Ornaments for Proof Reuse in Coq

Talia Ringer
University of Washington, USA
tringer@cs.washington.edu

Nathaniel Yazdani
University of Washington, USA
nyazdani@cs.washington.edu

John Leo
Halfaya Research, USA
leo@halfaya.org

Dan Grossman
University of Washington, USA
djg@cs.washington.edu

Abstract

Ornaments express relations between inductive types with the same inductive structure. We implement fully automatic proof reuse for a particular class of ornaments in a Coq plugin, and show how such a tool can give programmers the rewards of using indexed inductive types while automating away many of the costs. The plugin works directly on Coq code; it is the first ornamentation tool for a non-embedded dependently typed language. It is also the first tool to automatically identify ornaments: To lift a function or proof, the user must provide only the source type, the destination type, and the source function or proof. In taking advantage of the mathematical properties of ornaments, our approach produces faster functions and smaller terms than a more general approach to proof reuse in Coq.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases ornaments, proof reuse, proof automation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.26

Supplement Material The Coq plugin, examples, and case study code for this paper can be found at `http://github.com/uwplse/ornamental-search/tree/itp+equiv`.

Acknowledgements We thank Jasper Hugunin, James Wilcox, Jason Gross, Pavel Panchekha, and Marisa Kirisame for ideas that helped inform tool design. We thank Thomas Williams, Josh Ko, Matthieu Sozeau, Cyril Cohen, Nicolas Tabareau, and Enrico Tassi for help navigating related work. We thank Emilio J. Gallego Arias, Gaëtan Gilbert, Pierre-Marie Pédrot, and Yves Bertot for help understanding Coq plugin APIs. We thank Shachar Itzhaky and Tej Chajed for ideas for future directions. We thank the UW and UCSD programming languages labs for feedback. 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.

1 Introduction

Indexed inductive types make it possible to internalize data into the type level, eliminating the need for certain functions and proofs. Consider, for example, a theorem from the Coq standard library [17] which states that mapping a function over lists preserves length:

```
map_length T1 T2 (f : T1 → T2) : ∀ (l : list T), length (List.map f l) = length l.
```

```
list (T : Type) : Type :=
| nil : list T
| cons :
    T → list T → list T.
```

```
vector (T : Type) : nat → Type :=
| nilV : vector T 0
| consV :
    ∀ (n :  nat), T → vector T n → vector T (S n).
```

Figure 1 A `vector` (right) is a `list` (left) indexed by its length (highlighted in orange).

One way to eliminate the need for this theorem is to internalize the length of a list into its type, creating a dependently typed vector (Figure 1). The map function for vectors in Coq's standard library, for example, carries a proof that it preserves length:

```
Vector.map {T1} {T2} (f : T1 → T2) : ∀ (n : nat) (v : vector T1 n), vector T2 n.
```

so that a theorem like `map_length` is no longer necessary.

Unfortunately, for all of the benefits they bring, indexed inductive types are notoriously difficult to use. Dependently typed vectors, for example, impose proof obligations about their lengths on the user; these can quickly spiral out of control. In recent coq-club threads asking for advice on how to use dependently typed vectors, experts called them "not suitable for extended use" [7] and noted that "almost no one should be using [them] for anything" [8].

We show how *proof reuse* – reusing existing proofs to derive new proofs – can tackle many of the challenges posed by indexed inductive types, allowing the user to move between unindexed and indexed versions of a type (for example, lists and vectors) and reap the benefits of indexed types without many of the costs. We focus in particular on the benefits of this approach in deriving functions and proofs for fully-determined indexed types, when the index is a fold over the unindexed version (such as the length of a list). In our approach, the user writes functions and proofs over the unindexed version, and a tool then automatically *lifts* those functions and proofs to the indexed version. The user can then switch back to working with the unindexed version by running the tool in the opposite direction. In that way, the user can use lists when lists are convenient, and vectors when vectors are convenient.

Our approach uses *ornaments* [23], which express relations between types that preserve inductive structure, and which enable lifting of functions and proofs along those relations. Recent work introduced ornaments to a subset of ML and was heavily focused on automatically lifting functions [33]; until now, such an approach was not available in a dependently typed language. Existing implementations of ornaments in dependently typed languages work only in embedded languages, and have little to no automation [20, 23, 11].

Our main contribution is a Coq plugin for automatic function and proof reuse using ornaments. Our plugin Devoid (Dependent Equivalences Via Ornamenting Inductive Definitions) works directly on Coq code, rather than on an embedded language. Devoid automates lifting functions and proofs along *algebraic ornaments* [23], a particular class of ornaments that represent fully-determined indexed types like lists and vectors. Devoid implements an algorithm to search for ornaments between these types – to the best of our knowledge, the first search algorithm for ornaments – and an algorithm to lift functions and proofs along the ornaments it discovers.

We motivate (Section 2), specify (Section 3), and formalize (Section 4) the search and lifting algorithms that Devoid implements (Section 5). A comparison to a more general proof reuse approach (Section 6) demonstrates the benefits of using ornaments: Devoid imposes less of a proof burden on the user, and produces smaller terms and faster functions.

2 Motivating Example: Porting a Library

Devoid is a plugin for Coq 8.8; it can be found in the repository linked to as **Supplement Material** under the abstract of this paper. To see how it works, consider an example using the types from Figure 1, the code for which is in `Example.v`. In this example, we lift two list zip functions and a proof of a theorem relating them from the Haskell CoreSpec library [29]:

```
zip {T₁ T₂}: list T₁ → list T₂ → list (T₁ * T₂).
zip_with {T₁ T₂ T₃} (f : T₁ → T₂ → T₃): list T₁ → list T₂ → list T₃.
zip_with_is_zip {T₁ T₂}: ∀(l₁:list T₁)(l₂:list T₂), zip_with pair l₁ l₂ = zip l₁ l₂.
```

Devoid runs a preprocessing step before lifting, which we describe in Section 5; we assume this step has already run. We use the cyan background color to denote tool-produced terms and the names that refer to them. We run Devoid to lift functions and proofs from lists to vectors, but it can also lift in the opposite direction.

Step 1: Search. We first use Devoid's `Find ornament` command to search for the relation between lists and vectors:

```
Find ornament list vector.
```

This produces functions which together form an equivalence (denoted $\simeq$):

```
list T ≃ Σ (n : nat).vector T n
```

Step 2: Lift. We then lift our functions and proofs along that equivalence using Devoid's `Lift` command. For example, to lift `zip`, we run the command:

```
Lift list vector in zip as zipV_p.
```

This produces a function with this type:

```
zipV_p {T₁ T₂} : Σ n.vector T₁ n → Σ n.vector T₂ n → Σ n.vector (T₁ * T₂) n.
```

that behaves like `zip`, but whose body no longer refers to lists. We lift our proof similarly:

```
Lift list vector in zip_with_is_zip as zip_with_is_zipV_p.
```

This produces a proof of the analogous result (denoting projections by π_l and π_r):

```
zip_with_is_zipV_p {T₁ T₂} : ∀ (v₁ : Σ n.vector T₁ n) (v₂ : Σ n.vector T₂ n),
  zip_withV_p pair (∃ (πl v₁) (πr v₁)) (∃ (πl v₂) (πr v₂)) =
  zipV_p (∃ (πl v₁) (πr v₁)) (∃ (πl v₂) (πr v₂)).
```

that no longer refers to lists, `zip`, or `zip_with` in any way.

Step 3: Unpack. The lifted terms operate over vectors whose lengths are *packed* inside of a sigma type. While this lets `Lift` provide strong theoretical guarantees, it can make it difficult to interface with the lifted code. We can recover *unpacked* terms using Devoid's `Unpack` command. For example, to unpack `zipV_p`, we run the command:

```
Unpack zipV_p as zipV.
```

This produces functions and proofs that operate directly over vectors, like `zipV`:

```
zipV {T₁ T₂} {n₁} (v₁ : vector T₁ n₁) {n₂} (v₂ : vector T₂ n₂) :
  vector (T₁ * T₂) (πl (zipV_p (∃ n₁ v₁) (∃ n₂ v₂))).
```

and `zip_with_is_zipV`:

```
zip_with_is_zipV : ∀ {T1 T2} {n1} (v1 : vector T1 n1) {n2} (v2 : vector T2 n2),
  eq_dep _ _ _ (zip_withV pair v1 v2) _ (zipV v1 v2).
```

Step 4: Interface. For any two inputs of the same length, `zipV` and `zipV_with` contain proofs that the output has the same length as the inputs. However, the types obscure this information. `Example.v` explains how to recover more user-friendly types, like that of `zipV_uf`:

```
zipV_uf {T1 T2} {n} : vector T1 n → vector T2 n → vector (T1 * T2) n.
```

and that of `zip_withV_uf`:

```
zip_withV_uf {T1 T2 T3} (f : T1 → T2 → T3) {n} :
  vector T1 n → vector T2 n → vector T3 n.
```

which both restrict input lengths. We can then use our lifted functions and proofs in client code. For example, we can write a different version of Coq's `BVand` function for bitvectors:

```
BVand {n} (v1 : vector bool n) (v2 : vector bool n) : vector bool n :=
  zip_withV_uf andb v1 v2.
```

By working over lists, we are able to reason about only the interesting pieces, thinking about indices only when relevant; in contrast, when writing proofs over vectors, even simple theorems can generate tricky proof obligations. With Devoid, the programmer can use the lifted functions and proofs to interface with code that uses vectors, then switch back to lists when vectors are unmanageable. In essence, ornaments form the glue between these types.

3 Specification

This section specifies the two commands that Devoid implements:

1. `Find ornament` searches for ornaments (specified in Section 3.1, described in Section 4.1).
2. `Lift` lifts along those ornaments (specified in Section 3.2, described in Section 4.2).

Algebraic Ornaments. Devoid searches for and lifts along *algebraic ornaments* in particular. An algebraic ornament relates an inductive type A to an indexed version of that type B with a new index of type I_B, where the new index is fully determined by a unique fold over A. For example, `vector` is exactly `list` with a new index of type `nat`, where the new index is fully determined by the `length` function. Consequentially, there are two functions:

```
ltv : list T                → Σ(n : nat).vector T n.
vtl : Σ(n : nat).vector T n → list T.
```

that are mutual inverses:

```
∀ (l : list T),                vtl (ltv l) = l.
∀ (v : Σ(n : nat).vector T n), ltv (vtl v) = v.
```

and therefore form the type equivalence from Section 2. Moreover, since the new index is fully determined by `length`, we can relate `length` to `ltv`:

```
∀ (l : list T), length l = π1 (ltv l).
```

In general, we can view an algebraic ornament as a type equivalence:

$$A\ \vec{i}\ \simeq\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})$$

where $\vec{i}$ are the indices of A, I_B is a function over those indices, and the `index` operation inserts the new index n at the right offset. Such a type equivalence consists of two functions [32]:

$$\begin{array}{lll}\texttt{promote} & : A\ \vec{i} & \rightarrow\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}).\\ \texttt{forget} & : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}) & \rightarrow\ A\ \vec{i}.\end{array}$$

that are mutual inverses:[1]

$$\begin{array}{lll}\texttt{section} & : \forall\ (a : A\ \vec{i}), & \texttt{forget}\ (\texttt{promote}\ a)\ = a.\\ \texttt{retraction} & : \forall\ (b_\Sigma : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})), & \texttt{promote}\ (\texttt{forget}\ b_\Sigma) = b_\Sigma.\end{array}$$

An algebraic ornament is additionally equipped with an indexer, which is a unique fold:

$$\texttt{indexer} : A\ \vec{i}\ \rightarrow\ I_B\ \vec{i}.$$

which projects the promoted index:

$$\texttt{coherence} : \forall (a : A\ \vec{i}),\ \texttt{indexer}\ a = \pi_l\ (\texttt{promote}\ a).$$

Following existing work [20], we call this equivalence the *ornamental promotion isomorphism*; when it holds and the indexer exists, we say that B is an algebraic ornament of A.

`Find ornament` searches for algebraic ornaments between types and is, to the best of our knowledge, the first search algorithm for ornaments. `Lift` then lifts functions and proofs along those ornaments, removing all references to the old type. Both commands make some additional assumptions for simplicity; detailed explanations for these are in `Assumptions.v`.

3.1 Find ornament

In their original form, ornaments are a programming mechanism: Given a type A, an ornament determines some new type B. We invert this process for algebraic ornaments: Given types A and B, Devoid searches for an ornament between them. This is possible for algebraic ornaments precisely because the indexer is extensionally unique. For example, all possible indexers for `list` and `vector` must compute the length of a list; if we were to try doubling the length instead, we would not be able to satisfy the equivalence.

`Find ornament` takes two inductive types and searches for the components of the ornamental promotion isomorphism between them:

- **Inputs**: Inductive types A and B, assuming:
  - B is an algebraic ornament of A,
  - B has the same number of constructors in the same order as A,
  - A and B do not contain recursive references to themselves under products, and
  - for every recursive reference to A in A, there is exactly one new hypothesis in B, which is exactly the new index of the corresponding recursive reference in B.
- **Outputs**: Functions `promote`, `forget`, and `indexer`, guaranteeing:
  - the outputs form the ornamental promotion isomorphism between the inputs.

`Find ornament` includes an option to generate a proof that the outputs form the ornamental promotion isomorphism; by default, this option is false, since `Lift` does not need this proof.

[1] The adjunction condition follows from section and retraction.

3.2 Lift

`Lift` lifts a term along the ornamental promotion isomorphism between A and B. That is, it lifts types to corresponding types and terms of those types to corresponding terms:

```
Lift list vector in list as vector_p. (* vector_p T := Σ (n : nat).vector T n *)
Lift list vector in (cons 5 nil) as v_p. (* v_p := ∃ 1 (consV 0 5 nilV) *)
```

Furthermore, it recursively preserves this equivalence, lifting non-dependent functions like `zip` so that they map equivalent inputs to equivalent outputs:

```
∀ {T₁ T₂} l₁ l₂, promote (zip l₁ l₂) = zipV_p (promote l₁) (promote l₂).
```

This intuition breaks down with dependent types. With equivalence alone, we can't state the relationship between `zip_with_is_zip` and `zip_with_is_zipV_p`, since the unlifted conclusion:

```
zip_with pair l₁ l₂ = zip l₁ l₂.
```

does not have the same type as the conclusion of the lifted version applied to promoted arguments; any relation between these terms must be heterogenous.

In particular, `Lift` preserves the *univalent parametric relation* [30], a heterogenous parametric relation that strengthens an existing parametric relation for dependent types [2] to make it possible to state preservation of an equivalence: Two terms t and t' are related by the univalent parametric relation $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$ at type T in environment Γ if they are equivalent up to transport. The details of this relation can be found in the cited work.

`Lift` preserves this relation using the components that `Find ornament` discovers, and additionally guarantees that the lifted term does not refer to the old type in any way:

- **Inputs**: The inputs to and outputs from `Find ornament`, along with a term t, assuming:
  - the assumptions and guarantees from `Find ornament` hold,
  - I_B is not A,
  - t is well-typed and fully η-expanded,
  - t does not apply `promote` or `forget`, and
  - t does not reference B.
- **Outputs**: A term t', guaranteeing:
  - if t is $A\ \vec{i}$, then t' is $\Sigma(n : I_B\ \vec{i}\ ).B\ (\texttt{index}\ n\ \vec{i}\ )$,
  - t' does not reference A, and
  - if in the current environment $\Gamma \vdash t : T$, then $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$.

`Lift` does not require a proof that the input components form the ornamental promotion isomorphism, but they must for the guarantees to hold. It can operate in either direction, promoting from A to packed B or forgetting in the opposite direction; the specification for the forgetful direction is similar, with extra restrictions on how B is used within t.

4 Algorithms

This section describes the algorithms that implement the specifications from Section 3.

Presentation. We present both algorithms relationally, using a set of judgments; to turn these relations into algorithms, prioritize the rules by running the derivations in order, falling back to the original term when no rules match. The default rule for a list of terms is to run the derivation on each element of the list individually.

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

$$
\begin{array}{l}
\langle t \rangle ::= \langle v \rangle \mid \langle s \rangle \mid \Pi\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \\
\quad \lambda\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \langle t \rangle\ \langle t \rangle \mid \\
\quad \mathrm{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\} \mid \mathrm{Constr}\ (\langle i \rangle, \langle t \rangle) \mid \\
\quad \mathrm{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\}
\end{array}
$$

$\Gamma \vdash t : T$ // type checking
$\Gamma \vdash t_1 \equiv_{\beta\delta\iota} t_2$ // definitional equality
t_β // beta-reduction
$t_{\beta\delta\iota}$ // normalization
$t\ [y\ /\ x]$ // substitution
$\xi\ (I,\ Q,\ c,\ C)$ // type of eliminator

Figure 2 CIC_ω syntax (left, from existing work [31]) and judgments and operations (right).

$$
\begin{array}{ll}
A\ :=\ \mathrm{Ind}(Ty_A : \Pi(\vec{i_A} : \vec{\mathrm{X}_A}).\mathrm{s}_A)\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\} & \mathrm{P}_A\ :=\ \Pi(\vec{i_A} : \vec{\mathrm{X}_A})(a : A\ \vec{i_A}).\mathrm{s}_A \\
B\ :=\ \mathrm{Ind}(Ty_B : \Pi(\vec{i_B} : \vec{\mathrm{X}_B}).\mathrm{s}_B)\{\mathrm{C}_{B_1}, \ldots, \mathrm{C}_{B_n}\} & \mathrm{P}_B\ :=\ \Pi(\vec{i_B} : \vec{\mathrm{X}_B})(b : B\ \vec{i_B}).\mathrm{s}_B \\
\forall 1 \leq i \leq n, & \\
\quad \mathrm{E}_{A_i}\ (p_A : \mathrm{P}_A)\ :=\ \xi(A,\ p_A,\ \mathrm{Constr}(i,\ A),\ C_{A_i}) & \mathrm{index}\ :=\ \mathrm{insert}\ (\mathrm{off}\ A\ B) \\
\quad \mathrm{E}_{B_i}\ (p_B : \mathrm{P}_B)\ :=\ \xi(B,\ p_B,\ \mathrm{Constr}(i,\ B),\ C_{B_i}) & \mathrm{deindex}\ :=\ \mathrm{remove}\ (\mathrm{off}\ A\ B)
\end{array}
$$

Figure 3 Common definitions for both algorithms.

Notes on Syntax. The language the algorithms operate over is CIC_ω with primitive eliminators; this is a simplified version of the type theory underlying Coq. Figure 2 contains the syntax (which includes variables, sorts, product types, functions, inductive types, constructors, and eliminators), as well as the syntax for some judgments and operations, the rules for which are standard and thus omitted. For simplicity of presentation, we assume variables are names; we assume that all names are fresh. As in Coq, we assume the existence of an inductive type Σ for sigma types with projections π_l and π_r; for simplicity, we assume projections are primitive. Throughout, we use $\vec{i}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms, and we use $\vec{i}[j]$ to denote accessing the element of the list $\vec{i}$ at offset j.

Common Definitions. The algorithms assume list insertion and removal functions `insert` and `remove`, plus two functions Devoid implements: `off` computes the offset of the new index of type I_B in B's indices, and `new` determines whether a hypothesis in a case of the eliminator type of B is new. Figure 3 contains other common definitions, the names for which are reserved: The `index` and `deindex` functions insert an index into and remove an index from a list at the index computed by `off`. Input type A expands to an inductive type with indices of types $\vec{\mathrm{X}_A}$, sort s_A, and constructors $\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\}$. P_A denotes the type of the motive of the eliminator of A, and each E_{A_i} denotes the type of the eliminator for the ith constructor of A. Analogous names are also reserved for input type B.

4.1 Find ornament

The `Find ornament` algorithm implements the specification from Section 3.1. It builds on three intermediate steps: one to generate each of `indexer`, `promote`, and `forget`. Figure 4 shows the algorithm for generating `indexer`. The algorithms for generating `promote` and `forget` are similar; Figure 5 shows only the derivations for generating `promote` that are different from those for generating `indexer`, and the derivations for generating `forget` are omitted.

4.1.1 Searching for the Indexer

Search generates the `indexer` by traversing the types of the eliminators for A and B in parallel using the algorithm from Figure 4, which consists of three judgments: one to generate the motive, one to generate each case, and one to compose the motive and cases.

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_m} t}$$

Index-Motive

$$\frac{}{\Gamma \vdash (A, B) \Downarrow_{i_m} \lambda(\vec{i_A} : \vec{X_A})(a : A\ \vec{i_A}).(I_B\ \vec{i_A})_\beta}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_c} t}$$

Index-Conclusion

$$\frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{i_c} \vec{i_B}[\text{off}\ A\ B]}$$

Index-Hypothesis

$$\frac{\text{new}\ n_B\ b_B \qquad \Gamma,\ n_B : t_B \vdash (\Pi(n_A : t_A).b_A,\ b_B) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \Downarrow_{i_c} t}$$

Index-IH

$$\frac{\Gamma \vdash (A, B) \Downarrow_{i_m} p \qquad \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/\vec{i_B}[\text{off}\ A\ B]]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \Downarrow_{i_c} \lambda(n_A : p\ \vec{i_A}\ a).t}$$

Index-Prod

$$\frac{\Gamma,\ n_A : t_A \vdash (b_A,\ b_B[n_A/n_B]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \Downarrow_{i_c} \lambda(n_A : t_A).t}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i} t}$$

Index-Ind

$$\frac{\Gamma \vdash (A,\ B) \Downarrow_{i_m} p \qquad \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{i_c} \vec{f}}{\Gamma \vdash (A,\ B) \Downarrow_{i} \lambda(\vec{i_a} : \vec{X_A})(a : A\ \vec{i_a}).\mathrm{Elim}(a, p)\vec{f}}$$

Figure 4 Identifying the indexer function.

Generating the Motive. The $(T_A,\ T_B) \Downarrow_{i_m} t$ judgment consists of only the derivation Index-Motive, which computes the indexer motive from the types A and B (expanded in Figure 3). It does this by constructing a function with A and its indices as premises, and the type I_B in the conclusion with the appropriate indices. Consider `list` and `vector`:

```
list T := Ind (Ty_A : Type) {...}     vector T := Ind (Ty_B : Π(n :  nat).Type) {...}
```

For these types, Index-Motive computes the motive:

```
λ (l:list T) . nat
```

Generating Each Case. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_c} t$ judgment generates each case of the indexer by traversing in parallel the corresponding cases of the eliminator types for A and B. It consists of four derivations: Index-Conclusion handles base cases and conclusions of inductive cases, while Index-Hypothesis, Index-IH, and Index-Prod recurse into products.

Index-Hypothesis handles each new hypothesis that corresponds to a new index in an inductive hypothesis of an inductive case of the eliminator type for B. It adds the new index to the environment, then recurses into the body of only the type for which the index already exists. For example, in the inductive case of `list` and `vector`, `new` determines that `n` is the new hypothesis. Index-Hypothesis then recurses into the body of only the `vector` case:

```
Π (t_l:T) (l:list T) (IH_l:p_A l), ...     Π (t_v:T) (v:vector T n) (IH_v:p_B n v), ...
```

Index-Prod is next. It recurses into product types when the hypothesis is neither a new index nor an inductive hypothesis. Here, it runs twice, recursing into the body and substituting names until it hits the inductive hypothesis for both types:

```
Π (IH_l:p_A l), p_A (cons t_l l)          Π (IH_v:p_B n l), p_B (S n) (consV n t_l l)
```

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{p_m} t}$$

$$\textsc{Promote-Motive}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi}{\Gamma \vdash (A,\ B) \Downarrow_{p_m} \lambda(\vec{i_a} : \vec{\mathrm{X}_A})(a : A\ \vec{i_a}).B\ (\mathrm{index}\ (\pi\ \vec{i_a}\ a)\ \vec{i_a})}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{p_c} t}$$

$$\textsc{Promote-Conclusion}\ \frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{p_c} b}$$

$$\textsc{Promote-IH}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/b][\pi\ \vec{i_A}\ a/\vec{i_B}[\mathrm{off}\ A\ B]]) \Downarrow_{p_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \Downarrow_{p_c} \lambda(n_A : p\ \vec{i_A}\ a).t}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_p t}$$

$$\textsc{Promote-Ind}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{p_c} \vec{f}}{\Gamma \vdash (A,\ B) \Downarrow_p \lambda(\vec{i_A} : \vec{\mathrm{X}_A})(a : A\ \vec{i_A}).\exists\ (\pi\ \vec{i_A}\ a)\ (\mathrm{Elim}(a, p)\vec{f})}$$

Figure 5 Identifying the promotion function.

Index-IH then takes over. It substitutes the new motive in the inductive hypothesis, then recurses into both bodies, substituting the new inductive hypothesis for the index in the eliminator type for B. Here, it substitutes the new motive for p_A in the type of `IH`$_l$, extends the environment with `IH`$_l$, then substitutes `IH`$_l$ for `n`, so that it recurses on these types:

```
pA (cons tl l)                    pB (S IHl) (consV IHl tl l)
```

Finally, Index-Conclusion computes the conclusion by taking the index of motive p_B at off $A\ B$, here `S IH`$_l$. In total, this produces a function that computes the length of `cons t l`:

```
λ (tl:T) (l:list T) (IHl:(λ (l:list T).nat) l).S IHl
```

Composing the Result. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_i t$ judgment consists of only Index-Ind, which identifies the motive and each case using the other two judgments, then composes the result. In the case of `list` and `vector`, this produces a function that computes the length of a list:

```
λ (l:list T).Elim(l, λ (l:list T).nat)
  {0, λ (tl:T) (l:list T) (IHl:(λ (l:list T).nat) l).S IHl}
```

4.1.2 Searching for Promote and Forget

Figure 5 shows the interesting derivations for the judgment $(T_A,\ T_B) \Downarrow_p t$ that searches for `promote`: Promote-Motive identifies the motive as B with a new index (which it computes using `indexer`, denoted by metavariable π). When Promote-IH recurses, it substitutes the inductive hypothesis for the term rather than for its index, and it substitutes the new index (which it also computes using `indexer`) inside of that term. Promote-Conclusion returns the entire term, rather than its index. Finally, Promote-Ind not only recurses into each case, but also packs the result.

```
↑    {i⃗_a : X⃗_A} := promote i⃗_a.          ↓    {i⃗_b : X⃗_B} := forget i⃗_b.
π_{I_B} {i⃗_a : X⃗_A} := indexer i⃗_a.       ∃_{I_B} {i⃗_b : X⃗_B} (b : B i⃗_b) := ∃ i⃗_b[off] b.
↑_B  := π_r ∘ ↑.                          ↓_A  := ↓   ∘ ∃_{I_B}.
↑_{I_B} := π_l ∘ ↑.                       ↓_{I_B} := π_{I_B} ∘ ↓_A.
```

Figure 6 Common definitions for the core lifting algorithm.

The omitted derivations to search for `forget` are similar, except that the domain and range are switched. Consequentially, `indexer` is never needed; Forget-Motive removes the index rather than inserting it, and Forget-IH no longer substitutes the index. Additionally, Forget-Hypothesis adds the hypothesis for the new index rather than skipping it, and Forget-Ind eliminates over the projection rather than packing the result.

4.1.3 Core Search Algorithm

The core search algorithm produces `indexer`, `promote`, and `forget`, then composes them into a tuple. This tuple is how Devoid represents ornaments internally. Devoid includes an option to generate a proof that these components form the ornamental promotion isomorphism; by default, this is disabled, since `Lift` does not need this proof. The implementation of this option gives intuition for correctness of the search algorithm, and is described in Section 5.3.

4.2 Lift

The `Lift` algorithm implements the specification from Section 3.2. We show only one direction of the algorithm, promoting from A to packed B; the forgetful direction is similar. The core algorithm (Figure 9) builds on a set of common definitions (Figure 6) and two intermediate judgments: one to lift eliminators (Figure 7) and one to lift constructors (Figure 8).

Common Definitions. The common definitions (Figure 6) define some useful syntax: $\uparrow$ applies `promote`, $\downarrow$ applies `forget`, and π_{I_B} applies `indexer`. $\exists_{I_B}$ packs a term of type B into an existential with the index at the appropriate offset. $\uparrow_B$ and $\uparrow_{I_B}$ promote and then project; $\downarrow_A$ packs and forgets, and $\downarrow_{I_B}$ packs, forgets, and then applies `indexer` to project the index.

4.2.1 Lifting Eliminators

The $\Gamma \vdash t \Uparrow_E t'$ judgment (Figure 7) defines rules for lifting the motive and case of an eliminator, changing the *domain of induction* from A to B. The intuition is that any term of type A is the result of forgetting some term of type packed B. Then, since A and B have the same inductive structure, we can lift the eliminator of A to the eliminator of B, and move that forgetfulness *inside of each case*. For example, the following terms are propositionally equal:

```
Elim(↓_A b, p_A){                    Elim(b, λ(n:nat)(v:vector T n).p_A (↓_A v)){
  f_nil,                               f_nil,
  (λ(t_l:T)(l:list T)(IH_l:p_A l).     (λ(n:nat)(t_v:T)(v:vector T n)(IH_v:p_A (↓_A v)).
    f_cons t_l l IH_l)                   f_cons t_v (↓_A v) IH_v)
}                                    }
```

The induction rules implement this transformation. Case lifts a case of the eliminator by first recursively lifting the motive, then using the lifted motive to compute the type of the new case, and then using that type to compute the body of the new case. In the example

$\boxed{\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'}$

$$\textsc{Drop-Index}\ \frac{\text{new } n\ b \qquad \Gamma,\ n : t \vdash (f,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'}$$

$$\textsc{Forget-Arg}\ \frac{\Gamma \vdash \vec{i} : \vec{\mathrm{X}_B} \qquad \Gamma,\ n : B\ \vec{i} \vdash ((f\ (\downarrow_A n))_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : B\ \vec{i}).b) \Uparrow_{E_x} \lambda(n : B\ \vec{i}).b'}$$

$$\textsc{Arg}\ \frac{\Gamma,\ n : t \vdash ((f\ n)_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'}$$

$$\textsc{Concl}\ \frac{}{\Gamma \vdash (t,\ p_B\ \vec{y}) \Uparrow_{E_x} t}$$

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

$$\textsc{Motive}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A}{\Gamma \vdash p_A \Uparrow_E \lambda(\vec{i} : \vec{\mathrm{X}_B})(b : B\ \vec{i}).(p_A\ (\text{deindex } \vec{i})\ (\downarrow_A b))_\beta}$$

$$\textsc{Case}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A \qquad \Gamma \vdash f_i : \mathrm{E}_{A_i}\ p_A \qquad \Gamma \vdash p_A \Uparrow_E p_B \qquad \Gamma \vdash (f_i,\ \mathrm{E}_{B_i}\ p_B) \Uparrow_{\mathrm{E_x}} f_i'}{\Gamma \vdash f_i \Uparrow_E f_i'}$$

Figure 7 Lifting eliminators.

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

$$\textsc{Normalize}\ \frac{}{\Gamma \vdash \mathrm{Constr}(j,\ A)\ \vec{x} \Uparrow_C (\uparrow (\mathrm{Constr}(j,\ A)\ \vec{x}))_{\beta\delta\iota}}$$

Figure 8 Lifting constructors.

above, when lifting the inductive case, it first recursively lifts the motive p_A using Motive, which drops the index, packs and forgets the argument of type B, and then β-reduces the result, eliminating references to B. This produces the new motive:

```
λ(n:nat)(v:vector T n).pA (↓A v)
```

which Case then uses to compute the type of the inductive case of the eliminator for B:

```
Π(tv:T)(n:nat)(v:vector T n)(IHv:pA (↓A v)).pA (↓A (consV tv (S n) v))
```

The $\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'$ judgment then uses that type to compute the lifted function body. It computes this in a similar way to Motive, except that there are as many indices to drop and arguments to pack and forget as there are inductive hypotheses, and these do not occur in predictable places, so more rules are involved. This computes the new function:

```
λ(n:nat)(tv:T)(v:vector T n)(IHv:pA (↓A v)).fcons tv (↓A v) IHv
```

4.2.2 Lifting Constructors

The $\Gamma \vdash t \Uparrow_C t'$ judgment (Figure 8) lifts applications of constructors of A to applications of constructors of B. This judgment computes one step of the promotion, leaving the recursive lifting of the arguments to the final algorithm. Using the same types, in the base case:

```
↑ nil ≡βδι ∃ O nilV
```

and in the inductive case:

```
↑ (cons t l) ≡βδι ∃ (S (↑IB l)) (consV (↑IB l) t (↑B l))
```

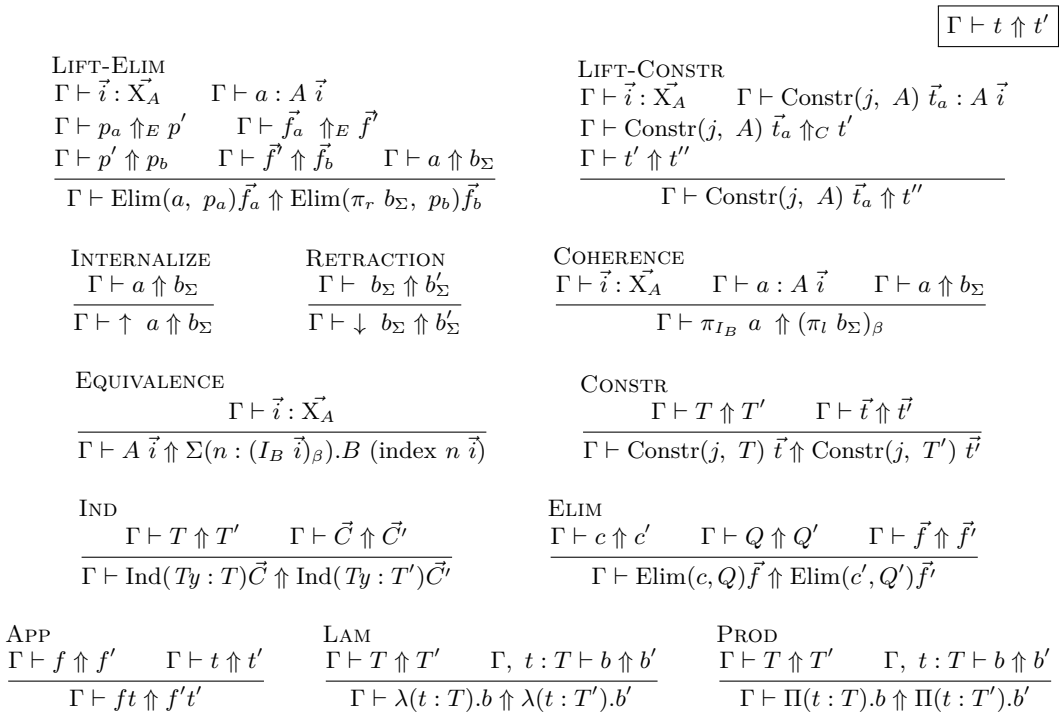

Figure 9 Core lifting algorithm.

This derivation consists of only one rule: Normalize, which normalizes the promotion of the constructor. This is guaranteed to succeed because the application of the constructor is fully η-expanded. The core algorithm later internalizes the promotion functions in the result.

4.2.3 Core Lifting Algorithm

The core algorithm (Figure 9) builds on these intermediate judgments. The interesting derivations for correctness are the first six: Lift-Elim and Lift-Constr use the judgments for lifting eliminators and constructors of A. Internalize internalizes the explicit `promote` functions from the lifted constructors to recursive applications of the algorithm. Retraction and Coherence use the respective properties of the ornamental promotion isomorphism metatheoretically: the first to drop the explicit `forget` functions from the lifted eliminators, and the second to lift the `indexer` to a projection (in the forgetful direction, Section replaces Retraction). Finally, Equivalence lifts A along the equivalence to packed B. The remaining derivations recurse predictably.

5 Implementation

The Devoid Coq plugin implements the algorithms from Section 4; the link to the code is in **Supplement Material**. Devoid cannot produce an ill-typed term, since Coq type checks all terms that plugins produce and rejects ill-typed terms. The implementations of `Find ornament` (`search.ml`) and `Lift` (`lift.ml`) are mostly the same as the algorithms, but with changes to address implementation challenges that scale the algorithms to a Coq tool for proof engineers. This section describes a sample of these changes from each of three categories: addressing differences between Coq and the type theory that the algorithms assume (Section 5.1), optimizing for efficiency (Section 5.2), and improving usability (Section 5.3).

5.1 Addressing Language Differences

Fixpoints. Coq implements eliminators in terms of pattern matching and fixpoints. To handle terms that use these features, Devoid includes a `Preprocess` command that translates these terms into equivalent eliminator applications. This command can preprocess a definition (like `zip` from Section 2) or an entire module (like `List`, as shown in `ListToVect.v`) for lifting. It currently supports fixpoints that are structurally recursive on only immediate substructures. To translate such a fixpoint, it first extracts a motive, then generates each case by partially reducing the function's body under a hypothetical context for the constructor arguments. This is enough to preprocess `List`; Section 8 discusses possible extensions.

Non-Primitive Projections. By default, projections in Coq are non-primitive. That is, this:

```
∀ (T : Type) (v : Σ (n : nat).vector T n), v = ∃ (π_l v) (π_r v).
```

cannot be proven by reflexivity alone (see `Projections.v`). Therefore, Devoid must pack terms like `v` into existentials; otherwise, lifting will sometimes fail. This is why the type of `zip_with_is_zipV_p` in the example from Section 2 packs $\mathtt{v}_1$ and $\mathtt{v}_2$. For the sake of performance and readability of lifted code, Devoid is strategic about when it packs.

Constants. Because Coq has constants, the implementation of Normalize refolds [3] after normalizing. That is, it acts like the `simpl` tactic in Coq, but with special support for sigma types. For example, to lift the `cons` constructor of a list, after normalizing the promotion of `cons t l`, Devoid substitutes the projections of the promotion of `l` for their normal forms, which determines and saves the following fact:

```
∀ {T} (l : list T), ↑ (cons t l) = ∃ (S (↑_{I_B} l)) (consV (↑_{I_B} l) t (↑_B l)).
```

Refolding helps produce more readable lifted code. It also improves lifting performance, since it occurs just once for each constructor.

5.2 Optimizing for Efficiency

Delayed Reduction. When lifting eliminators, Devoid computes a list of arguments and delays reduction. It computes this list backwards, storing the new indices that inductive hypotheses refer to as it recurses. This removes the call to `new` in the premise of Drop-Index.

Lazy η-Expansion. The lifting algorithm assumes that all terms are fully η-expanded. Sometimes, however, η-expansion is not necessary. For efficiency, rather than fully η-expand ahead of time, Devoid η-expands lazily, only when it is necessary for correctness.

Caching. To prevent extra recursion, Devoid caches the outputs of search, as well as lifted constants, inductive types, and constructors. Since these are constants, lookup is low-cost.

5.3 Improving Usability

Correctness Proofs. Devoid has options (used in `Example.v`) that tell search to generate proofs that its outputs are correct, thereby increasing confidence in and usefulness of those outputs. The proof of `coherence` is reflexivity. The intuition behind the automation to prove `section` and `retraction` (`equivalence.ml`) is that `promote` and `forget` map along corresponding constructors, so inductive cases preserve equalities. Thus, each inductive case of these proofs is generated by a fold that rewrites each recursive reference, with reflexivity as identity.

Unpacking. Devoid includes an `Unpack` command (used in `Example.v`) that unpacks packed types in functions and proofs. This way, users may access unpacked terms without writing boilerplate code. For simple functions, this command packs arguments and projects results. It splits higher-order functions into two functions. For proofs that use equality, it applies one lemma convert to dependent equality, and one lemma to deal with non-primitive projections.

User-Friendly Types. `Example.v` describes how the user can recover user-friendly types after unpacking. For example, to recover a function with an output of type `vector T n`, the user lifts a proof that the length of the output of the unlifted `list` version of that function is `n`, then rewrites by that lifted proof. The intuition behind this is that this equivalence holds:

```
{ l : list T & length l = n } ≃ vector T n
```

Recovering a user-friendly type for a proof relating these functions is more complex, since it necessitates reasoning at some point about equalities between equalities. For some index types like `nat`, this follows simply from the fact that the type forms an h-set [32]: all proofs of equality between the same two terms of that type are equal. There is preliminary work on determining a general methodology for deriving user-friendly types for proofs that does not rely on any properties of the index type. The idea is to use the adjunction condition along with the proof of `coherence` by reflexivity; see GitHub issue #39 for the status of this work.

6 Case Study

We used Devoid to automatically discover and lift along ornaments for two scenarios:

1. Single Iteration: from binary trees to sized binary trees
2. Multiple Iterations: from binary trees to binary search trees to AVL trees

For comparison, we also used the ornaments that Devoid discovered to lift functions and proofs using *Equivalences for Free!* [30] (EFF), a more general framework for lifting across equivalences. Devoid produced faster functions and smaller terms, especially when composing multiple iterations of lifting. In addition, Devoid imposed little burden on the user, and the ornaments Devoid discovered proved useful to EFF.

We chose EFF for comparison because Devoid is the only tool for ornaments in Coq, and because doing so demonstrates the benefits of specialized automation for ornaments. Devoid can handle only a small class of equivalences compared to EFF, and it can currently handle only incremental changes to types (one new index at a time). Our experiences suggest that it is possible to use both tools in concert. Section 7 discusses EFF in more detail.

Setup. The case study code is in the `eval` folder of the repository. For each scenario, we ran Devoid to search for an ornament, and then lifted functions and proofs along that ornament using both Devoid and EFF. We noted the amount of user interaction (Section 6.1), as well as the performance of lifted terms (Section 6.2). To test the performance of lifted terms, we tested runtime by taking the median of ten runs using `Time Eval vm_compute` with test values in Coq 8.8.0, and we tested size by normalizing and running `coqwc` on the result.[2]

[2] i5-5300U, at 2.30GHz, 16 GB RAM

In the first scenario, we lifted traversal functions along with proofs that their outputs are permutations of each other from binary trees (`tree`) to sized binary trees (`Sized.tree`). In the second scenario, we lifted the traversal functions to AVL trees (`avl`) through four intermediate types (one for each new index), and we lifted a search function from BSTs (`bst`) to AVL trees through one intermediate type. Both scenarios considered only full binary trees.

To fit `bst` and `avl` into algebraic ornaments for Devoid, we used boolean indices to track invariants. While the resulting types are not the most natural definitions, this scenario demonstrates that it is possible to express interesting changes to structured types as algebraic ornaments, and that lifting across these types in Devoid produces efficient functions.

6.1 User Experience

For each intermediate type in each scenario, we used Devoid to discover the components of the equivalence. These components were enough for Devoid to lift functions and proofs with no additional proof burden and no additional axioms. To use EFF, we also had to prove that these components form an equivalence; we set the appropriate option to generate these proofs using Devoid. In addition, to use EFF, we had to prove univalent parametricity of each inductive type; these proofs were small, but required specialized knowledge. To lift the proof of the theorem `pre_permutes` using EFF, we had to prove the univalent parametric relation between the unlifted and lifted versions of the functions that the theorem referenced; this pulled in the functional extensionality axiom, which was not necessary using Devoid.

In the second scenario, to simulate the incremental workflow Devoid requires, we lifted to each intermediate type, then unpacked the result. For example, the ornament from `bst` to `avl` passed through an intermediate type; we lifted `search` to this type first, unpacked the result, and then repeated this process. In this scenario, using EFF differently could have saved some work relative to Devoid, since with EFF, it is possible to skip the intermediate type;[3] Devoid is best fit where an incremental workflow is desirable.

6.2 Performance

Relative to EFF, Devoid produced faster functions. Table 1 summarizes runtime in the first scenario for `preorder`, and Table 2 summarizes runtime in the second scenario for `preorder` and `search`. The `inorder` and `postorder` functions performed similarly to `preorder`. The functions Devoid produced imposed modest overhead for smaller inputs, but were tens to hundreds of times faster than the functions that EFF produced for larger inputs. This performance gap was more pronounced over multiple iterations of lifting.

Devoid also produced smaller terms: in the first scenario, 13 vs. 25 LOC for `preorder`, 12 vs. 24 LOC for `inorder`, and 17 vs. 29 LOC for `postorder`; and in the second scenario, 21 vs. 120 LOC for `preorder`, 20 vs. 119 LOC for `inorder`, 24 vs. 125 LOC for `postorder`, and 31 vs. 52 LOC for `search`. In the first scenario, the lifted proof of `pre_permutes` using Devoid was 85 LOC; the lifted proof of `pre_permutes` using EFF was 1463184 LOC.

We suspect Devoid provided these performance benefits because it directly lifted induction principles, whereas EFF produced lifted functions in terms of unlifted functions. The multiple iteration case in particular highlights this, since EFF's approach makes lifted terms much slower and larger as the number of iterations increases, while Devoid's approach does not.

[3] The performances of the terms that EFF produces are sensitive to the equivalence used; for a 100 node tree, this alternate workflow produced a search function which is hundreds of times slower and traversal functions which are thousands of times slower than the functions that Devoid produced. In addition, the lifted proof of `pre_permutes` using EFF failed to normalize with a timeout of one hour.

Table 1 Median runtime (ms) of unlifted (`tree`) and lifted (`Sized.tree`) `preorder` over ten runs with test inputs ranging from about 10 to about 10000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	0.0	0.0	0.0	3.0 (1.00x)	35.0 (0.95x)
	EFF	0.0	1.0	27.0	486.5 (162.17x)	8078.5 (218.33x)

Table 2 Median runtime (ms) of unlifted (`tree`) and lifted (`avl`) `preorder`, plus unlifted (`bst`) and lifted (`avl`) `search`, over ten runs with inputs ranging from about 10 to about 100000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	71.5	71.0	69.0	75.0 (25.00x)	109.0 (2.95x)
	EFF	1.0	11.0	152.0	2976.5 (992.17x)	56636.5 (1530.72x)
`search`	**Unlifted**	0.0	0.0	2.0 (1.00x)	3.0 (1.00x)	29.0 (1.00x)
	Devoid	12.0	14.0	12.0 (6.00x)	15.0 (5.00x)	50.0 (1.72x)
	EFF	1.0	5.0	67.0 (33.50x)	1062.0 (354.00x)	15370.5 (530.02x)

7 Related Work

Ornaments. Devoid automates discovery of and lifting across algebraic ornaments in a higher-order dependently typed language. In the decade since the discovery of ornaments [23], there have been a number of formalizations and embedded implementations of ornaments [10, 19, 11, 20, 9]. Devoid is the first tool for ornamentation to operate over a non-embedded dependently typed language. It essentially moves the automation-heavy approach of Ornamentation in ML [33], which operates on non-embedded ML code, into the type theory that forms the basis of theorem provers like Coq. In doing so, it takes advantage of the properties of algebraic ornaments [23]. It also introduces the first search algorithm to identify ornaments, which in the past was identified as a "gap" in the literature [20].

Lifting Proofs. Devoid identifies and lifts proofs along a specific equivalence similar to that from existing ornaments work [20]. The need to automatically lift functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [22, 1, 21, 16, 34, 6]. The univalence axiom from Homotopy Type Theory [32] enables transparent transport of proofs; cubical type theory [5] gives univalence a constructive interpretation.

Our work is closely related to *Equivalences for Free!* [30], which brings this full circle, using mathematical properties of univalence to enable lifting across equivalences in a substantial subset of CIC_ω without relying on the univalence axiom. In doing so, it introduces and formalizes the relation that our specification depends upon, and implements a framework for lifting in Coq. This framework is more general than Devoid: It lifts along any equivalence, not just ornamental promotions, and can handle opaque terms, with the caveat that users must prove each equivalence themselves; Devoid requires non-opaque terms and lifts along the class of equivalences that correspond to ornamental promotions, taking advantage of the mathematical properties of ornaments to eliminate the need for explicit applications of section and retraction, and to discover and prove certain equivalences automatically. These mathematical properties allow us to automatically lift the induction principle and eliminate references to old terms, which is beneficial for performance.

Similarly, our work is related to CoqEAL [6], which transfers functions along arbitrary relations between types. As these relations do not necessarily need to be equivalences, this framework is more general than our work. Similar tradeoffs between automation and generality apply: CoqEAL produces functions that refer to the old type, and does not yet support automatic inference of relations. In addition, CoqEAL currently only supports automatic transfer of functions, and does not yet handle proofs.

These tools may provide an alternative backend for Devoid. Furthermore, our search algorithm may help discover relations that make these tools easier to use, and our lifting algorithm may help improve automation and efficiency for certain relations in these tools.

Program and Proof Reuse. The problem that we solve is fundamentally about proof reuse, which applies software reuse principles to ITPs. There is a wealth of work in proof reuse, from tactic languages [15] and logical frameworks [4], to tools for proof abstraction and generalization [26, 18], to domain-specific methodologies [12] and frameworks [13].

Devoid focuses on the specific problem of reuse when adding fully-determined indices to types. Other approaches to this problem include combinators which definitionally reduce to desirable terms [14] in the language Cedille, and automatic generation of conversion functions in Ghostbuster [24] for GADTs in Haskell. Our work focuses on a type theory different from both of these, in which the properties that allow for such combinators in Cedille are not present, and in which dependent types introduce challenges not present in Haskell.

Devoid is not the first tool to combine search with reuse. Optician [25] synthesizes bidirectional string transformations; a similar approach may help extend tooling to handle transformations for low-level data. Pumpkin Patch [27] searches the difference in proofs for patches that can be used to repair proofs broken by changes; Devoid uses a similar approach to identify functions that form an equivalence. The resulting tools are complementary: Devoid supports the addition of indices and hypotheses, which Pumpkin Patch does not support; Pumpkin Patch supports changes in values, which Devoid does not support.

8 Conclusions & Future Work

We presented Devoid: a tool for searching for and lifting across algebraic ornaments in Coq. Devoid is the first tool to lift across ornaments in a non-embedded dependently typed language, and to automatically infer certain kinds of ornaments from types alone. Our algorithms give efficient transport across equivalences arising from algebraic ornaments; our case study demonstrates that such automation can make lifted terms smaller and faster as part of an incremental workflow.

Future Work. A future version may support other ornaments beyond algebraic ornaments, with additional user interaction as needed; this may help support, for example, the ornament between `nat` and `list`, where `list` has a new element in the `cons` case. A future version may loosen restrictions on input types to support adding constructors while preserving inductive structure, recursive references under products, and coinductive types. Integrating with Pumpkin Patch [27] may help remove restrictions Devoid makes about the hypotheses of B. `Preprocess` currently supports only certain fixpoints; a more general translation may help Devoid support more terms, and discussions with Coq developers suggest that the implementation of such a translation building on work from the equations [28] plugin is in progress. Extending Devoid to generate proofs of coherence conditions for lifted terms

may increase user confidence. Proofs that the commands that DEVOID implements satisfy their specifications may also increase user confidence. Better automating the recovery of user-friendly types may improve user experience.

References

1 Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.

2 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22:107–152, March 2012. doi:10.1017/S0956796812000056.

3 Pierre Boutillier. *New tool to compute with inductive in Coq*. Theses, Université Paris-Diderot - Paris VII, February 2014. URL: https://tel.archives-ouvertes.fr/tel-01054723.

4 Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 106–113. ACM, 1995.

5 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint*, 2016. arXiv:1611.02108.

6 Cyril Cohen and Damien Rouhling. A refinement-based approach to large scale reflection for algebra. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017. URL: https://hal.inria.fr/hal-01414881.

7 coq-club. [Coq-Club] Dealing with equalities in dependent types. http://sympa.inria.fr/sympa/arc/coq-club/2017-01/msg00099.html, 2017. Accessed: 2019-03-25.

8 coq-club. [Coq-Club] Trouble with dependent induction. http://sympa.inria.fr/sympa/arc/coq-club/2017-12/msg00079.html, 2017. Accessed: 2019-03-25.

9 Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27, 2017.

10 Pierre-Evariste Dagand and Conor McBride. A Categorical Treatment of Ornaments. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 530–539, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/LICS.2013.60.

11 Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

12 Benjamin Delaware, William Cook, and Don Batory. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 595–608, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048113.

13 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429094.

14 Larry Diehl, Denis Firsov, and Aaron Stump. Generic Zero-Cost Reuse for Dependent Types. *CoRR*, abs/1803.08150, 2018. arXiv:1803.08150.

15 Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 1–15. Springer, 1994.

16 Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 131–146. Springer, 2013.

17 Inria. The Coq Standard Library. http://coq.inria.fr/distrib/current/stdlib. Accessed: 2019-03-15.

18 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher Order Logics*, pages 152–167. Springer, 2004.

19 Hsiang-Shang Ko and Jeremy Gibbons. Relational algebraic ornaments. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming*, pages 37–48. ACM, 2013.

20 Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016.

21 Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

22 Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.

23 Conor McBride. Ornamental algebras, algebraic ornaments, 2011. URL: `http://plv.mpi-sws.org/plerg/papers/mcbride-ornaments-2up.pdf`.

24 Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. Ghostbuster: A Tool for Simplifying and Converting GADTs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 338–350, New York, NY, USA, 2016. ACM. `doi:10.1145/2951913.2951914`.

25 Anders Miltner, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages*, 2(POPL):1, 2017.

26 Olivier Pons. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs*, pages 217–232. Springer, 2000.

27 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129. ACM, 2018.

28 Matthieu Sozeau. Equations: A Dependent Pattern-Matching Compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

29 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. hs-to-coq. `https://github.com/antalsz/hs-to-coq`, 2018-2019. Accessed: 2019-03-12.

30 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, July 2018. `doi:10.1145/3236787`.

31 Amin Timany and Bart Jacobs. First Steps Towards Cumulative Inductive Types in CIC. In *ICTAC*, 2015.

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

33 Thomas Williams and Didier Rémy. A Principled Approach to Ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, December 2017. `doi:10.1145/3158109`.

34 Theo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. *arXiv preprint*, 2015. `arXiv:1505.05028`.