naturalness of repair as a formal question for the proof engineer: is there an obvious, or natural, shared algebraic structure between the datatypes? Without the algebraic viewpoint, the only questions we could ask were informal.

To the second — what changes can configurations handle? — it shows that they handle exactly those datatype changes representable by an algebra isomorphism. This class is broad in practice, since the initial algebra of a functor pins the data structure the programmer has in mind, so changes of representation that preserve that structure fall within its reach. We caution that this characterization has force only relative to a fixed polynomial: quantified over all polynomials it trivializes, since any equivalence $C \simeq D$ is an algebra isomorphism for a degenerate constant polynomial, so the class of repairable changes is delimited by the same single choice — the structure map — to which we localized naturalness above.

Example 3.12 (Motivating Example with Algebra Equivalence ✓)**.** Because `Nat` and `Bin` are initial for the same functor, initiality hands us a homomorphism each way and forces their two composites to be endomorphisms of an initial algebra, hence the identities. This gives the unique algebra isomorphism `toNat/fromNat : Nat ≃ Bin` Now the swap of initial algebras from the previous example is the action of this isomorphism. The new recursor `recBin` $z\ s$ precomposed with `fromNat` is again a homomorphism so by initiality of `Nat` it must coincide with `recNat` $z\ s$; the swap is forced, not chosen. Each old program is thus equal to the repaired program conjugated by the isomorphism:

```
recNatSwap : recNat z s = recBin z s ∘ fromNat

isZeroCoh : (n : Nat) -> isZeroN n = isZeroB (fromNat n)
isZeroCoh n = happly recNatSwap n
```

The same story plays out one level up for proofs: the dependent eliminators commute with the isomorphism (elim_ok), so the repaired proof transports back to the original along the round-trip witness `toNatFromNat`. Writing `zUnitB` for the `Bin`-side proof obtained by the swap (ranging over `toNat` b, so the predicate stays on the `Nat` side), we have

```
zUnitB : (b : Bin) -> toNat b + zero = toNat b
zUnitCoh : (n : Nat) -> tpt (λ m -> m + zero = m) (toNatFromNat n) (zUnitB (fromNat n)) = zUnit n
```

Repair across `Nat` and `Bin` is, in this precise sense, the unique algebra isomorphism between them: the proof engineer's swap of constructors and eliminators is exactly its action on terms, and the coherence conditions of Proposition 2.4 are precisely the witnesses that this action sends each original term to its repair. However, the naive use of an algebra isomorphism is not repair since it

does not rid the code of the old dependency. Shortly, we characterize the ability to get rid of old datatypes in terms of P-algebras.

3.3 Algebraic Repair

Now, we can formulate repair entirely in P-algebraic terms.

Definition 3.13 (Algebra Configuration ✓)**.** Fix a polynomial (A : Type, $B : A \to$ Type) and carriers C, D : Type. An *initial algebra structure* on C packages an algebra with a proof of initiality:

$$\mathsf{InitAlgOn}\, A\, B\, C \;=_{\mathrm{def}}\; (\sigma : P\, C \to C) \times \mathsf{isInitAlg}\,(C, \sigma).$$

An *algebra configuration* between C and D is a pair of initial algebra structures on each carrier:

$$\mathsf{AlgConfig}\, A\, B\, C\, D \;=_{\mathrm{def}}\; \mathsf{InitAlgOn}\, A\, B\, C \;\times\; \mathsf{InitAlgOn}\, A\, B\, D.$$

An algebra configuration is deliberately the algebraic analogue of a configuration (Definition 2.2): where a configuration carries an inductive algebra on each carrier, an algebra configuration carries an initial P-algebra on each carrier.

An algebra configuration determines an algebra isomorphism between its two carriers by the initiality of each side, which we refer to as the repair map. Forgetting the homomorphism witnesses and keeping the underlying carrier maps recovers exactly the type equivalence $C \simeq D$ of Proposition 2.3; the algebraic statement is the upgrade, an equivalence whose maps preserve the inductive structure rather than a bare bijection of carriers.

This algebraic reformulation is more than a change of vocabulary. The algebraic picture replaces the syntax-heavy bookkeeping of a configuration with a single mathematical object, a pair of initial algebras for one polynomial functor. This puts the foundations of proof repair on well-trodden mathematical ground, and clarifies what repair computes. It equally clarifies the proof engineer's task. The engineer needs only the common algebraic structure shared by the source and target, then the repair, together with its correctness, is determined. The payoff of the slogan that proof repair is an algebra isomorphism is that it names the goal of a repair algorithm precisely and grounds it in a mathematical fact independent of any particular implementation.

The coherence conditions are subsumed by this viewpoint. In the configuration setting of Proposition 2.4, the three conditions are properties one must establish of the induced map. From an algebra configuration they fall out of initiality, and our development derives all three from the two initial algebras alone ✓. Commutation with the constructors (constr_ok) is given by the homomorphism square and the remaining two rest on the source being canonically generated by its constructors, which for an initial algebra is exactly its dependent eliminator and the eliminator's computation rule. This is the precise content of the foreshadowing in Section 2.

Proposition 3.14 (Algebra Config iff Configuration ✓)**.** *Over carriers* C : Type, D : Type*, write*

$$\mathsf{SigConfig}\, C\, D =_{def} (\mathsf{sig} : \mathsf{Signature}) \times \mathsf{Config}\, \mathit{sig}\, C\, D$$

$$\mathsf{PAlgConfig}\, C\, D =_{def} (A : \mathsf{Type}) \times (B : A \to \mathsf{Type}) \times \mathsf{AlgConfig}\, A\, B\, C\, D$$

for a configuration and algebra configuration over a signature. The two notions are logically equivalent:

$$\mathsf{SigConfig}\, C\, D \;\leftrightarrow\; \mathsf{PAlgConfig}\, C\, D$$

Proof sketch. Both maps are the carrier-level bridge of Awodey et al. [12] applied on each carrier, which identifies a W-type inductive algebra with an h-initial algebra. This is only a logical equivalence; the backward map always lands on a W-type signature rather than the original *sig*. Restricting to a W-type signature upgrades it to the type equivalence of Theorem 3.11. □

In our example and other repair case studies, the repaired program is always the *same* program, with the universal property instantiated at the target, with the rest of the data untouched. The algebraic framework lets us formulate this into a general statement.

Proposition 3.15 (Repair by Instantiation ✓)**.** *Fix an algebra configuration between C and D with repair map φ, and let (Y, γ) be* any *P-algebra. Writing $\mathsf{fold}_C\,\gamma : C \to Y$ and $\mathsf{fold}_D\,\gamma : D \to Y$ for the unique homomorphisms out of each initial algebra,*

$$\mathsf{fold}_C\,\gamma \;=\; \mathsf{fold}_D\,\gamma \circ \varphi.$$

Proof. Both sides are algebra homomorphisms $(C, \sigma_C) \to (Y, \gamma)$, and the hom-space out of an initial algebra is contractible. □

A program written against the universal property consists of defining data γ together with an instantiation of fold at one initial algebra. By Proposition 3.15, repair is re-instantiation: replace fold_C by fold_D, keeping γ. The repaired term $\mathsf{fold}_D\,\gamma$ is built from D's universal property and γ alone, so it mentions the source type C exactly when γ does. When the program is written generically, its defining data is carrier-free (like the recursor cases `true` and $\lambda\_ \to$ `false`), and the repaired term then mentions C nowhere. When γ does mention C, those occurrences are further uses of C's algebra structure, repairable by another instance of the same swap. Whether a term mentions C is a property of its syntax, which the type theory cannot express internally, but this gives a mathematical justification for the ability of repair to get rid of the source type.

Proposition 3.15 folds into a fixed algebra (Y, γ), covering recursion but not induction, whose motive $P : C \to \mathsf{Type}$ mentions the carrier. It applies anyway, since induction is a fold into the total-space algebra on $(x : C) \times P\,x$, with the eliminator its first-projection section ✓. Because the motive mentions C, the target is transported along the repair map (unique by Theorem 3.11) rather than shared. Projecting the resulting equation of pairs onto the proof component is exactly `elim_ok` (Proposition 2.4), which is why that condition carries transported cases $\varphi^* cs$. The transported motive still mentions C, but only through constructors and folds, themselves repairable by further swaps; iterating over motive and cases yields the native D-side proof, as in `zUnitB` of Section 2.

We view Proposition 3.15 as stating that repair exploits a kind of *representation independence* [9] for algebras in type theory. Programs and proofs written over a specific datatype are independent of the specific implementation of that datatype and are only sensitive to the underlying algebra, which is the key idea behind repair. This is the reason repair tools can operate by syntactic instantiation by putting the programs defining data into the same universal property, but for a different carrier.

3.4 Discussion: Repair without Univalence

While our results have made real use of the homotopy interpretation of the intensional identity type, nowhere have we relied upon univalence: the entire inductive development is carried out in intensional Martin-Löf type theory with function extensionality as the only axiom (Section 1). This is notable because prior proof repair work was largely framed within a univalent context, where the correctness conditions were thought to require univalence [44]. Our development thus broadens the class of type theories in which this kind of proof repair is known to be justified. For example, our results apply to settings where univalence is inconsistent, like a type theory with uniqueness-of-identity proofs, such as Lean [2].

4 Dualizing Repair: Coinductive Types and Final Coalgebras

With the algebraic perspective on proof repair in hand, we are able to dualize our development to coinductive types and final coalgebras. Coinductive types are how proof assistants model infinite data and ongoing processes and verified developments over them face the same burden of representation change that motivated repair for inductive types. Yet no existing repair tool addresses them. In this section, we work within Cubical Agda [53], which supports a notion of equality that is amenable to coinductive types and bisimulation.

Instead of being defined in terms of constructors, coinductive types are defined in terms of destructors, meaning the principal method with which programs interact with coinductive data

is by observation. Moreover, coinductive types enable reasoning about infinite objects through the principle of coinduction [29]. A classic infinite object in computer science and mathematics literature is a stream, a non-empty infinite sequence over a given type:

$$(a_0, a_1, a_2, \cdots) : \mathsf{Stream}\, A \qquad A : \mathsf{Type} \qquad a_i : A.$$

The coinductive type of streams is defined as follows:

```
record Stream (A : Type) : Type where
  coinductive
  field
    hd : A
    tl : Stream A
```

We define programs that construct streams by specifying the observations that can be made of the stream — the head and the tail — a style known as copattern matching [5]. Here is the program that adjoins an element to the beginning of a stream:

```
cons : A -> Stream A -> Stream A
hd (cons a s) = a
tl (cons a s) = s
```

Using the principle of coinduction, we may verify that any stream can be decomposed into `cons` applied to its head and tail. In Cubical Agda even this proof is built by copattern matching, a point we return to at the end of this subsection:

```
streamComp : (s : Stream A) -> cons (hd s) (tl s) = s
hd (streamComp s i) = hd s
tl (streamComp s i) = tl s
```

Just like inductive types, coinductive types that capture the same data structure can have different definitions. For example, a library may also define the type of streams with a single destructor that packages the head and tail:

```
record Stream' (A : Type) : Type where
  coinductive
  field
    obs : A × Stream' A
```

Our goal in this section is to lay the foundations for proof repair algorithms that can handle changes between coinductive types, such as the change from `Stream` to `Stream'`. We will work backwards, utilizing the coalgebraic perspective on coinductive types to develop repair in terms of universal properties, and then extract a notion of configurations for coinductive types that exposes syntactic detail for an implementation.

4.1 Warming up to Coalgebra

We begin by dualizing the algebraic setup in Section 3 to coalgebras. We internalize the definitions given in standard sources [6, 27, 48] on P-coalgebras within the type theory of Cubical Agda.

Definition 4.1 (Coalgebra ✓)**.** Given a polynomial endofunctor $P : \mathsf{Type} \to \mathsf{Type}$, a *coalgebra* for P is a pair (X, γ) where $X : \mathsf{Type}$ and $\gamma : X \to P\,X$.

$$\text{P-Coalg} : \mathsf{Type}$$
$$\text{P-Coalg} =_{\text{def}} (A : \mathsf{Type}) \times (A \to P\,A)$$

Example 4.2 (Stream Coalgebra ✓)**.** Fix a $A : \mathsf{Type}$ and consider the functor

$$P : \mathsf{Type} \to \mathsf{Type} \qquad P\,X =_{\text{def}} A \times X$$

A coalgebra for this functor is a pair (Z, γ) where $Z : \mathsf{Type}$ and $\gamma : Z \to A \times Z$. We can understand this as a collection of streams of type A with a dynamics given by γ, which is really given by

$$\langle \gamma_1, \gamma_2 \rangle : (Z \to A) \times (Z \to Z)$$

via the universal mapping property of products. Thus, we have a "stream system" given by a collection of streams, an output function γ_1 (which gives the head of a stream), and a transition function γ_2 (which gives the tail of a stream).

Remark 4.3. Fix A : Type and $B : A \to$ Type. Because the definition of polynomial endofunctor (Definition 3.1) is the dependent sum given by A and B (the polynomial given by A and B), we can always present a coalgebra structure map $\gamma : Z \to P\,A\,B\,Z$ as a *shape observation* $\gamma_1 : Z \to A$ and a *position observation* $\gamma_2 : (z : Z) \to B\,(\gamma_1\,z) \to Z$.

Definition 4.4 (Coalgebra Homomorphism ✓)**.** Given two coalgebras (X, γ) and (Y, δ) for a polynomial endofunctor P : Type $\to$ Type, a *coalgebra homomorphism* from (X, γ) to (Y, δ) is a function $f : X \to Y$ such that the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{\gamma} & PX \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle Pf} \\ Y & \xrightarrow{\delta} & PY \end{array}$$

Formally, we define the type of coalgebra homomorphisms as the function space between the carriers paired with the proof of the above commutative diagram:

$$\begin{aligned} &\text{isCoalgHom} : (X\ Y : \text{P-Coalg}\,A\,B) \to (\text{Carrier}\,X \to \text{Carrier}\,Y) \to \text{Type} \\ &\text{isCoalgHom}\,(X, \gamma)\,(Y, \delta)\,f =_{\text{def}} \text{P-map}\,f \circ \gamma = \delta \circ f \\ &\text{CoalgHom} : \text{P-Coalg}\,A\,B \to \text{P-Coalg}\,A\,B \to \text{Type} \\ &\text{CoalgHom}\,X\,Y =_{\text{def}} (f : \text{Carrier}\,X \to \text{Carrier}\,Y) \times \text{isCoalgHom}\,X\,Y\,f \end{aligned}$$

Example 4.5 (Stream Homomorphism ✓)**.** Homomorphisms between coalgebras are maps that preserve the dynamics of the source coalgebra. This means a homomorphism of stream coalgebras preserves outputs (heads) and transitions (tails):

$$\begin{array}{ccc} Z & \xrightarrow{\quad h \quad} & Y \\ \downarrow{\scriptstyle \langle \gamma_1, \gamma_2 \rangle} & & \downarrow{\scriptstyle \langle \sigma_1, \sigma_2 \rangle} \\ A \times Z & \xrightarrow{1_A \times h} & A \times Y \end{array}$$

Coalgebras support a notion of finality dual to the initiality of algebras. We introduce two notions that will become useful for our development: a final coalgebra, whose universal property pins down only the underlying function of the unique homomorphism, and a homotopy-final coalgebra, which additionally pins down the homomorphism witness.

Definition 4.6 (Final Coalgebra ✓)**.** A coalgebra X for a polynomial endofunctor P is *final* if for every coalgebra Y there exists a coalgebra homomorphism $Y \to X$, and the underlying function of that homomorphism is unique.

$$\begin{aligned} &\text{isFinal} : \text{P-Coalg}\,A\,B \to \text{Type} \\ &\text{isFinal}\,X =_{\text{def}} (Y : \text{P-Coalg}\,A\,B) \to (h : \text{CoalgHom}\,Y\,X) \\ &\qquad\qquad \times \left((h' : \text{CoalgHom}\,Y\,X) \to \pi_1(h) = \pi_1(h')\right) \end{aligned}$$

Definition 4.7 (Homotopy-Final Coalgebra ✓)**.** A P-coalgebra X is *homotopy-final* if for every P-coalgebra C the type of coalgebra homomorphisms $Y \to X$ is contractible.

$$\begin{aligned} &\text{isHFinal} : \text{P-Coalg}\,A\,B \to \text{Type} \\ &\text{isHFinal}\,X =_{\text{def}} (Y : \text{P-Coalg}\,A\,B) \to \text{isContr}(\text{CoalgHom}\,Y\,X) \end{aligned}$$

The reason for distinguishing between these two notions of finality is that proof assistant support for coinductive types that have homotopy-finality is not fully developed. Cubical Agda

formalizations have confirmed finality [18] for containers, but whether coinductive records in Agda have homotopy-finality is open. As far as we know, whether coinductive types in Rocq have finality is open, and Lean would trivially have homotopy-finality (given finality) since it supports the uniqueness of identity proofs axiom. State of the art results [8, 24] formulate M-types internally to type theory, but not in a fashion that is suitable for repair of native definitions. We return to this point when developing repair for coinductive types.

Example 4.8 (Final Stream Coalgebra ✓)**.** The `Stream` type is the final coalgebra for the stream functor $P\,S =_{\text{def}} A \times S$. Its coalgebra structure is just the two destructors bundled up, unfolding a stream into its head and tail:

```
outStream : Stream X -> P X 𝟙 (Stream X)
outStream s = (hd s , λ _ -> tl s)
```

The universal property of final coalgebras is what enables the construction of coprograms. The homomorphism from a coalgebra to a final coalgebra gives coinductive types a corecursion principle, a method for defining functions from any coalgebra to a final coalgebra by specifying how the function interacts with the destructors of the coalgebra. For `Stream`, the head is read off from γ, and the tail corecurses on the unfolded position:

```
Stream-corec : (C : P-Coalg X 𝟙) -> CoalgCarrier C -> Stream X
hd (Stream-corec C c) = fst (CoalgOut C c)
tl (Stream-corec C c) = Stream-corec C (snd (CoalgOut C c) tt)
```

That `Stream-corec` C is a homomorphism holds definitionally and it is the unique such function.

The verification of finality for `Stream` requires reasoning about the equality of two streams to prove the homomorphism uniqueness. Equality for the infinite systems given by coalgebra (or coinductive types) requires a notion of indistinguishable behavior that is not captured by the usual intensional identity type: that type is inductively generated by refl, so its only closed proofs identify terms that are already definitionally equal, and it offers no way to build an identification of two infinite objects from an observation-by-observation comparison. To understand identity on coalgebra, we must define an adequate behavior preserving relation on coalgebra.

Definition 4.9 (Bisimulation ✓)**.** Fix a coalgebra $(X, \langle \gamma_1, \gamma_2 \rangle)$ for a polynomial endofunctor $P\,A\,B$ and a span $R : X \to X \to \mathsf{Type}$. The terms $x, y : X$ satisfy the one-step condition $\mathsf{Bisim}\,R\,x\,y$ when they have equal shape observations and R-related position observations.

$$\mathsf{Bisim}\,\mathsf{R} : (x\,y : X) \to \mathsf{Type}$$

$$\mathsf{Bisim}\,R\,x\,y =_{\text{def}} (\text{b-shape} : \gamma_1\,x = \gamma_1\,y) \times \big((b : B\,(\gamma_1\,x)) \to R\,(\gamma_2\,x\,b)\,(\gamma_2\,y\,b^*)\big)$$

where b^* indicates a transport across b-shape.

The span R is a *bisimulation* when every R-related pair satisfies this one-step condition.

$$\mathsf{isBisim} : (R : X \to X \to \mathsf{Type}) \to \mathsf{Type}$$

$$\mathsf{isBisim}\,R =_{\text{def}} \forall\{x\,y\} \to R\,x\,y \to \mathsf{Bisim}\,R\,x\,y$$

Terms $x, y : X$ are then bisimilar when they are related by some bisimulation, some R with $\mathsf{isBisim}\,R$ and $R\,x\,y$. That this notion of bisimulation is a bisimulation in the categorical sense of Rutten [48] is proven in the subsequent proposition.

Proposition 4.10 (Coinduction Principle for Final Coalgebra ✓)**.** *Fix a final coalgebra (X, γ). The coinduction principle says that bisimilar elements are equal. Given a relation R that is a bisimulation, every R-related pair is identified by a path.*

$$\mathsf{coind} : (R : X \to X \to \mathsf{Type}) \to \mathsf{isBisim}\,R \to \forall\{x\,y\} \to R\,x\,y \to x = y$$

Proof Sketch. This is the argument of Rutten [48], repeated in type theory by Ahrens et al. [8]. The argument goes by forming the coalgebra on $\widehat{R} =_{\text{def}} \Sigma\, x\, y.\, R\, x\, y$, whose projections are coalgebra morphisms into (X, γ), which must be the same, giving the desired identity by evaluating at x, y. □

Example 4.11 (Bisimulation on Streams ✓)**.** For streams the bisimulation data specialises to a relation R relating streams whose heads agree and whose tails are R-related:

```
record StreamBis (R : Stream X -> Stream X -> Type) (s t : Stream X) where
  field
    bhd : hd s = hd t
    btl : R (tl s) (tl t)
```

Coinduction then turns any such bisimulation into a path, defined by copattern matching on the path's head and tail:

```
Stream-coind : (R : Stream X -> Stream X -> Type)
            -> (∀ {s t} -> R s t -> StreamBis R s t)
            -> ∀ {s t} -> R s t -> s = t
hd (Stream-coind R isBis r i) = bhd (isBis r) i
tl (Stream-coind R isBis r i) = Stream-coind R isBis (btl (isBis r)) i
```

The ability to construct proofs of identity via copattern matching as in Examples 4.8 and 4.11 is the principal reason for our choice of Cubical Agda. The path type of cubical type theory is not an inductively generated type; it is defined as a function space out of the interval, which facilitates introduction by means other than refl. On the other hand, the usual identity type is amenable to the sort of breaking-down of structure that pattern matching on inductive types provides. Indeed, the bisimulation and the path type are equivalent in Cubical Agda (coind is an equivalence) [53], giving a type theory where equality is suited for coalgebraic reasoning.

4.2 Coalgebraic Repair

Finally, we have the tools in hand to cast coinductive repair in terms of final coalgebras. The goal of repair is always to establish a relationship across which types can be repaired. Coalgebra gives us the definitive notion of a relationship between coalgebras: our goal is to define a homomorphism between the coalgebras corresponding to `Stream` and `Stream'` so that we can transfer the programs and proofs between the two. However, as in the inductive case, the proof engineer would like an efficient repair mechanism that removes the source type as a dependency. We develop this in two parts: first with an algebraic configuration, and then with an algorithm-friendly configuration.

Definition 4.12 ((Homotopy) Coalgebra Configuration ✓)**.** Fix a polynomial $(A : \mathsf{Type}, B : A \to \mathsf{Type})$. A final coalgebra structure on a carrier C packages a destructor map with a proof that the resulting coalgebra is final:

$$\mathsf{FinalCoalgOn}\, A\, B\, C =_{\text{def}} (\gamma : C \to P\, A\, B\, C) \times \mathsf{isFinal}\, (C, \gamma).$$

A coalgebra configuration between carriers C and D is a pair of final coalgebra structures for the same polynomial, one on each carrier:

$$\mathsf{CoalgConfig}\, A\, B\, C\, D =_{\text{def}} \mathsf{FinalCoalgOn}\, A\, B\, C \;\times\; \mathsf{FinalCoalgOn}\, A\, B\, D.$$

A coalgebra configuration determines a coalgebra isomorphism: each side's finality yields a unique homomorphism into the other, and the two round-trips are forced to be the identities by uniqueness. A homotopy coalgebra configuration is given by replacing the finality condition with isHFinal.

Now, we can use such a configuration to do the swap induced by the underlying equivalence on programs and proofs, rather than compose with the equivalence.

Example 4.13 (Coalgebraic Repair with Stream and Stream' ✓)**.** The configuration equips each carrier with the same universal property, so a program written with the corecursor and a proof written with the coinduction principle are repaired by swapping which universal property they are

instantiated. The defining data of cons mentions its carrier only through the structure map, so it can be stated once, against an abstract coalgebra (E, out):

```
consSeed : P-Coalg X 𝟙 -> P-Coalg X 𝟙
consSeed (E , out) = ((X × E) ⊎ E , γ)
  where γ (inl (y , t)) = (y , λ _ -> inr t)
        γ (inr t)     = (fst (out t) , λ _ -> inr (snd (out t) tt))

cons-M : X -> Stream X -> Stream X
cons-M x s = Stream-corec (consSeed (Stream X , outStream )) (inl (x , s))

cons'-M : X -> Stream' X -> Stream' X
cons'-M x s = Stream'-corec (consSeed (Stream' X , outStream')) (inl (x , s))

streamComp-M : (s : Stream X) -> cons (hd s) (tl s) ≡ s
streamComp-M s = Stream-coind etaR etaStep (refl , refl)

streamComp'-M : (s : Stream' X) -> cons' (π₁ (obs s)) (π₂ (obs s)) ≡ s
streamComp'-M s = Stream'-coind etaR' etaStep' (refl , refl)
```

This is dual to the observation after Proposition 3.15: occurrences of the source type in defining data are further uses of its (co)algebra structure, repaired by the same swap. The bisimulations etaR/etaR′ likewise read their carriers only through the structure maps. The repaired program and proof carry no reference to the source type.

4.3 Coinductive Repair via Configurations

With the coalgebraic perspective on coinductive types, we can now develop the framework for repair algorithms for coinductive types. Akin to the standard presentation of W-types, we give coinductive types a set of type-theoretic rules, and refer to types satisfying these rules as M-types. The state of the art work on M-types (in the formulation of rules) has not established their homotopy-finality. In fact, the contemporary work has focused on proving the finality and homotopy-finality for various formulations of M-types [8, 18, 24], none of which are a rule-based presentation. We are primarily interested in developing repair for coinductive types, and thus are concerned with rule-based presentations that facilitate programming and proving. To develop repair for contemporary and future type theories, our development handles both a notion of M-types that correspond to final coalgebras and a stronger notion that correspond to homotopy-final coalgebras.

Definition 4.14 (M-Types ✓)**.** An M-type for a polynomial $(A : \mathrm{Type}, B : A \to \mathrm{Type})$ is a carrier $\mathsf{M}\,A\,B$ equipped with destructors

$$\mathsf{shape} : \mathsf{M}\,A\,B \to A \qquad \mathsf{pos} : (x : \mathsf{M}\,A\,B) \to B\,(\mathsf{shape}\,x) \to \mathsf{M}\,A\,B$$

satisfying the following rules (Figure 1). The corecursor maps any coalgebra into $\mathsf{M}\,A\,B$, and its two computation (β) rules say that destructing a corecursive value computes (the position rule lives over the shape rule, hence a dependent path). Finally, the uniqueness (η) rule states that any map agreeing with the source coalgebra under the destructors equals the corecursor. We bundle these rules — corec, the two β rules, and η — for a carrier X with destructors $\mathsf{shape} : X \to A$ and $\mathsf{pos} : (x : X) \to B\,(\mathsf{shape}\,x) \to X$ into a single type $\mathsf{HasMRules}\,X\,\mathsf{shape}\,\mathsf{pos}$, so that possessing the M-type structure is the property of inhabiting this type.

Every M-type enjoys a coinduction principle, which says that any bisimulation entails a path. Unlike corec/β/η, however, this is not assumed as a rule; it is derivable, so HasMRules carries no coinduction data ✓.

We emphasize that M-types are not a primitive of our ambient theory: HasMRules is a property that a given carrier with given destructors may or may not enjoy, and asserting that some type has it is a theorem obligation, not a rule of the type theory. There exists an M-type for any polynomial

$$\frac{s : C \to A \qquad p : (c : C) \to B\,(s\,c) \to C}{\mathrm{corec}\,s\,p : C \to \mathsf{M}\,A\,B}\ \mathrm{corec} \qquad \frac{}{\beta_s : \mathrm{shape}\,(\mathrm{corec}\,s\,p\,c) = s\,c}\ \beta_s$$

$$\frac{}{\beta_p : \mathrm{pos}\,(\mathrm{corec}\,s\,p\,c)\ =_{\beta_s}\ \mathrm{corec}\,s\,p \circ (p\,c)}\ \beta_p$$

$$\frac{\begin{array}{c} h : C \to X \\ \beta_s^h : (c : C) \to \mathrm{shape}\,(h\,c) = s\,c \qquad \beta_p^h : (c : C) \to \mathrm{pos}\,(h\,c) = h \circ (p\,c) \end{array}}{\eta : h = \mathrm{corec}\,s\,p}\ \eta$$

$$\frac{h : C \to X \qquad \beta_h : (c : C) \to \big(\mathrm{shape}\,(h\,c) = s\,c\big) \times \big(\mathrm{pos}\,(h\,c) = h \circ (p\,c)\big)}{(h, \beta_h)\ =\ (\mathrm{corec}\,s\,p,\ \beta)}\ \eta_\Sigma$$

$$\frac{R : \mathsf{M}\,A\,B \to \mathsf{M}\,A\,B \to \mathrm{Type} \qquad b : \mathrm{isBisim}\,R\,\langle \mathrm{shape}, \mathrm{pos}\rangle \qquad r : R\,x\,y}{\mathrm{coind}\,R\,b\,r : x = y}\ \mathrm{coind}$$

Fig. 1. Rules of the (homotopy) M-type structure over a polynomial (A, B): the corecursor corec, its computation rules β_s and β_p, the function-level uniqueness rule η and its Σ-level strengthening η_Σ, and the derived coinduction principle coind.

$(A : \mathrm{Type}, B : A \to \mathrm{Type})$, and any M-type has a coalgebra given by $\langle \mathrm{shape}, \mathrm{pos}\rangle$, which enables the formation of bisimulations on M-types, as in the derived coind principle.

Concretely, we use the coinductive records of Cubical Agda to model these M-types and verify that they satisfy the given rules ✓.

```
record M (A : Type) (B : A -> Type) : Type where
  coinductive
  field
    shape : A
    pos   : B shape -> M A B

corec : (C : Type) (s : C -> A) (p : (c : C) -> B (s c) -> C) -> C -> M A B
shape (corec C s p c) = s c
pos   (corec C s p c) b = corec C s p (p c b)

M-HasMRules : HasMRules (M A B) shape pos
M-HasMRules = isFinal→HasMRules M-isFinal

M-coind = HasMRules-coind M-HasMRules
```

Now, we develop M-types that directly correspond to homotopy-final coalgebras.

Definition 4.15 (Homotopy M-Types ✓)**.** A homotopy M-type strengthens the η rule of an M-type from the function level to the Σ level: a homomorphism (h, β_h) agreeing with the source coalgebra under the destructors equals the corecursor bundle $(\mathrm{corec}\,s\,p, \beta)$ by a Σ-path (η_Σ in Figure 1). Exactly as HasMRules bundles the plain rules, we bundle the M-rules with η_Σ in place of η into a single type HasHMRules X shape pos.

A homotopy M-type requires just this one additional rule over an M-type. Note that while the existence of homotopy M-types in HoTT is known via the limit construction of Ahrens et al. [8], our development and prior work does not prove that they exist within Cubical Agda. The proof that the native `M` record is a homotopy M-type requires characterizing `M` path types at higher dimensions

which the state of the art [18, 24] does not do, and, in fact, there is a documented issue that our attempted formalization faced [1]. The obstruction disappears under additional hypotheses on the polynomial. In fact, the Stream type is a set when its underlying type is ✓ a set. This fact holds true for M-types when the shape type is a set as well, but we don't pursue the general statement here.

Configurations expose the syntactic detail of a particular representation that a repair algorithm walks over. We give a configuration framework for coinductive types by focusing on destructors and outputs. We introduce the destructor-level signature and algebra, then define a single carrier's coinductive structure, and finally give a configuration as a pair of these.

Definition 4.16 (Destructor Algebra ✓)**.** The output telescope of a single destructor is described by a *destructor arity*:

$$\begin{aligned}\mathsf{Done} &: \mathsf{DestrArity}\\ \mathsf{Nonrec} &: (A : \mathsf{Type}) \to (A \to \mathsf{DestrArity}) \to \mathsf{DestrArity}\\ \mathsf{Rec} &: (D : \mathsf{Type}) \to \mathsf{DestrArity} \to \mathsf{DestrArity}.\end{aligned}$$

Reading an arity off at a carrier X produces the type of *outputs* of such a destructor:

$$\begin{aligned}&\mathsf{Outputs}\ \mathsf{Done}\ X =_{\mathrm{def}} \mathbb{1}\\ &\mathsf{Outputs}\ (\mathsf{Nonrec}\ A\ k)\ X =_{\mathrm{def}} (a : A) \times \mathsf{Outputs}\ (k\ a)\ X\\ &\mathsf{Outputs}\ (\mathsf{Rec}\ D\ cs)\ X =_{\mathrm{def}} (D \to X) \times \mathsf{Outputs}\ cs\ X.\end{aligned}$$

This action is functorial in the carrier via mapOutputs. A CoSignature *sig* is a type Op of destructor labels together with an arity assignment arity : Op → DestrArity. A *destructor algebra* for *sig* on a carrier X interprets every label as a destructor exposing its outputs:

$$\mathsf{DestrAlgebra}\ sig\ X =_{\mathrm{def}} (op : \mathsf{Op}) \to \big(X \to \mathsf{Outputs}\ (\mathsf{arity}\ op)\ X\big).$$

Definition 4.17 (Coinductive Configuration ✓)**.** A *coinductive algebra* carries a destructor algebra equipped with corecursion and computation rules. Formally, a CoindCoalg for *sig* on X packages:

- destr : DestrAlgebra *sig* X, the syntactic coalgebra on X;
- corec : DestrAlgebra *sig* $C \to C \to X$ for any source carrier C, the corecursor;
- corec-β, the per-destructor computation rule: for every source coalgebra δ and label op,
$$\mathsf{destr}\ op\ (\mathsf{corec}\ \delta\ c) \;=\; \mathsf{mapOutputs}\ (\mathsf{arity}\ op)\ (\mathsf{corec}\ \delta)\ (\delta\ op\ c);$$
- corec-η, function-level uniqueness: any $h : C \to X$ whose destructors agree with δ agrees with corec δ.

Again, in Cubical Agda the coinduction principle coind is derivable from these fields, so CoindCoalg already carries it implicitly. A *coinductive configuration* for *sig* between carriers C and D is a pair of coinductive coalgebras, one on each carrier:

$$\mathsf{CoConfig}\ sig\ C\ D =_{\mathrm{def}} \mathsf{CoindCoalg}\ sig\ C \;\times\; \mathsf{CoindCoalg}\ sig\ D.$$

Coinductive configurations admit a homotopy strengthening just like the M-type formulations.

Definition 4.18 (Homotopy Coinductive Configuration ✓)**.** An HCoindCoalg replaces the function-level corec-η with the Σ-level uniqueness rule η_Σ of a homotopy M-type (corec-uniq-Σ). A homotopy coinductive configuration is a pair of these:

$$\mathsf{HCoConfig}\ sig\ C\ D =_{\mathrm{def}} \mathsf{HCoindCoalg}\ sig\ C \;\times\; \mathsf{HCoindCoalg}\ sig\ D,$$

As with inductive configurations, we can verify that the coinductive configurations satisfy the correctness conditions a proof engineer would expect of a repair algorithm. These conditions are new correctness criteria for coinductive repair, but they follow the idea that the configuration behaves coherently with the underlying equivalence.

Proposition 4.19 (Correctness Conditions ✓). *Fix a coinductive configuration* CoConfig *sig C D with destructor coalgebras* $\mathsf{destr}_C, \mathsf{destr}_D$, *corecursors* $\mathsf{corec}_C, \mathsf{corec}_D$, *and* β *laws. Let* $\varphi : C \to D$ *be the forward map of the equivalence induced by the configuration. It satisfies the coinductive duals of the PUMPKIN Pi coherence conditions.*

(1) destr_ok. *The destructors commute with* φ*: for every label* op *and* $c : C$,

$$\mathsf{destr}_D\ op\ (\varphi\ c)\ =\ \mathsf{mapOutputs}\ (\mathsf{arity}\ op)\ \varphi\ (\mathsf{destr}_C\ op\ c).$$

Since $\varphi = \mathsf{corec}_D\ \mathsf{destr}_C$ *by construction, this is exactly its per-destructor* β *rule.*

(2) iota_ok. *Each corecursor computes on its own destructors: this is exactly* corec-β*, one law for* corec_C *and one for* corec_D.

(3) corec_ok. *The corecursors commute with* φ*: for every source coalgebra* ec : DestrAlgebra *sig E and* $e : E$,

$$\varphi\ (\mathsf{corec}_C\ ec\ e)\ =\ \mathsf{corec}_D\ ec\ e.$$

(4) coind_ok. *The coinduction principles commute with* φ*: writing* $\mathsf{coind}_C, \mathsf{coind}_D$ *for coinduction on each side, for every bisimulation* R *on* C *and* $r : R\ x\ y$,

$$\mathsf{ap}\ \varphi\ (\mathsf{coind}_C\ r)\ =\ \mathsf{coind}_D\ (\varphi_* r),$$

an equality of the parallel paths $\varphi\ x \equiv \varphi\ y$, *where* φ_* *transports the* C*-bisimulation to a* D*-bisimulation. It only holds for a homotopy configuration (*HCoConfig*).*

Proof sketch. All four follow directly from the structure the configuration carries. The interesting part is coind_ok. The bulk of this proof is deriving the two-dimensional coinduction principle from the uniqueness of corecursors (corec-uniq-Σ) ✓. □

The first three conditions are the duals of constr_ok, iota_ok, and elim_ok from Proposition 2.4, obtained by the same recipe. Among them, corec_ok deserves emphasis: it is repair by instantiation (Proposition 3.15) in its coinductive form — the exact dual of $\mathsf{fold}_C\ \gamma = \mathsf{fold}_D\ \gamma \circ \varphi$ — witnessed concretely for streams by corecSwap'. It can take the source coalgebra *ec* verbatim because that defining data lives on a third type E and its type never mentions the repaired carriers, just as the type of a fold's target algebra γ does not (so corecursion, too, introduces no new occurrences of the source type); by contrast, the data of elim_ok (a motive over C) and of coind_ok below (a bisimulation on C) mention the carrier itself, and so must be transported along φ rather than shared. The fourth condition, coind_ok, is the one condition that demands the homotopy tier. Streams over an h-set assemble into an HCoConfig ✓, so coind_ok holds in particular for the running Stream/Stream' example whenever the element type is a set.

Example 4.20 (Stream and Stream' Repair via Configurations ✓)**.** Now we can phrase our running example through coinductive configurations.

```
StreamSig : CoSignature
StreamSig = MTypeSignature X 𝟙

streamCoindCoalg : CoindCoalg StreamSig (Stream X)
streamCoindCoalg = coalgBridgeBwd (outStream , Stream-isFinal)
stream'CoindCoalg : CoindCoalg StreamSig (Stream' X)
stream'CoindCoalg = coalgBridgeBwd (outStream' , Stream'-isFinal)

streamCoConfig : CoConfig StreamSig (Stream X) (Stream' X)
streamCoConfig = record { coindCoalgC = streamCoindCoalg; coindCoalgD = stream'CoindCoalg }
```

Given any source coalgebra, unfold ind corecurses into that component's carrier. Written against the abstract CoindCoalg, it names neither Stream nor Stream'.

```
unfold : CoindCoalg StreamSig E -> DestrAlgebra StreamSig S -> S -> E
unfold ind = CoindCoalg.corec ind
```

Repair is now executed through a swap of the configuration's components. `consVia` writes `cons` against an abstract component. Instantiating at `streamCoindCoalg` yields `cons` on `Stream`; the repair is the *same program* instantiated at the swapped component, now landing in `Stream'`.

```
consVia : CoindCoalg StreamSig E -> X -> E -> E
consVia ind x e = unfold ind δ (inl (x , e))
  where δ op (inl (y , t)) = y , (λ _ -> inr t) , tt
        δ op (inr t)    = mapOutputs _ inr (destr ind op t)

consCfg : X -> Stream X -> Stream X
consCfg = consVia streamCoindCoalg
consCfg' : X -> Stream' X -> Stream' X
consCfg' = consVia stream'CoindCoalg
```

The proof `streamComp` is repaired the same way, through the coinduction principle each component carries (Definition 4.17).

```
streamCompCfg : (s : Stream X) -> cons (hd s) (tl s) ≡ s
streamCompCfg s = deriveCoind streamCoindCoalg etaR etaRIsBisim (refl , refl)

streamCompCfg' : (s : Stream' X) -> cons' (π₁ (obs s)) (π₂ (obs s)) ≡ s
streamCompCfg' s = deriveCoind stream'CoindCoalg etaR' etaRIsBisim' (refl , refl)
```

This models the execution of a proof repair algorithm for coinductive types on the running programs and proofs. These repaired terms compute as we would expect. Observing one reduces the configuration scaffolding away, leaving native `Stream'` data. Each equation below holds definitionally.

```
consCfg'-head : (x : X) (s : Stream' X) -> π₁ (obs (consCfg' x s)) ≡ x
consCfg'-next : (x : X) (s : Stream' X) -> π₁ (obs (π₂ (obs (consCfg' x s)))) ≡ π₁ (obs s)
```

4.4 Connecting Coinductive Configurations and Coalgebra

We now characterize M-types as carriers of final coalgebras, setting the foundation for cementing our slogan that proof repair is an algebra isomorphism in the case of coinductive repair. We give the coalgebraic dual of Awodey et al.'s [12] characterisation of W-types as homotopy-initial algebras. The novelty is the left-hand side of the equivalence below, a rule-based presentation of (homotopy) M-types, whereas prior treatments establish finality for coalgebraic or limit constructions of M-types [8, 18, 24], not for such an interface.

Theorem 4.21 ((Homotopy) M-Types are (Homotopy) Final Coalgebras)**.** *Fix a polynomial* $(A : \mathsf{Type}, B : A \to \mathsf{Type})$*. On a carrier* $X : \mathsf{Type}$ *with destructors* $(s : X \to A,\ p : (x : X) \to B\,(s\,x) \to X)$*, having the (homotopy) M-rules is exactly the induced coalgebra* $(X, \langle s, p\rangle)$ *being (homotopy-)final.*

(1) (Final ✓) $\mathsf{HasMRules}\,X\,s\,p \ \simeq\ \mathsf{isFinal}\,(X, \lambda x.\,(s\,x, p\,x))$

(2) (Homotopy-final ✓) $\mathsf{HasHMRules}\,X\,s\,p \ \simeq\ \mathsf{isHFinal}\,(X, \lambda x.\,(s\,x, p\,x))$

Proof sketch. The corecursor together with its two β rules is exactly a coalgebra homomorphism out of any source, and η is exactly function-level uniqueness of that homomorphism, and conversely rec, β_s, β_p, and η are read back off the finality data. Both round-trips hold definitionally. The homotopy-final tier strengthens η from function-level uniqueness to contractibility of the coalgebra-homomorphism space. Contractibility transports either way and both sides are propositions, so the bi-implication is again an equivalence. □

Coinductive configurations have a canonical form we call *M-type configurations*. These configurations emulate M-types by capturing the coinductive structure of types with a single destructor that exposes a non-recursive and a recursive output.

Definition 4.22 (M-type Configuration ✓)**.** For every polynomial $(A : \mathsf{Type},\ B : A \to \mathsf{Type})$, we define a one-destructor M-type signature: a single observation whose output telescope exposes a

shape $a : A$ and then a $B\,a$-indexed family of recursive positions.

$$\mathsf{MTypeArity} : \mathbb{1} \to \mathsf{DestrArity} \qquad \mathsf{MTypeSignature} : \mathsf{CoSignature}$$
$$\mathsf{MTypeArity}\,A\,B\,\_ =_{\mathrm{def}} \mathsf{Nonrec}\,A\,(\lambda a \to \mathsf{Rec}\,(B\,a)\,\mathsf{Done})$$
$$\mathsf{MTypeSignature}\,A\,B\,.\mathsf{CoSignature.Op} =_{\mathrm{def}} \mathbb{1}$$
$$\mathsf{MTypeSignature}\,A\,B\,.\mathsf{CoSignature.arity} =_{\mathrm{def}} \mathsf{MTypeArity}\,A\,B$$
$$\mathsf{MTypeCoindCoalg} : (C : \mathsf{Type}) \to \mathsf{Type}$$
$$\mathsf{MTypeCoindCoalg}\,C =_{\mathrm{def}} \mathsf{CoindCoalg}\,(\mathsf{MTypeSignature}\,A\,B)\,C$$

An M-type configuration is the special coinductive configuration whose signature is an M-type signature, collapsing all per-destructor data into one destructor with one non-recursive output and one recursive output. For some carriers C : Type, D : Type, MTypeCoConfig $=_{\mathrm{def}}$ CoConfig (MTypeSignature $A\,B$) $C\,D$.

M-type configurations collapse the detail of a configuration into the canonical one-destructor form over which we identify configurations with coalgebra isomorphisms. Every coinductive configuration is faithfully represented by an M-type configuration.

The theorem below is stated with the coinductive dual of Definition 3.6, in two tiers matching the two notions of finality. A *coalgebra equivalence* CoalgEquiv $X\,Y$ ✓ is a type equivalence of the carriers whose forward and backward maps both carry coalgebra-homomorphism witnesses. Its homotopy equivalent is $\mathsf{CoalgEquiv}^{\mathsf{H}}$ ✓ is the exact dual of Definition 3.6.

Lemma 4.23 (Coinductive Configurations Faithfully Embed into M-type Configurations ✓)**.** *Every coinductive configuration over an arbitrary coinductive signature sig determines an M-type configuration on the same carriers. We read off the polynomial from the signature:*

$$A =_{\mathit{def}} (op : \mathrm{Op}) \to \mathrm{Outputs}(\mathrm{arity}\,op)\ \mathbb{1}, \qquad B\,a =_{\mathit{def}} (op : \mathrm{Op}) \times \mathrm{recPos}(\mathrm{arity}\,op)\,(a\,op),$$

and the configuration converts to an M-type configuration for this (A, B):

$$\mathsf{CoConfig}\,\mathit{sig}\,C\,D \to (A : \mathsf{Type}) \times ((B : (A \to \mathsf{Type})) \times \mathsf{MTypeCoConfig}\,C\,D).$$

The embedding is faithful: the forward and backward maps of the repair equivalence induced by the M-type configuration are equal to those induced by the original configuration.

Theorem 4.24 (Coinductive Configurations are Coalgebra Isomorphisms ✓)**.** *Over the M-type signature, a coinductive configuration is a coalgebra isomorphism whose source is final.*

(1) (Final) A configuration is logically equivalent to a coalgebra isomorphism between final coalgebras:

$$\mathsf{CoConfig}\,(\mathsf{MTypeSignature}\,A\,B)\,C\,D \longleftrightarrow (\gamma_C : C \to P\,A\,B\,C) \times (\gamma_D : D \to P\,A\,B\,D) \times \mathsf{CoalgEquiv}\,(C, \gamma_C)\,(D, \gamma_D) \times \mathsf{isFinal}\,(C, \gamma_C)$$

(2) (Homotopy-final) A homotopy configuration is a coalgebra isomorphism between homotopy-final coalgebras:

$$\mathsf{HCoConfig}\,(\mathsf{MTypeSignature}\,A\,B)\,C\,D \simeq (\gamma_C : C \to P\,A\,B\,C) \times (\gamma_D : D \to P\,A\,B\,D) \times \mathsf{CoalgEquiv}^{\mathsf{H}}\,(C, \gamma_C)\,(D, \gamma_D) \times \mathsf{isHFinal}\,(C, \gamma_C)$$

Proof sketch. The first link is a carrier-level bridge given by our characterisation of (homotopy) M-types as (homotopy) final coalgebras (Theorem 4.21). We then identify a pair of final coalgebras with one final coalgebra plus a coalgebra isomorphism to the other. The two directions are mutually inverse exactly when the matched data is propositional. This holds at the homotopy-final tier, giving an equivalence. It fails at the final tier, where isFinal doesn't pin the homomorphism witness. □

This theorem is the coinductive completion of our slogan, paralleling Theorem 3.11: a coinductive configuration is strictly more than a type equivalence — it witnesses that its carriers are final for a

common polynomial functor. Repair between coinductive types is therefore natural precisely when the types have natural presentations as final coalgebras for the same functor, and the correspondence used for repair is the coalgebra isomorphism that finality forces to exist uniquely between them. Uniqueness of repair dualizes as well: at the homotopy-final tier a coalgebra isomorphism out of a final source is a proposition (as the proof above observes), so a homotopy coinductive configuration, too, determines exactly one repair. It equally delimits the changes coinductive configurations can handle: exactly those representable by a coalgebra isomorphism, the changes of destructor presentation that preserve the observable behaviour of the type.

It is important that this development largely rests upon the discovery that proof repair for inductive types corresponds to algebra isomorphisms. Before establishing the connection to repair and P-algebras, the extension of proof repair to coinductive types was not at all clear because it was muddled by the many different implementation approaches taken by various proof assistants, which don't really capture what a coinductive type is. By focusing on the universal properties that coinductive types have, we are able to construct a high-level foundation for proof repair algorithms in a general manner, independent of any specific implementation approach.

5 Related Work

Proof Repair. Our work gives the first foundational treatment of proof repair. Proof repair was first introduced in 2018 [45, 47]. It is the proof engineering [43] analogue of program repair [10, 33]. The class of proof repair algorithms that our foundations support comes from a line of work on the automatic repair of datatypes in Rocq in response to breaking changes [42, 44, 46, 54]. That line of work focuses on algorithmic approaches, but leaves open questions about what a configuration represents and what makes a configuration "natural." The thesis work in particular [42] presents a diagram that gives intuition that inspired our foundations, but goes no further in developing that intuition. We formalize this line of work and answer these open questions. The supplementary automation beyond the core repair functionality, like decompilation, are out of the scope of our foundations.

Other proof repair algorithms exist that are out of scope of our foundations. For example, Sisyphus [23] repairs separation logic proofs about imperative programs written in OCaml. Proof Mate [32] repairs proofs written in PVS. Preliminary work explores proof repair for JML [35]. All of these tools concern specification and proof languages that are quite different from the dependent type theory that we concern ourselves with.

Recently, neural methods for proof repair have begun to emerge, both as standalone tools [37, 38, 40, 41, 51, 55, 57] and as a way of augmenting other models for proofs [21, 28, 39]. The scope of these methods is defined by their training data, rather than by any foundational system. In the long run, foundations like ours may help in defining kinds of data that these tools may wish to support, or in building well-scoped and general neurosymbolic tools that integrate symbolic repair into other neural models (in the fashion of PALM [31]).

Proof Reuse and Transfer. Proof repair is one way of reusing proofs, particularly in the presence of imperfect foresight or dependencies outside of one's control. The key conditions that yield repair rather than the more general reuse are that (1) proof repair is primarily concerned with adapting proofs to changes rather than to reusing proofs across developments, and (2) proof repair is most useful in the presence of imperfect foresight or breaking changes in dependencies that are outside of one's control. Giving up the first condition yields tools for proof reuse, like refinement frameworks [17] and transfer methods [16, 26, 49, 50]; giving up the second condition yields design principles that aim to make proofs reusable or robust to changes to begin with [15, 19, 43, 56].

The relationships between these tools and technologies have been discussed extensively in prose [16, 42, 43, 54]. Our foundations add a formal dimension to this discussion. By formalizing the connection between equivalences, configurations, and algebras, we draw an explicit bridge

between transfer methods whose inner workings use automatic transport. Moreover, with Proposition 3.15, we learn that proof repair satisfies its distinguishing conditions by exploiting a kind of representation independence [9]. This property may also hold of other "white-box" approaches to automatic transfer [50], which can be used for repair.

(Co)algebras and (Co)inductive types. We briefly cover previous work on (co)algebras and (co)inductive types from which our development greatly benefited, with no claims to completeness.

Inductive types are a well-researched area within programming languages and type theory. The seminal work of Dybjer [20] gives a schema for inductive and recursive definitions within MLTT. The work of Chapman et al. [14] presents a dependent type theory which gives inductive definitions via an encoding in a universe. The connection with inductive datatypes, algebras, and polynomial endofunctors has received significant attention. The primary inspiration for our work is the discovery by Awodey et al. [12] that the rules for W-types are equivalent to the existence of homotopy-initial algebras for polynomial endofunctors. Abbott et al. [4] show that the notion of containers (similar to polynomial functors) can be used to represent all strictly positive types and the polymorphic functions between them. Damato et al. [18] generalise the container development to a setting without the uniqueness of identity proofs axiom, giving a formulation in intensional type theory and formalization in Cubical Agda.

Coinductive types and coalgebras are a prominent and growing research area. The work of Hagino [25] utilizes categorical logic to develop a simply typed lambda calculus with algebraic and coalgebraic datatypes. Basold and Geuvers [13] extend Hagino's categorical datatypes to a dependent type theory constructed solely from inductive and coinductive types, while leaving the (co)induction principles out. Ahrens et al. [8] establish that the existence of M-types (in the formulation of homotopy-final coalgebras) is derivable in HoTT in the presence of inductive types. In the course of developing models of non-wellfounded sets in HoTT, Gylterud et al. [24] characterize the identity type of the M-type (formulated as a homotopy-final coalgebra) as an indexed M-type. Most modern proof assistants feature some form of coinductive types, which has been made possible by various research programs [5, 22, 30].

6 Conclusion

We have given the first foundational treatment of type-theoretic proof repair. Its fundamental construct, the configuration, is exactly an algebra isomorphism between initial algebras for a common polynomial functor (Theorem 3.11). This discovery greatly clarifies the foundational understanding of repair: the coherence conditions imposed by prior tools are consequences of initiality rather than side obligations, the repair equivalence a configuration determines is unique, and programs written against the universal property are repaired by re-instantiation alone (Proposition 3.15). Dualizing this account yielded the foundations of repair for a class of types no existing tool handles—coinductive types—together with new correctness criteria for coinductive repair and a novel characterization of (homotopy) M-types as (homotopy-)final coalgebras (Theorem 4.21).

Two aspects of the development deserve emphasis. First, none of it uses univalence: the inductive side lives in intensional MLTT with function extensionality, and the coinductive side uses Cubical Agda only for its path type, not for univalent principles. Since prior repair work was framed univalently, this broadens the class of type theories in which repair is known to be justified. Second, the algebraic framing is extensible. Higher inductive types, quotient inductive types, inductive-inductive types, and inductive-recursive types all admit algebraic presentations over richer signatures, and we expect configurations, their coherence conditions, and the uniqueness of repair to transfer along the same lines. We hope this foundation serves both as an explanation of why existing repair tools work and as a blueprint for the repair tools that do not yet exist.

References

[1] 2020. *cubical primitives should preserve guardedness · Issue #4740 · agda/agda.* https://github.com/agda/agda/issues/4740

[2] 2026. *Propositional Equality.* https://lean-lang.org/doc/reference/latest/Basic-Propositions/Propositional-Equality/#UIP

[3] 2026. *The Rocq Prover 9.2 documentation.* https://rocq-prover.org/doc/V9.2.0/refman/index.html

[4] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3–27. doi:10.1016/j.tcs.2005.06.002 Applied Semantics: Selected Topics.

[5] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2429069.2429075

[6] Peter Aczel and Nax Mendler. 1989. A final coalgebra theorem. In *Category Theory and Computer Science* (Berlin, Heidelberg, 1989), David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné (Eds.). Springer, 357–365. doi:10.1007/BFb0018361

[7] Jiří Adámek, Stefan Milius, and Lawrence S. Moss. 2025. *Initial Algebras and Terminal Coalgebras: The Theory of Fixed Points of Functors.* Cambridge University Press.

[8] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. 2015. Non-Wellfounded Trees in Homotopy Type Theory. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 38)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 17–30. doi:10.4230/LIPIcs.TLCA.2015.17

[9] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing representation independence with univalence. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 12 (2021). doi:10.1145/3434293

[10] Ashif Anwar. 2026. Neuro Symbolic Automated Program Repair: A Systematic Review of LLM-Based and Symbolic Techniques. *International Journal of Computer Engineering and Data Science (IJCEDS)* 5, 1 (Mar. 2026), 1–16. https://www.ijceds.com/ijceds/article/view/118

[11] Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., USA.

[12] Steve Awodey, Nicola Gambino, and Kristina Sojakova. 2017. Homotopy-Initial Algebras in Type Theory. 63, 6 (2017), 1–45. doi:10.1145/3006383

[13] Henning Basold and Herman Geuvers. 2016. Type Theory based on Dependent Inductive and Coinductive Types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 327–336. doi:10.1145/2933575.2934514

[14] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1863543.1863547

[15] 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

[16] 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. doi:10.1007/978-3-031-57262-3_10

[17] 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. doi:10.1007/978-3-319-03545-1_10

[18] Stefania Damato, Thorsten Altenkirch, and Axel Ljungström. 2025. Formalising Inductive and Coinductive Containers. doi:10.48550/arXiv.2409.02603 arXiv:2409.02603 [cs.LO]

[19] 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 2011)*. ACM, New York, NY, USA, 595–608. doi:10.1145/2048066.2048113

[20] Peter Dybjer. 1994. Inductive families. *Form. Asp. Comput.* 6, 4 (July 1994), 440–465. doi:10.1007/BF01211308

[21] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. doi:10.1145/3611643.3616243

[22] Eduardo Giménez. 1994. Codifying Guarded Definitions with Recursive Schemes. In *Selected Papers from the International Workshop on Types for Proofs and Programs (TYPES '94)*. Springer-Verlag, Berlin, Heidelberg, 39–59.

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

[24] Hakon Robbestad Gylterud, Elisabeth Stenholm, and Niccolò Veltri. 2026. Terminal Coalgebras and Non-wellfounded Sets in Homotopy Type Theory. Volume 22, Issue 2 (2026), 14325. doi:10.46298/lmcs-22(2:35)2026 arXiv:2001.06696 [math.LO]

[25] Tatsuya Hagino. 1987. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 140–157.

[26] 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. doi:10.1007/978-3-319-03545-1_9

[27] Bart Jacobs and Jan Rutten. 2011. *An introduction to (co)algebra and (co)induction.* Cambridge University Press, 38–99.

[28] Prithwish Jana, Kaan Kale, Ahmet Ege Tanriverdi, Cruise Song, Sriram Vishwanath, and Vijay Ganesh. 2026. ProofBridge: Auto-Formalization of Natural Language Proofs in Lean via Joint Embeddings. arXiv:2510.15681 [cs.LO] https://arxiv.org/abs/2510.15681

[29] DEXTER KOZEN and ALEXANDRA SILVA. 2017. Practical coinduction. *Mathematical Structures in Computer Science* 27, 7 (2017), 1132–1152. doi:10.1017/S0960129515000493

[30] Bastiaan Laarakker, Daniël Otten, and Benno van den Berg. 2026. Constructing (Co)inductive Types via Large Sizes. doi:10.48550/arXiv.2602.18921 arXiv:2602.18921 [cs.LO]

[31] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. doi:10.1145/3691620.3695521

[32] Paolo Masci and Aaron Dutle. 2022. Proof Mate: An Interactive Proof Helper for PVS (Tool Paper). In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings* (Pasadena, CA, USA). Springer-Verlag, Berlin, Heidelberg, 809–815. doi:10.1007/978-3-031-06773-0_44

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

[34] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[35] Amirfarhad Nilizadeh, Gary T. Leavens, and David R. Cok. 2023. Automated Reasoning Repair. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs* (Berlin, Germany) *(FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 11–14. doi:10.1145/3611096.3611099

[36] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (Savannah, GA, USA) *(TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/1481861.1481862

[37] Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. 2026. APOLLO: Automated LLM and Lean Collaboration for Advanced Formal Reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=fxDCgOruk0

[38] Manooshree Patel, Bartosz Piotrowski, Leopold Haller, and Hugh James Leather. 2026. ProofRepairBench: Exploring Proof Repair in Lean. In *ICLR 2026 Workshop: VerifAI-2: The Second Workshop on AI Verification in the Wild.* https://openreview.net/forum?id=6SwWVNwEJK

[39] Santhana Srinivasan R and Maithilee Patawar. 2026. LAMP: Lean-based Agentic framework with MCP and Proof Repair. arXiv:2606.28841 [cs.LO] https://arxiv.org/abs/2606.28841

[40] Thomas Reichel. 2024. *Neural approaches to theorem search & proof repair.* Master's thesis. University of Illinois at Urbana-Champaign.

[41] Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. 2023. Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:20. doi:10.4230/LIPIcs.ITP.2023.26

[42] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington. https://digital.lib.washington.edu/researchworks/handle/1773/47429

[43] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.* 5, 2–3 (Sept. 2019), 102–281. doi:10.1561/2500000045

[44] 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 112–127. doi:10.1145/3453483.3454033

[45] 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. doi:10.1145/3167094

[46] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:19. doi:10.4230/LIPIcs.ITP.2019.26

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

[48] J. J. M. M. Rutten. 2000. Universal coalgebra: a theory of systems. 249, 1 (2000), 3–80. doi:10.1016/S0304-3975(00)00056-6

[49] 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. doi:10.1145/3236787

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

[51] Amer Tahat, David Hardin, Adam Petz, and Perry Alexander. 2024. Proof repair utilizing large language models: A case study on the copland remote attestation proofbase. In *International Conference on Bridging the Gap between AI and Reality*. Springer, 145–166.

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

[53] 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 (July 2019), 29 pages. doi:10.1145/3341691

[54] Cosmo Viola, Max Fan, and Talia Ringer. 2025. Proof Repair across Quotient Type Equivalences. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 386 (Oct. 2025), 26 pages. doi:10.1145/3763164

[55] Evan Wang, Simon Chess, Daniel Lee, Siyuan Ge, Ajit Mallavarapu, Jarod Alper, and Vasily Ilin. 2026. Learning to Repair Lean Proofs from Compiler Feedback. arXiv:2602.02990 [cs.LG] https://arxiv.org/abs/2602.02990

[56] 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. doi:10.1145/2854065.2854081

[57] Jun Yang, Yuechun Sun, Yi Wu, Rodrigo Caridad, Yongwei Yuan, Jianan Yao, Shan Lu, and Kexin Pei. 2026. ExVerus: Verus Proof Repair via Counterexample Reasoning. In *Forty-third International Conference on Machine Learning*. https://openreview.net/forum?id=FNDkXA0OUJ