Abstract
We describe a new approach to automatically repairing broken proofs in the Coq proof assistant in response to changes in types. Our approach combines a configurable proof term transformation with a decompiler from proof terms to suggested tactic scripts. The proof term transformation implements transport across equivalences in a way that removes references to the old version of the changed type and does not rely on axioms beyond those Coq assumes.

We have implemented this approach in Pumpkin Pi, an extension to the Pumpkin Patch Coq plugin suite for proof repair. We demonstrate Pumpkin Pi’s flexibility on eight case studies, including supporting a benchmark from a user study, easing development with dependent types, porting functions and proofs between unary and binary numbers, and supporting an industrial proof engineer to interoperate between Coq and other verification tools more easily.

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

Keywords: proof engineering, proof repair, proof reuse

1 Introduction
Program verification with interactive theorem provers has come a long way since its inception, especially when it comes to the scale of programs that can be verified. The seL4 [21] verified operating system kernel, for example, is the effort of a team of proof engineers spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [12] (emphasis ours):

A sufficiently fanatical researcher might be willing to devote two or three years to verifying a significant piece of software if he could be assured that the software would remain stable.

we could argue that, over 40 years, either verification has become easier, or researchers have become more fanatical. Unfortunately, not all has changed (emphasis still ours):

But real-life programs need to be maintained and modified. There is no reason to believe that verifying a modified program is any easier than verifying the original the first time around.

Tools that can automatically refactor or repair proofs [1, 4, 13, 32, 34, 35, 43, 44] give us reason to believe that verifying a modified program can sometimes be easier than verifying the original, even when proof engineers do not follow good development processes, or when change occurs outside of proof engineers’ control [30]. Still, maintaining verified programs can be challenging: it means keeping not just the programs, but also specifications and proofs about those programs up-to-date. This remains so difficult that sometimes, even experts give up in the face of change [31].

The problem of automatically updating proofs in response to changes in programs or specifications is known as proof repair [30, 32]. While there are many ways proofs need to be repaired, one such need is in response to a changed type definition (Section 3). We make progress on two open challenges in proof repair in response to changes in type definitions:

1. Existing work supports very limited classes of these changes like non-structural changes [32] or a predefined set of changes [34, 44], and these are not informed by the needs of proof engineers [31].
2. Proof repair tools are not yet integrated with typical proof engineering workflows [30, 32, 34], and may impose additional proof obligations like proving relations corresponding to changes [35].

The typical proof engineering workflow in Coq is interactive: The proof engineer passes Coq high-level search procedures called tactics (like induction), and Coq responds to each tactic by refining the current goal to some subgoal (like the proof obligation for the base case). This loop of tactics and goals continues until no goals remain, at which point the proof engineer has constructed a sequence of tactics called a proof script. To check this proof script for correctness, Coq compiles it to a low-level representation called a proof term, then checks that the proof term has the expected type.

Our approach to proof repair works at the level of low-level proof terms, then builds back up to high-level proof scripts. In particular, our approach combines a configurable proof term transformation (Section 4) with a prototype decompiler from proof terms back to suggested proof scripts (Section 5). This is implemented (Section 6) in PUMPKIN Pi, an extension to the PUMPKIN PATCH [32] proof repair plugin suite for Coq 8.8 that is available on Github.1

Addressing Challenge 1: Flexible Type Support. The case studies in Section 7—summarized in Table 1—show that PUMPKIN Pi is flexible enough to support a wide range of proof engineering use cases. In general, PUMPKIN Pi can support any change described by an equivalence, though it takes the equivalence in a deconstructed form that we call a configuration. The configuration expresses to the proof term transformation how to translate functions and proofs defined over the old version of a type to refer only to the new version, and how to do so in a way that does not break definitional equality. The proof engine can write this configuration in Coq and feed it to PUMPKIN Pi (manual configuration in Table 1), configuring PUMPKIN Pi to support the change.

Addressing Challenge 2: Workflow Integration. Research on workflow integration for proof repair tools is in its infancy. PUMPKIN Pi is built with workflow integration in mind. For example, PUMPKIN Pi is the only proof repair tool we are aware of that produces suggested proof scripts (rather than proof terms) for repaired proofs, a challenge highlighted in existing proof repair work [32, 34] and in a survey of proof engineering [30]. In addition, PUMPKIN Pi implements search procedures that automatically discover configurations and prove the equivalences they induce for four different classes of changes (automatic configuration in Table 1), decreasing the burden of proof obligations imposed on the proof engineer. Our partnership with an industrial proof engineer has informed other changes to further improve workflow integration (Sections 6.1 and 7).

1We annotate each claim to which code is relevant with a circled number like (1). These circled numbers are links to code, and are detailed in GUIDE.md.

2We use induction instead of pattern matching.
When we change the list type, this proof no longer works. To repair this proof with Pumpkin Pi, we run this command:

```
Repair Old.list New.list in rev_app_distr.
```

assuming the old and new list types from Figure 2 are in modules Old and New. This suggests a proof script that succeeds (in light blue to denote Pumpkin Pi produces it automatically):

```
swap T (l : Old.list T) : New.list T :=
Old.list_rect T (fun (l : Old.list T) => New.list T)
        (fun t _ (IHl : New.list T) => New.cons T t IHl)
l.

Lemma section: ∀ T (l : Old.list T),
swap⁻¹ T (swap T l) = l.
Proof.
intros T l. symmetry. induction l as [ | a l IHl].
- auto.
- simpl rewrite ← H. auto.
Qed.
```

where the dependencies (rev, ++, app_assoc, and app_nil_r) have also been updated automatically ①. If we would like, we can manually modify this to something that more closely matches the style of the original proof script:

```
Proof. (* by induction over x and y *)
intros x. induction x as [ | a l IHl].
- (* both cons: *) simpl rewrite IHl. simpl rewrite app_assoc auto.
- (* x nil: *) induction y0 as [ | a l H].
  + (* y cons: *) simpl rewrite app_nil_r auto.
  + (* y nil: *) auto.
Qed.
```

We can even repair the entire list module from the Coq standard library all at once by running the Repair module command ①. When we are done, we can get rid of Old.list.

The key to success is taking advantage of Coq’s structured proof term language: Coq compiles every proof script to a proof term in a rich functional programming language called Gallina—Pumpkin Pi repairs that term. Pumpkin Pi then decompiles the repaired proof term (with optional hints from the original proof script) back to a suggested proof script that the proof engineer can maintain.

In contrast, updating the poorly structured proof script directly would not be straightforward. Even for the simple proof script above, grouping tactics by line, there are 6! = 720 permutations of this proof script. It is not clear which lines to swap since these tactics do not have a semantics beyond the searches their evaluation performs. Furthermore, just swapping lines is not enough: even for such a simple change, we must also swap arguments, so induction x as [ | a l IHl] becomes induction x as [ | a l IHl]. Robert [34] describes the challenges of repairing tactics in detail.

Pumpkin Pi’s approach circumvents this challenge.

### 3 Problem Definition

Pumpkin Pi can do much more than permute constructors. Given an equivalence between types A and B, Pumpkin Pi repairs functions and proofs defined over A to instead refer to B (Section 3.1). It does this in a way that allows for removing references to A, which is essential for proof repair, since A may be an old version of an updated type (Section 3.2).

#### 3.1 Scope: Type Equivalences

Pumpkin Pi repairs proofs in response to changes in types that correspond to type equivalences [41], or pairs of functions that map between two types and are mutual inverses. When a type equivalence between types A and B exists, those types are equivalent (denoted A ≃ B). Figure 3 shows a type equivalence between the two versions of list from Figure 2 that Pumpkin Pi discovered and proved automatically ①.

①The adjoint follows, and Pumpkin Pi includes machinery to prove it. ②
To give some intuition for what kinds of changes can be described by equivalences, we preview two changes below. See Table 1 on page 11 for more examples.

**Factoring out Constructors.** Consider changing the type \(I\) to the type \(J\) in Figure 4. \(J\) can be viewed as \(J\) with its two constructors \(A\) and \(B\) pulled out to a new argument of type \(\text{bool}\) for a single constructor. With \(\text{PUMPKIN}\ \Pi\), the proof engineer can repair functions and proofs about \(I\) to instead use \(J\), as long as she configures \(\text{PUMPKIN}\ \Pi\) to describe which constructor of \(I\) maps to true and which maps to false. This information about constructor mappings induces an equivalence \(I \simeq J\) across which \(\text{PUMPKIN}\ \Pi\) repairs functions and proofs. File \(\text{\#2}\) shows an example of this, mapping \(A\) to true and \(B\) to false, and repairing proofs of De Morgan’s laws.

**Adding a Dependent Index.** At first glance, the word equivalence may seem to imply that \(\text{PUMPKIN}\ \Pi\) can support only changes in which the proof engineer does not add or remove information. But equivalences are more powerful than they may seem. Consider, for example, changing a list to a length-indexed vector (Figure 5). \(\text{PUMPKIN}\ \Pi\) can repair functions and proofs about lists to functions and proofs about vectors of particular lengths \(\text{\#3}\), since \(\Sigma (l : \text{list } T).\text{length } l = n \simeq \text{vector } T n\). From the proof engineer’s perspective, after updating specifications from \(\text{list}\) to \(\text{vector}\), to fix her functions and proofs, she must additionally prove invariants about the lengths of her lists. \(\text{PUMPKIN}\ \Pi\) makes it easy to separate out that proof obligation, then automates the rest.

More generally, in homotopy type theory, with the help of quotient types, it is possible to form an equivalence from a relation, even when the relation is not an equivalence \(\text{\#2}\). While Coq lacks quotient types, it is possible to achieve a similar outcome and use \(\text{PUMPKIN}\ \Pi\) for changes that add or remove information when those changes can be expressed as equivalences between \(\Sigma\) types or sum types.

### 3.2 Goal: Transport with a Twist

The goal of \(\text{PUMPKIN}\ \Pi\) is to implement a kind of proof reuse known as transport \(\text{\#4}\), but in a way that is suitable for repair. Informally, transport takes a term \(t\) and produces a term \(t’\) that is the same as \(t\) modulo an equivalence \(A \simeq B\). If \(t\) is a function, then \(t’\) behaves the same way modulo the equivalence; if \(t\) is a proof, then \(t’\) proves the same theorem the same way modulo the equivalence.

When transport across \(A \simeq B\) takes \(t\) to \(t’\), we say that \(t\) and \(t’\) are equal up to transport across that equivalence (denoted \(t \equiv_{A \simeq B} t’\)).4 In Section 2, the original append function ++ over \(\text{old}\).\text{list}\) and the repaired append function ++ over

---

4This notation should be interpreted in a metatheory with univalence—a property that Coq lacks—or it should be approximated in Coq. The details of transport with univalence are in Univalent Foundations Program \(\text{\#4}\), and an approximation in Coq is in Tabarue et al. \(\text{\#8}\). For equivalent \(A\) and \(B\), there can be many equivalences \(A \simeq B\). Equality up to transport is across a particular equivalence, but we erase this in the notation.

---

**Figure 4.** The old type \(I\) (left) is either \(A\) or \(B\). The new type \(J\) (right) is \(I\) with \(A\) and \(B\) factored out to \(\text{bool}\) (orange).

**Figure 5.** A vector (bottom) is a list (top) indexed by its length (orange). Vectors effectively make it possible to enforce length invariants about lists at compile time.

New list that \(\text{PUMPKIN}\ \Pi\) produces are equal up to transport across the equivalence from Figure 3, since (by \text{app.ok} \(\text{\#1}\)):

\[
\forall \ T \ (11 : \ \text{old.list } T), \quad \text{swap } T (11 +\!\!+ 12) = (\text{swap } T 11) +\!\!+ (\text{swap } T 12).
\]

The original \text{rev_app.distr} is equal to the repaired proof up to transport, since both prove the same thing the same way up to the equivalence, and up to the changes in ++ and \text{rev}.

Transport typically works by applying the functions that make up the equivalence to convert inputs and outputs between types. This approach would not be suitable for repair, since it does not make it possible to remove the old type \(A\). \(\text{PUMPKIN}\ \Pi\) implements transport in a way that allows for removing references to \(A\)—by proof term transformation.

### 4 The Transformation

At the heart of \(\text{PUMPKIN}\ \Pi\) is a configurable proof term transformation for transporting proofs across equivalences \(\text{\#4}\). It is a generalization of the transformation from an earlier version of \(\text{PUMPKIN}\ \Pi\) called Devoid \(\text{\#3}\), which solved this problem a particular class of equivalences.

The transformation takes as input a deconstructed equivalence that we call a configuration. This section introduces the configuration (Section 4.1), defines the transformation that builds on that (Section 4.2), then specifies correctness criteria for the configuration (Section 4.3). Section 6.1 describes the additional work needed to implement this transformation.

**Conventions.** All terms that we introduce in this section are in the Calculus of Inductive Constructions (CIC\(_\omega\)), the type theory that Coq’s proof term language Gallina implements. CIC\(_\omega\) is based on the Calculus of Constructions (CoC), a variant of the lambda calculus with polymorphism (types that depend on types) and dependent types (types that depend on terms) \(\text{\#9}\). CIC\(_\omega\) extends CoC with inductive types \(\text{\#10}\). Inductive types are defined solely by their constructors (like \text{nil}\) and \text{cons}\) for \text{list}\) and eliminators (like the induction
The dependent constructors

$\text{DepConstr}(1, \text{list } T) t l : \text{list } T :=
\text{Constr}(1, \text{list } T) t l$.

DepConstr(0, list T) : list T := Constr(0, list T).
DepConstr(1, list T) t l : list T := Constr(1, list T) t l.

DepElim(1, P) { nnil, pcons } : P l :=
Elim(1, P) { nnil, pcons }.

DepElim(0, P) { pnil, pcons } : P l :=
Elim(0, P) { pnil, pcons }.

DepElim(0, P) { pnil, pcons } :=
Elim(0, P) { pnil, pcons }.

DepElim(1, P) { nnil, pcons } :=
Elim(1, P) { nnil, pcons }.


(Figure 7. The dependent constructors and eliminators for old (left) and new (right) list, with the difference in orange.)
(propositional equality). That is, two terms \( t \) and \( t' \) of type \( T \) are definitionally equal if they reduce to the same normal form, and propositionally equal if there is a proof that \( t = t' \) using the inductive equality type \( \equiv \) at type \( T \). Definitionally equal terms are necessarily propositionally equal, but the converse is not in general true.

When a datatype changes, sometimes, definitional equalities defined over the old version of that type must become propositional. A naive proof term transformation may fail to generate well-typed terms if it does not account for this. Otherwise, if the transformation transforms a term \( t : T \) to some \( t' : T' \), it does not necessarily transform \( T \) to \( T' \) [39].

\( \eta \) and \( \iota \) describe how to transport equalities. More formally, they define \( \eta \)-expansion and \( \iota \)-reduction of \( A \) and \( B \), which may be propositional rather than definitional, and so must be explicit in the transformation. \( \eta \)-expansion describes how to expand a term to apply a constructor to an eliminator in a way that preserves propositional equality, and is important for defining dependent eliminators [27]. \( \iota \)-reduction (\( \beta \)-reduction for inductive types) describes how to reduce an elimination of a constructor [26].

The configuration for the change from \( \text{l} \text{i} \text{s} \text{t} \) to \( \text{p} \text{a} \text{c} \text{k} \text{e} \text{d} \_ \text{v} \text{e} \text{c} \text{t} \) has propositional \( \iota \eta \). It uses \( \eta \)-expansion for \( \Sigma \):

\[
\text{Eta}(\text{packed\_vect}) := \lambda (s:\text{packed\_vect}). \exists (\pi_1 s) (\pi_2 s).
\]

which is propositional and not definitionial in Coq. Thanks to this, we can forego the assumption that our language has primitive projections (definitional \( \eta \) for \( \Sigma \)).

Each \( \iota \eta \) — one per constructor — describes and proves the \( \iota \)-reduction behavior of \text{DepElim} on the corresponding case. This is needed, for example, to port proofs about unary numbers \( \text{nat} \) to proofs about binary numbers \( N \) (Figure 8). While we can define \text{DepConstr} and \text{DepElim} to induce an equivalence between them \([5]\), we run into trouble reasoning about applications of \text{DepElim}, since proofs about that hold by reflexivity do not necessarily hold by reflexivity over \( N \). For example, in Coq, while \( S (n + m) = S n + m \) holds by reflexivity over \( \text{nat} \), when we define \( + \) with \text{DepElim} over \( N \), the corresponding theorem over \( N \) does not hold by reflexivity.

To transform proofs about \( \text{nat} \) to proofs about \( N \), we must transform \( \text{definitional} \ i \) reduction over \( \text{nat} \) to \( \text{propositional} \ i \)-reduction over \( N \). For our choice of \text{DepConstr} and \text{DepElim}, \( \iota \)-reduction is definitionial over \( \text{nat} \), since a proof of:

\[
\forall P \, p_0 \, p_5 \, n \, , \\
\text{DepElim}(\text{DepConstr}(1, \text{nat}) \, n) \, (P) \, \{ p_0, p_5 \} = p_5 n \, (\text{DepElim}(n, P) \, \{ p_0, p_5 \} )
\]

holds by reflexivity. \( \iota \eta \) for \( \text{nat} \) in the \( S \) case is a rewrite by that proof by reflexivity \([5]\), with type:

\[
\forall P \, p_0 \, p_5 \, n \, (Q : P (\text{DepConstr}(1, \text{nat}) \, n) \rightarrow s), \\
\text{Iota}(1, \text{nat}, Q) : Q (p_5 n (\text{DepElim}(n, P) \, \{ p_0, p_5 \} )) \rightarrow Q (\text{DepElim}(\text{DepConstr}(1, \text{nat}) \, n) \, (P) \, \{ p_0, p_5 \} )
\]

In contrast, \( \iota \) for \( N \) is propositional, since the theorem:

\[
\forall P \, p_0 \, p_5 \, n \, , \\
\text{DepElim}(\text{DepConstr}(1, N) \, n) \, (P) \, \{ p_0, p_5 \} = p_5 n \, (\text{DepElim}(n, P) \, \{ p_0, p_5 \} )
\]

no longer holds by reflexivity. \( \iota \eta \) for \( N \) is a rewrite by the propositional equality that proves this theorem \([5]\), with type:

\[
\forall P \, p_0 \, p_5 \, n \, (Q : P (\text{DepConstr}(1, N) \, n) \rightarrow s), \\
\text{Iota}(1, N, Q) : Q (p_5 n (\text{DepElim}(n, P) \, \{ p_0, p_5 \} )) \rightarrow Q (\text{DepElim}(\text{DepConstr}(1, N) \, n) \, (P) \, \{ p_0, p_5 \} )
\]

By replacing \( \iota \eta \) over \( \text{nat} \) with \( \iota \eta \) over \( N \), the transformation replaces rewrites by reflexivity over \( \text{nat} \) to rewrites by propositional equalities over \( N \). That way, \text{DepElim} behaves the same over \( \text{nat} \) and \( N \).

Taken together over both \( A \) and \( B \), \( \iota \eta \) describes how the inductive structures of \( A \) and \( B \) differ. The transformation requires that \text{DepElim} over \( A \) and \( B \) have the same structure as each other, so if \( A \) and \( B \) themselves have the same inductive structure (if they are \textit{ornaments} \([25]\)), then if \( \iota \) is definitionial for \( A \), it will be possible to choose \text{DepElim} with definitionial \( \iota \) for \( B \). Otherwise, if \( A \) and \( B \) (like \text{nat} and \( N \)) have different inductive structures, then definitionial \( \iota \) over one would become propositional \( \iota \) over the other.

### 4.2 The Proof Term Transformation

Figure 9 shows the proof term transformation \( \Gamma \vdash t \downarrow t' \) that forms the core of PUMPKIN\( P_i \). The transformation is parameterized over equivalent types \( A \) and \( B \) (\textit{Equivalence}) as well as the configuration. It assumes \( \eta \)-expanded functions. It implicitly constructs an updated context \( \Gamma' \) in which to interpret \( t' \), but this is not needed for computation.

The proof term transformation is (perhaps deceptively) simple by design: it moves the bulk of the work into the configuration, and represents the configuration explicitly. Of course, typical proof terms in Coq do not apply these configuration terms explicitly. PUMPKIN\( P_i \) does some additional work using \textit{unification heuristics} to get real proof terms.
The transformation does not fully run the proof term transformation, which transports proofs into this format before running the transformation. It then runs the proof term transformation, which transports proofs across the equivalence that corresponds to the configuration.

**Unification Heuristics.** The transformation does not fully describe the search procedure for transforming terms that *PUMPKIN Pi* implements. Before running the transformation, *PUMPKIN Pi* unifies subterms with particular *A* (fixing parameters and indices), and with applications of configuration terms over *A*. The transformation then transforms configuration terms over *A* to configuration terms over *B*. Reducing the result produces the output term over *B*.

Figure 10 shows this with the list append function ++ from Section 2. To update ++ (top left), *PUMPKIN Pi* unifies Old .list *T* with *A*, and Constr and Elim with DepConstr and DepElim (bottom left). After unification, the transformation recursively substitutes *B* for *A*, which moves DepConstr and DepElim to construct and eliminate over the updated type (bottom right). This reduces to a term with swapped constructors and cases over New.list *T* (top right).

In this case, unification is straightforward. This can be more challenging when configuration terms are dependent. This is especially pronounced with definitional *Eta* and *Iota*, which typically are implicit (reduced) in real code. To handle this, *PUMPKIN Pi* implements custom unification heuristics for each search procedure that unify subterms with applications of configuration terms, and that instantiate parameters and dependent indices in those subterms. The transformation in turn assumes that all existing parameters and indices are determined and instantiated by the time it runs.

**Specifying a Correct Transformation.** The implementation of this transformation in *PUMPKIN Pi* produces a term that Coq type checks, and so does not add to the trusted computing base. As *PUMPKIN Pi* is an engineering tool, there is no need to formally prove the transformation correct.
section: \( \forall (a : A), g (f a) = a \).
retraction: \( \forall (b : B), f (g b) = b \).
constr_ok: \( \forall j \ x_A \ x_B, x_A \equiv_{A=B} x_B \rightarrow 
\text{DepConstr}(j, A) \ x_A \equiv_{A=B} \text{DepConstr}(j, B) \ x_B \).
elim_ok: \( \forall a \ b \ P_a \ P_b \ \overline{f} \overline{g} \ x_B, \overline{a} \equiv_{A=B} \overline{b} \rightarrow 
\text{DepElim}(a, P_a) \overline{f} \overline{a} = \overline{g} \overline{b} \rightarrow \text{DepElim}(b, P_b) \overline{f} \overline{a} \).
eta_ok(A): \( \forall (a : A), \text{Eta}(A) a = a \).
elim_eta(A): \( \forall a \ P_f \overline{f} \rightarrow \text{DepElim}(a, P) \overline{f} : P (\text{Eta}(A) a) \).
eta_ok(A): \( \forall (a : A), \text{Eta}(A) a = a \).

Figure 11. Correctness criteria for a configuration to ensure that the transformation preserves equivalence (left) coherently with equality (right, shown for \( A; B \) is similar). \( f \) and \( g \) are defined in text. \( s, \overline{f}, \overline{x}, \overline{\overline{I}H} \) represent sorts, eliminator cases, constructor arguments, and inductive hypotheses. \( \xi (A, P, j) \) is the type of \( \text{DepElim}(A, P) \) at \( \text{DepConstr}(j, A) \) (similarly for \( B \)).

though doing so would be satisfying. The goal of such a proof would be to show that if \( t \Downarrow t' \) then \( t \) and \( t' \) are equal up to transport, and \( t' \) refers to \( B \) in place of \( A \). The key steps in this transformation that make this possible are porting terms along the configuration (\text{Dep-Const}, \text{Dep-Elim}, \text{Eta}, and \text{Iota}). For metatheoretical reasons, without additional axioms, a proof of this theorem in Coq can only be approximated [38]. It would be possible to generate per-transformation proofs of correctness, but this does not serve an engineering need.

4.3 Specifying Correct Configurations

Choosing a configuration necessarily depends in some way on the proof engineer’s intentions: there can be infinitely many equivalences that correspond to the change, only some of which are useful (for example 7), any \( A \) is equivalent to unit refined by \( A \). And there can be many configurations that correspond to an equivalence, some of which will produce terms that are more useful or efficient than others (consider \text{DepElim} converting through several intermediate types).

While we cannot control for intentions, we can specify what it means for a chosen configuration to be correct: Fix a configuration. Let \( f \) be the function that uses \text{DepElim} to eliminate \( A \) and \text{DepConstr} to construct \( B \), and let \( g \) be similar. Figure 11 specifies the correctness criteria for the configuration. These criteria relate \text{DepConstr}, \text{DepElim}, \text{Eta}, and \text{Iota} in a way that preserves equivalence coherently with equality.

Equivalence. To preserve the equivalence (Figure 11, left), \text{DepConstr} and \text{DepElim} must form an equivalence (section and retraction must hold for \( f \) and \( g \), \text{DepConstr} over \( A \) and \( B \) must be equal up to transport across that equivalence (\text{constr_ok}), and similarly for \text{DepElim} (\text{elim_ok}). Intuitively, \text{constr_ok} and \text{elim_ok} guarantee that the transformation correctly transports dependent constructors and dependent eliminators, as doing so will preserve equality up to transport for those subterms. This makes it possible for the transformation to avoid applying \( f \) and \( g \), instead porting terms from \( A \) directly to \( B \).

Equality. To ensure coherence with equality (Figure 11, right), \text{Eta} and \text{Iota} must prove \( \eta \) and \( i \). That is, \text{Eta} must have the same definitional behavior as the dependent eliminator (\text{elim_eta}), and must behave like identity (\text{eta_ok}). Each \text{Iota} must prove and rewrite along the simplification (refolding [5]) behavior that corresponds to a case of the dependent eliminator (\text{Iota_ok}). This makes it possible for the configuration to avoid applying section and retraction.

Correctness. With these correctness criteria for a configuration, we get the completeness result (proven in Coq 8) that every equivalence induces a configuration. We also obtain an algorithm for the soundness result that every configuration induces an equivalence.

Algorithm to prove section is as follows (retraction is similar): replace \( a \) with \( \text{Eta}(A) a \) by \text{eta_ok}(A). Then, induct using \text{DepElim} over \( A \). For each case \( i \), the proof obligation is to show that \( f (g a) = a \), where \( a \) is \text{DepConstr}(A, i) applied to the non-inductive arguments (by \text{elim_eta}(A)). Expand the right-hand side using \text{Iota}(A, i), then expand it again using \text{Iota}(B, i) (destructing over each \text{eta_ok} to apply the corresponding \text{Iota}). The result follows by definition of \( g \) and \( f \), and by reflexivity.

Automatic Configuration. \textsc{Pumpkin} Pi implements four search procedures for automatic configuration 6. Three of the four procedures are based on the search procedure from \textsc{Devold} [33], while the remaining procedure instantiates the types \( A \) and \( B \) of a generic configuration that can be defined inside of Coq directly.

The algorithm above is essentially what \text{Configure} uses to generate functions \( f \) and \( g \) for these configurations 9, and also generate proofs section and retraction that these functions form an equivalence 10. To minimize dependencies, \textsc{Pumpkin} Pi does not produce proofs of \text{constr_ok} and \text{elim_ok} directly, as stating these theorems cleanly would require either a special framework [38] or a univalent type theory [41]. If the proof engineer wishes, it is possible to prove these in individual cases 8, but this is not necessary in order to use \textsc{Pumpkin} Pi.
(\nu) \in \text{Vars}, (\tau) \in \text{CIC}_{\omega}

(t) ::= \text{intro} (\nu) \mid \text{rewrite} \langle \tau \rangle \langle \tau \rangle \mid \text{symmetry} \mid \text{apply} \langle \tau \rangle \mid \text{induction} \langle \tau \rangle \langle \tau \rangle \{ (p), \ldots, (p) \} \mid \text{split} \{ (p), (p) \} \mid \text{left} \mid \text{right} \mid \langle p \rangle \langle p \rangle

\text{Figure 12. Qtac syntax.}

\begin{align*}
\text{Intro} & : \quad \Gamma \vdash n : T \Rightarrow b \Rightarrow p \\
\text{Symmetry} & : \quad \Gamma \vdash H \Rightarrow p \\
\text{Split} & : \quad \Gamma \vdash l \Rightarrow p \quad \Gamma \vdash r \Rightarrow q \\
\text{Left} & : \quad \Gamma \vdash \text{Constr}(0, \lor) H \Rightarrow \text{left} \langle p \rangle \\
\text{Right} & : \quad \Gamma \vdash \text{Constr}(1, \lor) H \Rightarrow \text{right} \langle p \rangle \\
\text{Rewrite} & : \quad \Gamma \vdash \text{Elim}(H_1, P(x, H_2, y)) \Rightarrow \text{symmetry} \Rightarrow \text{rewrite} \Rightarrow P H_1, p \\
\text{Induction} & : \quad \Gamma \vdash \text{Elim}(t, P) \Rightarrow \text{induction} \Rightarrow P t, p \\
\text{Apply} & : \quad \Gamma \vdash ft \Rightarrow \text{apply} \Rightarrow f \Rightarrow p \\
\text{Base} & : \quad \Gamma \vdash t \Rightarrow \text{apply} \Rightarrow t
\end{align*}

\text{Figure 13. Qtac decompiler semantics.}

5 Decompiling Proof Terms to Tactics

\textbf{Transform} produces a proof term, while the proof engineer typically writes and maintains proof scripts made up of tactics. We improve usability thanks to the realization that, since Coq’s proof term language Gallina is very structured, we can decompile these Gallina terms to suggested Ltac proof scripts for the proof engineer to maintain.

\textbf{Decompile} implements a prototype of this translation (11): it translates a proof term to a suggested proof script that attempts to prove the same theorem the same way. Note that this problem is not well defined: while there is always a proof script that works (applying the proof term with the \textit{apply} tactic), the result is often qualitatively unreadable. This is the baseline behavior to which the decompiler defaults. The goal of the decompiler is to improve on that baseline as much as possible, or else suggest a proof script that is close enough to correct that the proof engineer can manually massage it into something that works and is maintainable.

\textbf{Decompile} achieves this in two passes: The first pass decompiles proof terms to proof scripts that use a predefined set of tactics. The second pass improves on suggested tactics by simplifying arguments, substituting tactics, and using hints like custom tactics and decision procedures.

\textbf{First Pass: Basic Proof Scripts}. The first pass takes Coq terms and produces tactics in Ltac, the proof script language for Coq. Ltac can be confusing to reason about, since Ltac tactics can refer to Gallina terms, and the semantics of Ltac depends both on the semantics of Gallina and on the implementation of proof search procedures written in OCaml. To give a sense of how the first pass works without the clutter of these details, we start by defining a mini decompiler that implements a simplified version of the first pass. Section 6.2 explains how we scale this to the implementation.

The mini decompiler takes CIC\(_{\omega}\) terms and produces tactics in a mini version of Ltac which we call Qtac. The syntax for Qtac is in Figure 12. Qtac includes hypothesis introduction (\textit{intro}), rewriting (\textit{rewrite}), symmetry of equality (\textit{symmetry}), application of a term to prove the goal (\textit{apply}), induction (\textit{induction}), case splitting of conjunctions (\textit{split}), constructors of disjunctions (\textit{left} and \textit{right}), and composition (\textit{.}). Unlike in Ltac, \textit{induction} and \textit{rewrite} take a motive explicitly (rather than relying on unification), and \textit{apply} creates a new subgoal for each function argument.

The semantics for the mini decompiler \(\Gamma \vdash t \Rightarrow p\) are in Figure 13 (assuming =, \textit{eq_sym}, \land, and \lor are defined as in Coq). As with the real decompiler, the mini decompiler defaults to the proof script that applies the entire proof term with \textit{apply} (\textit{Base}). Otherwise, it improves on that behavior by recursing over the proof term and constructing a proof script using a predefined set of tactics.

For the mini decompiler, this is straightforward: Lambda terms become introduction (\textit{intro}). Applications of \textit{eq_sym} become symmetry of equality (\textit{symmetry}). Constructors of conjunction and disjunction map to the respective tactics (\textit{Split}, \textit{Left}, and \textit{Right}). Applications of equality eliminators compose symmetry (to orient the rewrite direction) with rewrites (\textit{rewrite}), and all other applications of eliminators become induction (\textit{induction}). The remaining applications become apply tactics (\textit{apply}). In all cases, the decompiler recurses, breaking into cases, until only the \textit{Base} case holds.

While the mini decompiler is very simple, only a few small changes are needed to move this to Coq. The generated proof term of \textit{rev_app_distr} from Section 2, for example, consists only of induction, rewriting, simplification, and reflexivity (solved by \textit{auto}). Figure 14 shows the proof term for the base case of \textit{rev_app_distr} alongside the proof script that \textit{Pumpkin Pi} suggests. This script is fairly low-level and close to
the proof term, but it is already something that the proof engineer can step through to understand, modify, and maintain. There are few differences from the mini decompiler needed to produce this, for example handling of rewrites in both directions (eq_ind_r as opposed to eq_ind), simplifying rewrites, and turning applications of eq_refl into reflexivity or auto.

Second Pass: Better Proof Scripts. The implementation of Decompile first runs something similar to the mini decompiler, then modifies the suggested tactics to produce a more natural proof script \([11]\). For example, it cancels out sequences of intros and revert, inserts semicolons, and removes extra arguments to apply and rewrite. It can also take tactics from the proof engineer (like part of the old proof script) as hints, then iteratively replace tactics with those hints, checking for correctness. This makes it possible for suggested scripts to include custom tactics and decision procedures.

6 Implementation

The transformation and mini decompiler abstract many of the challenges of building a tool for proof engineers. This section describes how we solved some of these challenges.

6.1 Implementing the Transformation

Termination. When a subterm unifies with a configuration term, this suggests that PUMPKIN Pi can transform the subterm, but it does not necessarily mean that it should. In some cases, doing so would result in nontermination. For example, if \(B\) is a refinement of \(A\), then we can always run EQUIVALENCE over and over again, forever. We thus include some simple termination checks in our code \([12]\).

Intent. Even when termination is guaranteed, whether to transform a subterm depends on intent. That is, PUMPKIN Pi automates the case of porting every \(A\) to \(B\), but proof engineers sometimes wish to port only some \(A\)s to \(B\)s. PUMPKIN Pi has some support for this using an interactive workflow \([13]\), with plans for automatic support in the future.

From CIC\(_\omega\) to Coq. The implementation \([4]\) of the transformation handles language differences to scale from CIC\(_\omega\) to Coq. We use the existing Preprocess \([33]\) command to turn pattern matching and fixpoints into eliminators. We handle refolding of constants in constructors using DepConstr.

Reaching Real Proof Engineers. Many of our design decisions in implementing PUMPKIN Pi were informed by our partnership with an industrial proof engineer (Section 7). For example, the proof engineer rarely had the patience to wait more than ten seconds for PUMPKIN Pi to port a term, so we implemented optional aggressive caching, even caching intermediate subterms encountered while running the transformation \([14]\). We also added a cache to tell PUMPKIN Pi not to \(\delta\)-reduce certain terms \([14]\). With these caches, the proof engineer found PUMPKIN Pi efficient enough to use on a code base with tens of thousands of lines of code and proof.

The experiences of proof engineers also inspired new features. For example, we implemented a search procedure to generate custom eliminators to help reason about types like \(\Sigma(1 : \text{list } T).\text{length } l = n\) by reasoning separately about the projections \([15]\). We added informative error messages \([22]\) to help the proof engineer distinguish between user errors and bugs. These features helped with workflow integration.

6.2 Implementing the Decompiler

From Qtac to Ltac. The mini decompiler assumes more predictable versions of rewrite and induction than those in Coq. Decompile includes additional logic to reason about these tactics \([11]\). For example, Qtac assumes that there is only one rewrite direction. Ltac has two rewrite directions, and so the decompiler infers the direction from the motive. Qtac also assumes that both tactics take the inductive motive explicitly, while in Coq, both tactics infer the motive automatically. Consequently, Coq sometimes fails to infer the correct motive. To handle induction, the decompiler strategically uses revert to manipulate the goal so that Coq can better infer the motive. To handle rewrites, it uses simpl to refold the goal before rewriting. Neither of these approaches is guaranteed to work, so the proof engineer may sometimes need to tweak the suggested proof script appropriately. Even if we pass Coq’s induction principle an explicit motive, Coq still sometimes fails due to unrepresented assumptions. Long term, using another tactic like change or refine before applying these tactics may help with cases for which Coq cannot infer the correct motive.

From CIC\(_\omega\) to Coq. Scaling the decompiler to Coq introduces let bindings, which are generated by tactics like rewrite in, apply in, and pose. Decompile implements \([11]\) support for rewrite in and apply in similarly to how it supports rewrite and apply, except that it ensures that the unmanipulated hypothesis does not occur in the body of the let expression, it swaps the direction of the rewrite, and it recurses into any generated subgoals. In all other cases, it uses pose, a catch-all for let bindings.

```coq
fun (y0 : list A) =>
list_rect y0 (fun a l H =>
eq_ind_r y0 (eq_refl (app_nil_r (rev l) (a::[l])))
eq_refl)
y0

-intro y0. induction y0 as [a l H].
+ simpl. rewrite app_nil_r. auto.
+ auto.
```

Figure 14. Proof term (top) and decompiled proof script (bottom) for the base case of rev_app_distr (Section 2), with corresponding terms and tactics grouped by color & number.
Table 1. Some changes using PUMPKIN Pi (left to right): class of changes, kind of configuration, examples, whether using PUMPKIN Pi saved development time relative to reference manual repairs (☐ if yes, ☑ if comparable, ☐ if no), and Coq tools we know of that support repair along (Repair) or automatic proof of (Search) the equivalence corresponding to each example. Tools considered are DEVOID [33], the Univalent Parametricity (UP) white-box transformation [39], and the tool from Magaud and Bertot [22]. PUMPKIN Pi is the only one that suggests tactics. More nuanced comparisons to these and more are in Section 8.

<table>
<thead>
<tr>
<th>Class</th>
<th>Config.</th>
<th>Examples</th>
<th>Sav.</th>
<th>Repair Tools</th>
<th>Search Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic Ornaments</td>
<td>Auto</td>
<td>List to Packed Vector, has-to-coq ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, DEVOID, UP</td>
<td>PUMPKIN Pi, DEVOID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>List to Packed Vector, Std. Library ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, DEVOID, UP</td>
<td>PUMPKIN Pi, DEVOID</td>
</tr>
<tr>
<td>Unpack Sigma Types</td>
<td>Auto</td>
<td>Vector of Given Length, has-to-coq ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td>Tuples &amp; Records</td>
<td>Auto</td>
<td>Simple Records ⚤</td>
<td>☑</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameterized Records ☐</td>
<td>☐</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Industrial Use ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td>Permute Constructors</td>
<td>Auto</td>
<td>List, Standard Library ☰</td>
<td>☰</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modifying PL, REPLICA Benchmark ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Large Ambiguous Enum ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi</td>
</tr>
<tr>
<td>Add new Constructors</td>
<td>Mixed</td>
<td>PL Extension, REPLICA Benchmark ☐</td>
<td>☰</td>
<td>PUMPKIN Pi, UP</td>
<td>PUMPKIN Pi (partial)</td>
</tr>
<tr>
<td>Factor Constructors</td>
<td>Manual</td>
<td>External Example ☰</td>
<td>☰</td>
<td>PUMPKIN Pi, UP</td>
<td>None</td>
</tr>
<tr>
<td>Permute Hypotheses</td>
<td>Manual</td>
<td>External Example ☰</td>
<td>☰</td>
<td>PUMPKIN Pi, UP</td>
<td>None</td>
</tr>
<tr>
<td>Change Ind. Structure</td>
<td>Manual</td>
<td>Unary to Binary, Classic Benchmark ☑</td>
<td>☑</td>
<td>PUMPKIN Pi, Magaud</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector to Fin, S', External Example ☰</td>
<td>☰</td>
<td>PUMPKIN Pi</td>
<td>None</td>
</tr>
</tbody>
</table>

Forfeiting Soundness. While there is a way to always produce a correct proof script, Decompile deliberately forfeits soundness to suggest more useful tactics. For example, it may suggest the induction tactic, but leave the step of motive inference to the proof engineer. We have found these suggested tactics easier to work with (Section 7). Note that in the case the suggested proof script is not quite correct, it is still possible to use the generated proof term directly.

Pretty Printing. After decomposing proof terms, Decompile pretty prints the result (11). Like the mini decompiler, Decompile represents its output using a predefined grammar of Ltac tactics, albeit one that is larger than Qtac, and that also includes tactics. It maintains the recursive proof structure for formatting. PUMPKIN Pi keeps all output terms from Transform in the Coq environment in case the decomplier does not succeed. Once the proof engine has the new proof, she can remove the old one.

7 Case Studies: PUMPKIN Pi Eight Ways

This section summarizes eight case studies using PUMPKIN Pi, corresponding to the eight rows in Table 1. These case studies highlight PUMPKIN Pi’s flexibility in handling diverse scenarios, the success of automatic configuration for better workflow integration, the preliminary success of the prototype decomplier, and clear paths to better serving proof engineers. Detailed walkthroughs are in the code.

Algebraic Ornaments: Lists to Packed Vectors. The transformation in PUMPKIN Pi is a generalization of the transformation from DEVOID. DEVOID supported proof reuse across algebraic ornaments, which describe relations between two inductive types, where one type is the other indexed by a fold [23]. A standard example is the relation between a list and a length-indexed vector (Figure 5).

PUMPKIN Pi implements a search procedure for automatic configuration of algebraic ornaments. The result is all functionality from DEVOID, plus tactic suggestions. In file (3), we used this to port functions and a proof from lists to vectors of some length, since list T ≃ packed_vec T. The decomplier helped us write proofs in the order of hours that we had found too hard to write by hand, though the suggested tactics did need massaging.

Unpack Sigma Types: Vectors of Particular Lengths. In the same file (3), we then ported functions and proofs to vectors of a particular length, like vector T n. DEVOID had left this step to the proof engineer. We supported this in PUMPKIN Pi by chaining the previous change with an automatic configuration for unpacking sigma types. By composition, this transported proofs across the equivalence from Section 3.

Two tricks helped with better workflow integration for this change: 1) have the search procedure view vector T n as Σ(v : vector T m).n = m for some m, then let PUMPKIN Pi instantiate those equalities via unification heuristics, and 2) generate a custom eliminator for combining list terms with length invariants. The resulting workflow works not just for lists and vectors, but for any algebraic ornament, automating manual effort from DEVOID. The suggested tactics were helpful for writing proofs in the order of hours that we had struggled with manually over the course of days, but only after massaging. More effort is needed to improve tactic suggestions for dependent types.

Tuples & Records: Industrial Use. An industrial proof engineer at the company Galois has been using PUMPKIN Pi in proving correct an implementation of the TLS handshake.
protocol. Galois had been using a custom solver-aided verification language to prove correct C programs, but had found that at times, the constraint solvers got stuck. They had built a compiler that translates their language into Coq’s specification language Gallina, that way proof engineers could finish stuck proofs interactively using Coq. However, due to language differences, they had found the generated Gallina programs and specifications difficult to work with.

The proof engineer used Pumpkin Pi to port the automatically generated functions and specifications to more human-readable functions and specifications, wrote Coq proofs about those functions and specifications, then used Pumpkin Pi to port those proofs back to proofs about the original functions and specifications. So far, they have used at least three automatic configurations, but they most often used an automatic configuration for porting compiler-produced anonymous tuples to named records, as in file `18`. The workflow was a bit nonstandard, so there was little need for tactic suggestions. The proof engineer reported an initial time investment learning how to use Pumpkin Pi, followed by later returns.

**Permute Constructors: Modifying a Language.** The swapping example from Section 2 was inspired by benchmarks from the Replica user study of proof engineers [31]. A change from one of the benchmarks is in Figure 15. The proof engineer had a simple language represented by an inductive type `Term`, as well as some definitions and proofs about the language. The proof engineer swapped two constructors in the language, and added a new constructor `Bool`.

This case study and the next case study break this change into two parts. In the first part, we used Pumpkin Pi with automatic configuration to repair functions and proofs about the language after swapping the constructors [1]. With a bit of human guidance to choose the permutation from a list of suggestions, Pumpkin Pi repaired everything, though the original tactics would have also worked, so there was not a difference in development time.

**Add new Constructors: Extending a Language.** We then used Pumpkin Pi to repair functions after adding the new constructor in Figure 15, separating out the proof obligations for the new constructor from the old terms [19]. This change combined manual and automatic configuration. We defined an inductive type `Diff` and (using partial automation) a configuration to port the terms across the equivalence `Old`. `Term + Diff = New.Term`. This resulted in case explosion, but was formulaic, and pointed to a clear path for automation of this class of changes. The repaired functions guaranteed preservation of the behavior of the original functions.

Adding constructors was less simple than swapping. For example, Pumpkin Pi did not yet save us time over the proof engineer from the user study; fully automating the configuration would have helped significantly. In addition, the repaired terms were (unlike in the swap case) inefficient compared to human-written terms. For now, they make good regression tests for the human-written terms—in the future, we hope to automate the discovery of the more efficient terms, or use the refinement framework CoqEAL [8] to get between proofs of the inefficient and efficient terms.

**Factor out Constructors: External Example.** The change from Figure 4 came at the request of a non-author. We supported this using a manual configuration that described which constructor to map to `true` and which constructor to map to `false` [2]. The configuration was very simple for us to write, and the repaired tactics were immediately useful. The development time savings were on the order of minutes for a small proof development. Since most of the modest development time went into writing the configuration, we expect time savings would increase for a larger development.

**Permute Hypotheses: External Example.** The change in [20] came at the request of a different non-author (a cubical type theory expert), and shows how to use Pumpkin Pi to swap two hypotheses of a type, since `T1 → T2 → T3 ≃ T2 → T1 → T3`. This configuration was manual. Since neither type was inductive, this change used the generic construction for any equivalence. This worked well, but necessitated some manual annotation due to the lack of custom unification heuristics for manual configuration, and so did not yet save development time, and likely still would not have had the proof development been larger. Supporting custom unification heuristics would improve this workflow.

**Change Inductive Structure: Unary to Binary.** In [5], we used Pumpkin Pi to support a classic example of changing inductive structure: updating unary to binary numbers, as in

```
| Inductive Term : Set :=
| Var : Identifier → Term
| Int : Z → Term
| Eq : Term → Term → Term
| Plus : Term → Term → Term
| Times : Term → Term → Term
| Minus : Term → Term → Term
| Choose : Identifier → Term → Term.
```

**Figure 15.** A simple language (left) and the same language with two swapped constructors and an added constructor (right).
Figure 8. Binary numbers allow for a fast addition function, found in the Coq standard library. In the style of Magaud and Bertot [22], we used Pumpkin Pi to derive a slow binary addition function that does not refer to nat, and to port proofs from unary to slow binary addition. We then showed that the ported theorems hold over fast binary addition.

The configuration for N used definitions from the Coq standard library for DepConstr and DepElim that had the desired behavior with no changes. Iota over the successor case was a rewrite by a lemma from the standard library that reduced the successor case of the eliminator that we used for DepElim:

\[
\begin{align*}
P.n.peano_rect_succ & : \forall P \ p0 \ pS \ n. \\
P.n.peano_rect & : \forall P \ p0 \ pS \ (N.\text{succ} \ n) = \\
& \ pS \ n \ (N.n.peano_rect \ P \ p0 \ pS \ n).
\end{align*}
\]

The need for nontrivial Iota comes from the fact that N and nat have different inductive structures. By writing a manual configuration with this Iota, it was possible for us to implement this transformation that had been its own tool.

While porting addition from nat to N was automatic after configuring Pumpkin Pi, porting proofs about addition took more work. Due to the lack of unification heuristics for manual configuration, we had to annotate the proof term to tell Pumpkin Pi that implicit casts in the inductive cases of proofs were applications of Iota over nat. These annotations were formulaic, but tricky to write. Unification heuristics would go a long way toward improving the workflow.

After annotating, we obtained automatically repaired proofs about slow binary addition, which we found simple to port to fast binary addition. We hope to automate this last step in the future using CoqEAL. Repaired tactics were partially useful, but failed to understand custom eliminators like N.peano_rect, and to generate useful tactics for applications of Iota; both of these are clear paths to more useful tactics. The development time for this proof with Pumpkin Pi was comparable to reference manual repairs by external proof engineers. Custom unification heuristics would help bring returns on investment for experts in this use case.

8 Related Work

**Proof Repair.** The search procedures in Configure are based partly on ideas from the original Pumpkin Patch prototype [32]. The Pumpkin Patch prototype did not apply the patches that it finds, handle changes in structure, or include support for tactics beyond the use of hints.

Proof repair can be viewed as a form of program repair [15, 24] for proof assistants. Proof assistants like Coq are a good fit for program repair: A recent paper [29] recommends that program repair tools draw on extra information such as specifications or example patches. In Coq, specifications and examples are rich and widely available: specifications thanks to dependent types, and examples thanks to constructivism.

**Proof Refactoring.** The proof refactoring tool Levity [4] for Isabelle/HOL has seen large-scale industrial use. Levity focuses on a different task: moving lemmas. Chick [34] and RefactorAgda [44] are proof refactoring tools in a Gallina-like language and in Agda, respectively. These tools support primarily syntactic changes and do not have tactic support.

A few proof refactoring tools operate directly over tactics: POLAR [13] refactors proof scripts in languages based on Isabelle/Isar [42], CoqPIE [35] is an IDE with support for simple refactorings of Ltac scripts, and Tactician [1] is a refactoring tool for switching between tactics and tacticals. This approach is not tractable for more complex changes [34].

**Proof Reuse.** A few proof reuse tools work by proof term transformation and so can be used for repair. Johnsen and Lüth [18] describes a transformation that generalizes theorems in Isabelle/HOL. Pumpkin Pi generalizes the transformation from DEVOID [33], which transformed proofs along algebraic ornaments [23]. Magaud and Bertot [22] implement a proof term transformation between unary and binary numbers. Both of these fit into Pumpkin Pi configurations, and none suggests tactics in Coq like Pumpkin Pi does. The expansion algorithm from Magaud and Bertot [22] may help guide the design of unification heuristics in Pumpkin Pi.

The widely used Transfer [17] package supports proof reuse in Isabelle/HOL. Transfer works by combining a set of extensible transfer rules with a type inference algorithm. Transfer is not yet suitable for repair, as it necessitates maintaining references to both datatypes. One possible path toward implementing proof repair in Isabelle/HOL may be to reify proof terms using something like Isabelle/HOL-Proofs, apply a transformation based on Transfer, and then (as in Pumpkin Pi) decompile those terms to automation that does not apply Transfer or refer to the old datatype in any way.

CoqEAL [8] transforms functions across relations in Coq, and these relations can be more general than Pumpkin Pi’s equivalences. However, while Pumpkin Pi supports both functions and proofs, CoqEAL supports only simple functions due to the problem that Iota addresses. CoqEAL may be most useful to chain with Pumpkin Pi to get faster functions. Both CoqEAL and recent ornaments work [45] may help with better workflow support for changes that do not correspond to equivalences.

The Pumpkin Pi transformation implements transport. Transport is realizable as a function given univalence [41]. UP [38] approximates it in Coq, only sometimes relying on functional extensionality. While powerful, neither approach removes references to the old type.

Recent work [39] extends UP with a white-box transformation that may work for repair. This imposes proof obligations on the proof engineer beyond those imposed by Pumpkin Pi, and it includes neither search procedures for equivalences nor tactic script generation. It also does not support changes in inductive structure, instead relying on its original blackbox functionality: Iota solves this in Pumpkin Pi. The most fruitful progress may come from combining these tools.
Proof Design. Much work focuses on designing proofs to be robust to change, rather than fixing broken proofs. This can take the form of design principles, like using information hiding techniques [20, 46] or any of the structures [7, 36, 37] for encoding interfaces in Coq. Design and repair are complementary: design requires foresight, while repair can occur retroactively. Repair can help with changes that occur outside of the proof engineer’s control, or with changes that are difficult to protect against even with informed design.

Another approach to this is to use heavy proof automation, for example through program-specific proof automation [6] or general-purpose hammers [3, 11, 19, 28]. The degree to which proof engineers rely on automation varies, as seen in the data from a user study [31]. Automation-heavy proof engineering styles localize the burden of change to the automation, but can result in terms that are large and slow to type check, and tactics that can be difficult to debug. While these approaches are complementary, more work is needed for Pumpkin Pi to better support developments in this style.

9 Conclusions & Future Work
We combined search procedures for equivalences, a proof term transformation, and a proof term to tactic decompiler to build Pumpkin Pi, a proof repair tool for changes in datatypes. The proof term transformation implements transport across equivalences in a way that is suitable for repair and that does not compromise definitional equality. The resulting tool is flexible and useful for real proof engineering scenarios.

Future Work. Moving forward, our goal is to make proofs easier to repair and reuse regardless of proof engineering expertise and style. We want to reach more proof engineers, and we want Pumpkin Pi to integrate seamlessly with Coq.

Three problems that we encountered scaling up the Pumpkin Pi transformation were lack of type-directed search, ad hoc termination checks, and inability for proof engineers to add custom unification heuristics. We hope to solve these challenges using e-graphs [25], a data structure for managing equivalences built with these kinds of problems in mind. E-graphs were recently adapted to express path equality in cubical [16]; we hope to repurpose this insight.

Beyond that, we believe that the biggest gains will come from continuing to improve the prototype decompiler. Two helpful features would be preserving indentation and comments, and automatically using information from the original proof script rather than asking for hints. One promising path toward the latter is to integrate the decompiler with a machine learning tool like TacTok [14] to rank tactic hints. Some improvements could also come from better tactics, or from a more structured tactic language. Integration with version control systems or with integrated development environments could also help. With that, we believe that the future of seamless and powerful proof repair and reuse for all is within reach.

Acknowledgments
This paper has really been a community effort. Anders Mörtberg and Conor McBride gave us hours worth of detailed feedback that was instrumental to writing this paper. Nicolas Tabareau helped us understand the need to port definitional to propositional equalities. Valentin Robert gave us feedback on usability that informed tool design. Gaëtan Gilbert, James Wilcox, and Jasper Hugunin all at some point helped us write Coq proofs; let this be a record that we owe Gaëtan a beer, and we owe James boba.

And of course, we thank our shepherd Gerwin Klein, and we thank all of our reviewers. We got other wonderful feedback on the paper from Cyril Cohen, Tej Chajed, Ben Delaware, Jacob Van Geffen, Janno, James Wilcox, Chandrakana Nandi, Martin Kellogg, Audrey Seo, James Decker, Ben Kashigian, John Regehr, and Justus Adam. The Coq developers have for years given us frequent and efficient feedback on plugin APIs for tool implementation. The programming languages community on Twitter (yes, seriously) has also been essential to this effort. Especially during a pandemic. And we’d like to extend a special thank you to Talia’s mentor, Derek Dreyer.

We have also gotten a lot of feedback on future ideas that we are excited to pursue. Carlo Angiuli has helped us understand some beautiful theory beneath our implementation (spoiler: we believe the analogy connecting DepConstr and DepElim to constructors and eliminators has formal meaning in terms of initial algebras), and we are excited to integrate these insights into future papers and use them to generalize our insights. Alex Polozov helped us sketch out ideas for future work with the decompiler. And we got wonderful feedback on e-graph integration for future work from Max Willsey, Chandrakana Nandi, Remy Wang, Zach Tatlock, Bas Spitters, Steven Lyubomirsky, Andrew Liu, Mike He, Ben Kashigian, Gus Smith, and Bill Zorn.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References


[40] Amin Timany and Bart Jacobs. 2015. First Steps Towards Cumulative Inductive Types in CIC. In ICTAC.


