Generic Zero-Cost Reuse for Dependent Types

LARRY DIEHL, University of Iowa, U.S.A
DENIS FIRSOV, University of Iowa, U.S.A
AARON STUMP, University of Iowa, U.S.A

Dependently typed languages are well known for having a problem with code reuse. Traditional non-indexed algebraic datatypes (e.g. lists) appear alongside a plethora of indexed variations (e.g. vectors). Functions are often rewritten for both non-indexed and indexed versions of essentially the same datatype, which is a source of code duplication.

We work in a Curry-style dependent type theory, where the same untyped term may be classified as both the non-indexed and indexed versions of a datatype. Many solutions have been proposed for the problem of dependently typed reuse, but we exploit Curry-style type theory in our solution to not only reuse data and programs, but do so at zero-cost (without a runtime penalty). Our work is an exercise in dependently typed generic programming, and internalizes the process of zero-cost reuse as the identity function in a Curry-style theory.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: dependent types, generic programming, reuse

ACM Reference Format:
Larry Diehl, Denis Firsov, and Aaron Stump. 2018. Generic Zero-Cost Reuse for Dependent Types. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (March 2018), 27 pages.

1 INTRODUCTION

Dependently typed languages (such as Agda [Norell 2007], Coq [The Coq Development Team 2008], Idris [Brady 2013], or Lean [de Moura et al. 2015]) can be used to define ordinary algebraic datatypes, as well as indexed versions of algebraic datatypes that enforce various correctness properties. For example, we can index lists by natural numbers to enforce that they have a particular length (i.e. $\texttt{Vec}_A : \mathbb{N} \to \star$). Similarly, we can index lists by two elements to enforce that they are ordered and have a lower and upper bound (i.e. $\texttt{OList}_{A,R} : A \to A \to \star$). We can even combine these two forms of indexing to enforce that lists have all of the aforementioned correctness properties (i.e. $\texttt{OVec}_{A,R} : A \to A \to \mathbb{N} \to \star$).

Which datatype a programmer uses depends upon how much correctness they wish to enforce at the time a function is written, versus proving correctness as a separate step sometime later (corresponding to intrinsic and extrinsic correctness proofs of functions). Certain types tend to be better suited to writing intrinsically correct functions than others, e.g. it is natural to define a safe `lookup` function that takes a `Vec` as an argument, and a correct `sort` function that returns an `OList`.

However, once we have written a function using a suitable indexed variant of a datatype, reusing the function to define a corresponding version for the unindexed (or less indexed) datatype variants can be painful. We may also wish to delay extrinsic verification, thereby suffering later while reusing a function over unindexed (or less indexed) datatype variants to define a corresponding

Authors' addresses: Larry Diehl, University of Iowa, U.S.A; Denis Firsov, University of Iowa, U.S.A; Aaron Stump, University of Iowa, U.S.A.

2018. 2475-1421/2018/3-ART1 $15.00
https://doi.org/

function over more indexed indexed datatypes. We refer to the former direction as *forgetful reuse*, and to the latter direction as *enriching reuse*.

One source of pain is *manually writing* functions over some datatypes by reusing functions over differently indexed variants of the same underlying datatypes. Another source of pain is that reusing functions involves linear time conversions between differently indexed types, resulting in a runtime *performance penalty* incurred by practicing the good software engineering practice of code reuse. In this paper we address both of these problems, for both the forgetful and enriching directions of reuse, by:

(1) Defining generic combinators to incrementally attack the problem of reuse for various types, where each combinator application results in simplified subgoals (similar to tactics).
(2) Ensuring that the combinators are closed operations with respect to a type abstraction, which can be eliminated to obtain reused functions at zero-cost (i.e. no performance penalty).

Our **primary contributions** are:

(1) Section 4.2: Generic combinator solutions to zero-cost *forgetful program reuse* (combinator `allArr2arr`, handling the type of non-dependent functions), and *proof reuse* (combinator `allPi2pi`, handling the type of dependent functions).
(2) Section 4.3: Generic combinator solutions to zero-cost *enriching program reuse* (combinator `arr2allArrP`, handling the type of non-dependent functions), and *proof reuse* (combinator `pi2allPiP`, handling the type of dependent functions).
(3) Section 5.3: A generic combinator solution to zero-cost *forgetful data reuse* (combinator `ifix2fix`, handling the type of fixpoints for generically encoded datatypes).
(4) Section 5.5: A generic combinator solution to zero-cost *enriching data reuse* (combinator `fix2ifix`, handling the type of fixpoints for generically encoded datatypes).

The remainder of our paper proceeds as follows:

- Section 2: We review background material, covering the Curry-style type theory that our results are developed within, and providing intuition for why zero-cost conversions are motivated by Curry-style type theory.
- Section 3: We explain the primary problems (linear time reuse of programs, proofs, and data) we are solving through concrete examples, and provide manual solutions (zero-cost, or constant time, reuse), which our primary contribution combinators generalize via generic programming.
- Section 4: We generically solve the problems of (both forgetful and enriching) zero-cost program and proof reuse (as combinators for the types of non-dependent and dependent functions).
- Section 5: We generically solve the problems of (both forgetful and enriching) zero-cost data reuse (as combinators for the type of fixpoints).
- Section 6: We compare what we have done with related work. This includes comparing our results with the closely related work of dependently typed reuse via ornaments [McBride 2011] and dependent interoperability [Dagand et al. 2016], the primary difference being that our work achieves *zero-cost* reuse.
- Section 7: We go over extensions that we have already made to our work, not covered herein, as well as planned future work.

All of our results have been formalized in Cedille [Stump 2017, 2018], a dependently typed language implementing the theory we work in (covered in Section 2.1).[1]

[1] The Cedille code accompanying this paper is here:
https://github.com/larrytheliquid/generic-reuse

$$\frac{\Gamma, x : T' \vdash t : T \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda\, x{:}T'.\, t : \forall\, x{:}T'.\, T} \qquad \frac{\Gamma \vdash t : \forall\, x{:}T'.\, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ \ {-}t' : [t'/x]T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \beta : t \simeq t} \qquad \frac{\Gamma \vdash q : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash \rho\, q\ -\ t : [t_2/x]T}$$

$$\frac{\Gamma \vdash q : t_1 \simeq t_2 \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \phi\, q\ -\ t_1\{t_2\} : T} \qquad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T.\, T'}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 : T} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 : [t.1/x]T'}$$

Fig. 1. Introduction and elimination rules for additional type constructs

$$\begin{array}{lcl}
|\Lambda\, x{:}T.\, t| & = & |t| \\
|t\ \ {-}t'| & = & |t| \\
|\beta| & = & \lambda\, x.\, x \\
|\rho\, q\ -\ t| & = & |t| \\
|\phi\, q\ -\ t_1\{t_2\}| & = & |t_2| \\
|[t_1, t_2]| & = & |t_1| \\
|t.1| & = & |t| \\
|t.2| & = & |t|
\end{array}$$

Fig. 2. Erasure rules for additional type constructs

2 BACKGROUND

2.1 The Type Theory (CDLE)

We briefly summarize the type theory, the Calculus of Lambda Eliminations (CDLE), that the results of this paper depend on. For full details on CDLE, including semantics and soundness results, please see the previous papers [Stump 2017, 2018]. The main metatheoretic property proved in the previous work is logical consistency: there are types which are not inhabited. Cedille is an implementation of CDLE, and all the code appearing in this paper is Cedille code.

CDLE is an extrinsic (i.e. Curry-style) type theory, whose terms are exactly those of the pure untyped lambda calculus (with no additional constants or constructs). The type-assignment system for CDLE is not subject-directed, and thus cannot be used directly as a typing algorithm. Indeed, since CDLE includes Curry-style System F as a subsystem, type assignment is undecidable [Wells 1999]. To obtain a usable type theory, Cedille thus has a system of annotations for terms, where the annotations contain sufficient information to type terms algorithmically. But true to the extrinsic nature of the theory, these annotations play no computational role. Indeed, they are erased both during compilation and before formal reasoning about terms within the type theory, in particular by definitional equality (see Figure 1 and Figure 2).

CDLE extends the (Curry-style) Calculus of Constructions (CC) with implicit products, primitive heterogeneous equality, and intersection types:

- `∀ x : T. T'`, the implicit product type of Miquel [2001]. This can be thought of as the type for functions which accept an erased input of type `x : T`, and produce a result of type `T'`. There are term constructs `Λ x. t` for introducing an implicit input `x`, and `t -t'` for instantiating such an input with `t'`. The implicit arguments exist just for purposes of typing so that they play no computational role and equational reasoning happens on terms from which the implicit arguments have been erased.
- $t_1 \simeq t_2$, a Curry-style heterogeneous equality type. The terms t_1 and t_2 are required to be typed, but need not have the same type. We introduce this with a constant β which erases to `λ x. x` (so our type-assignment system has no additional constants, as promised); β proves `t ≃ t` for any typeable term `t`. Combined with definitional equality, β proves $t_1 \simeq t_2$ for any β-equal t_1 and t_2 whose free variables are all declared in the typing context. We eliminate the equality type by rewriting, with a construct `ρ q - t`. Suppose `q` proves $t_1 \simeq t_2$ and we synthesize a type `T` for `t`, where `T` has several occurrences of terms definitionally equal to t_1. Then the type synthesized for `ρ q - t` is `T` except with those occurrences replaced by t_2. The construct φ `q -` $t_1\{t_2\}$ casts a term t_2 (of any type) to type `T`, provided that t_1 has type `T` and `q` proves $t_1 \simeq t_2$. The point of using the term φ `q -` $t_1\{t_2\}$ at type `T`, instead of the term t_1, is that the φ term erases to $|t_2|$. Note that the types of the terms are not part of the equality type itself, nor does the elimination rule require that the types of the left-hand and right-hand sides are the same to do an elimination.
- `ι x : T. T'`, the dependent intersection type of Kopylov [2003]. This is the type for terms `t` which can be assigned both the type `T` and the type `[t/x]T'`, the substitution instance of `T'` by `t`. In the annotated language, we introduce a value of `ι x : T. T'` by construct `[ t, t' ]`, where `t` has type `T` (algorithmically), `t'` has type `[t/x]T'`, and the erasure `|t|` is definitionally equal to the erasure `|t'|`. There are also annotated constructs `t.1` and `t.2` to select either the `T` or `[t.1/x]T'` view of a term `t` of type `ι x : T. T'`.

It is important to understand that the described constructs are erased before the formal reasoning (e.g. when checking if 2 terms are definitionally equal), according to the erasure rules in Figure 2.

2.2 Curry-Style Typing

There is an intuitive explanation for why zero-cost (i.e. no performance penalty) conversion should be possible between differently indexed data (i.e. `List` and `Vec`) and differently indexed programs (i.e. `appL` and `appV`). In a Curry-style theory, the same underlying untyped term can be typed multiple different ways. Therefore, if it is possible to type a term as both a list and a vector, then there is actually no need to do any conversion at all because the same term can inhabit both types! In a type-annotated (rather than type-assignement) setting, this translates to having 2 distinct terms at two distinct types, whose *erasures* are equal.

Curry-Style Data. As an example of Curry-style data, consider the standard definitions of Church-encoded lists and vectors below:

```
List ◀ ★ → ★ = λ A. ∀ X : ★. X → (A → X → X) → X.
nilL ◀ ∀ A : ★. List A = Λ A,X. λ cN,cC. cN.
consL ◀ ∀ A : ★. A → List A → List A =
  Λ A. λ x,xs. Λ X. λ cN,cC. cC x (xs -X cN cC).

Vec ◀ ★ → Nat → ★ = λ A,n. ∀ X : Nat → ★.
  X zero → (∀ n : Nat. A → X n → X (suc n)) → X n.
nilV ◀ ∀ A : ★. Vec A zero = Λ A,X. λ cN,cC. cN.
```

```
consV ◀ ∀ A : ★. ∀ n : Nat. A → Vec A n → Vec A (suc n) =
  Λ A,n. λ x,xs. Λ X. λ cN,cC. cC -n x (xs -X cN cC).
```

Notice that the only difference between the list constructor terms (`nilL` and `consL`) and vector constructor terms (`nilV` and `consV`) is the number of implicit abstractions (e.g. `Λ n`) and implicit applications (e.g. `-n`). According to the erasure rules of Figure 2, this means that after erasure, `nilL` and `nilV` share the same underlying untyped term (and the same holds for `consL` and `consV`):

```
|nilL| = |nilV| = λ cN,cC. cN
|consL| = |consV| = λ x,xs,cN,cC. cC x (xs cN cC)
```

Curry-Style Programs. As an example of Curry-style programs, consider the standard definitions of the append function for Church-encoded lists and vectors below:

```
appL ◀ ∀ A : ★. List A → List A → List A
  = Λ A. λ xs. xs -(List A → List A)
  (λ ys. ys)
  (Λ xs. λ x,ih,ys. consL -A x (ih ys)).

appV ◀ ∀ A : ★. ∀ n : Nat. Vec A n →
  ∀ m : Nat. Vec A m → Vec A (add n m)
  = Λ A. λ xs. xs -(λ n. ∀ m : Nat. Vec A m → Vec A (add n m))
  (Λ m. λ ys. ys)
  (Λ n. Λ xs,x,ih. Λ m. λ ys. consV A -(add n m) x (ih -m ys)).
```

Like before, `appL` and `appV` differ by implicit abstractions and applications. An additional difference is that `appL` uses `consL` in its second branch, while `appV` uses `consV` in its second branch. Because (as seen above) the erasure `|consL|` is equal to the erasure `|consV|`, it follows that `appL` and `appV` also share the same underlying untyped term:

```
|appL| = |appV| =
  λ xs. xs (λ ys. ys) (λ x,ih,ys,cN,cC. cC x (ih ys cN cC))
```

2.3 Inductive Datatypes

The enriching direction of reuse requires *dependent* function types, which must be proven by induction on their inputs using eliminators. The Church-encoded `List` and `Vec` datatypes of Section 2.2 do not support induction, due to a result by Geuvers [2001]. However, Stump [2018] shows that the dependent intersection (using the ι-type from Figure 1) of an impredicative Church-encoded type with a predicate, representing what it means for the type to be inductive, *does* support induction (or an eliminator):

```
List ◀ ★ → ★ = λ A. ι xs : ListChurch A. ListInductive A xs.

elimList ◀ ∀ A : ★. ∀ P : List A → ★.
  P (nilL -A) →
  (∀ xs : List A. Π x : A. P xs → P (consL -A x xs)) →
  Π xs : List A. P xs
```

Above, `ListChurch` is the renamed definition of `List` from Section 2.2. Intersection-type versions of the constructors `nilL` and `consL` can also be defined. We refer readers interested in the definitions of `nilL`, `consL`, and `elimList` to Stump [2018], as this section only depends on their type-level interface (rather than their term-level implementation). We also assume a corresponding ι-type definition of `Vec` (in terms of `VecChurch`), its constructors (`nilV` and `consV`), and its eliminator (`elimVec`).

An important thing to point out is that `List` is defined as the intersection of the `ListChurch` and `ListInductive` types, and that intersection pairs (i.e. $[t_1, t_2]$) erase to their first components (i.e. t_1 of type `ListChurch`) by Figure 2. Hence, the erased ι-style `nilL` is the same as the erased Church-style `nilL` (and the same holds for both styles of `consL`).

3 THE PROBLEM & MANUAL SOLUTION

Section 2.2 shows how differently indexed data (e.g. `List` and `Vec`) and programs (e.g. `appL` and `appV`) can share the same erased untyped terms in a Curry-style dependent type theory. Now we consider the problem of manually reusing data and programs, in both the forgetful (e.g. `Vec` to `List`, and `appV` to `appL`) and enriching (e.g. `List` to `Vec`, and `appL` to `appV`) directions.

3.1 The Problem: Manual Linear Time Reuse

First, we review how to manually reuse data and programs using linear time conversions, which is already possible in popular dependently typed languages. Then (in Section 3.2), we show how Cedille lets us manually derive zero-cost (or constant time) conversions from the linear time conversions. To aid readability, from now on we omit implicit abstractions (e.g. Λ `A`) and implicit applications (e.g. `-A`).[2]

Linear Time Forgetful Data Reuse. We can convert a vector to a list by iteration:

```
v2l ◀ ∀ A : ★. ∀ n : Nat. Vec A n → List A
  = elimVec nilL (λ x,ih. consL x ih).
```

The conversion above only requires iteration, rather than induction, because the codomain `List A` does not depend on the domain `Vec A n`. If we explicitly supplied the motive (or, predicate) `P` to `elimVec`, it would ignore its argument (i.e. `P = λ xs. List A`).

Linear Time Enriching Data Reuse. We can convert a list to a vector by induction:

```
l2v ◀ ∀ A : ★. Π xs : List A. Vec A (len xs)
  = elimList nilV (λ x,ih. consV x ih).
```

The conversion above requires induction, rather than iteration, because the codomain `Vec A (len xs)` depends on the domain `List A`. If we explicitly supplied the motive `P` to `elimList`, it would depend on its argument (i.e. `P = λ xs. Vec A (len xs)`).

Linear Time Forgetful Program Reuse. After defining the type synonyms `AppL` and `AppV` for the types of list and vector append, respectively, forgetful reuse of vector append to define list append corresponds to writing a function from `AppV` to `AppL`:

```
AppL ◀ ★ = ∀ A : ★. List A → List A → List A.
AppV ◀ ★ = ∀ A : ★. ∀ n : Nat. Vec A n →
  ∀ m : Nat. Vec A m → Vec A (add n m).

appV2appL ◀ AppV → AppL
  = λ appV,xs,ys. v2l (appV (l2v xs) (l2v ys)).
```

The function `appV2appL` first reuses vector append (`appV`) by applying `appV` to the result of translating both list arguments (`xs` and `ys`) to vectors (via `v2l`). Then, it translates the result of `appV` from a vector to a list (via `v2l`).

[2] While we omit most implicit abstractions and applications in this paper, the current implementation of Cedille only supports a limited form of type inference. Our accompanying Cedille formalization does not omit any implicits.

Linear Time Enriching Program Reuse. Enriching reuse of list append to define vector append is the difficult direction, which requires proving a lemma stating that once a vector has been converted to a list (via `v2l`, or forgetful data reuse), the length of the output list is equal to (or, preserves) the length index of the input vector:

```
v2lPresLen ◀ ∀ A : ★. ∀ n : Nat. Π xs : Vec A n. n ≃ len (v2l xs)
  = elimVec β (λ x,ih. ρ ih - β).
```

Recall (from Section 2.1) that β is the reflexivity constructor of an equality of type `t ≃ t` for any term `t`, and that ρ is a rewrite primitive that exchanges occurrences of `t` with occurrences of `t'` in the goal, when given evidence that `t ≃ t'`. The proof of `v2lPresLen` is an easy induction, which rewrites by the inductive hypothesis (`ih`) in the cons case of the input vector.

It is not possible to reuse function of type `AppL` to define a function of type `AppV` in general, because the result of the second function has specific index requirements (namely, that the output vector length is the sum of the input vector lengths). Enriching function reuse must be modulo an additional premise, which establishes a relationship between the input and output datatype indices.[3] The premise necessary to define `AppV` in terms of `AppL` requires list length (`len`) to distribute through list append (`appL`):

```
LenDistAppL ◀ AppL → ★ = λ appL. ∀ A : ★. Π xs,ys : List A.
  add (len xs) (len ys) ≃ len (appL xs ys).

appL2appV ◀ Π appL : AppL. LenDistAppL appL → AppV
  = λ appL,q,xs,ys.       // Vec A (add n m)
  ρ v2lPresLen xs -       // Vec A (add (len (v2l xs)) m)
  ρ v2lPresLen ys -       // Vec A (add (len (v2l xs)) (len (v2l ys)))
  ρ q (v2l xs) (v2l ys) - // Vec A (len (appL (v2l xs) (v2l ys)))
  l2v (appL (v2l xs) (v2l ys)).
```

After binding the arguments to `appL2appV`, the initial goal type, and the resulting goal type after each rewrite (using ρ), appears as a comment (i.e. to the right of the syntactic comment delimiter `//` on each line).

Initially, the length of the goal vector is the sum of the lengths of both input vectors `xs` and `ys`. First, we use the previously proven lemma `v2lPresLen` to state our goal in terms of the lists resulting from converting input vectors `xs` and `ys` (via `v2l`). After reusing `appL` applied to both converted list, we would like to convert the result to a vector (via `l2v`) and return it. However, the dependent data reuse function `l2v` returns a vector indexed by the `len` of its input list, but the current goal is stated in terms of a sum (i.e. `add` rather than `len`). Therefore, we must first rewrite the goal using our premise that length distributes through append, so that we may finally return the result of applying `l2v`.

3.2 Manual Solution: Zero-Cost Reuse

Now we derive zero-cost (constant time) data and program conversions from the linear time equivalents of Section 3.1. Linear time reuse (e.g. in Section 3.1) is already possible in conventional Church-style type theories, but zero-cost reuse is additionally possible in Curry-style type theories. This is semantically motivated because a Curry-style term can inhabit multiple types, so conversion is semantically unnecessary (as explained in Section 2.2). The zero-cost (data and program) conversions in this section are all defined in 2 parts:

[3] Enriching data reuse of a list (`List`) as a vector (`Vec`) does not require a premise, but in general enriching data reuse is also modulo a premise. For example, a list can only be enriched to an ordered list (`OList`) modulo a premise that the list is sorted (as discussed in Section 7).

(1) An extensional identity proof about the corresponding linear-time conversion.
(2) The actual zero-cost conversion, defined using φ from Figure 1, the linear-time conversion, and the extensional identity proof.

Zero-Cost Forgetful Data Reuse. First, we prove that the `v2l` conversion is extensionally the identity function:

```
v2lId ◀ ∀ A : ★. ∀ n : Nat. Π xs : Vec A n. v2l xs ≃ xs
  = elimVec β (λ x,ih. ρ ih - β).
```

Next, we use the φ primitive (of Figure 1) to return the vector input `xs` at type `List A`, by appealing to the proof (`v2lId`) that `v2l xs` is equal to `xs`.

```
v2l! ◀ ∀ A : ★. ∀ n : Nat. Vec A n ⟶ List A
  = λ xs. φ (v2lId xs) - (v2l xs) {xs}.
```

The φ expression erases to the term within the braces (`{xs}`) by the erasure rules of Figure 2, hence the erasure `|v2l!|` is the identity function. Thus, `v2l!` converts a vector to a list in constant time, as applying `v2l!` is definitionally equal to applying the identity function in CDLE:

```
|v2l!| = λ xs. xs
```

By convention, we suffix a conversion function with a bang (`!`) to denote its zero-cost equivalent.

Zero-Cost Enriching Data Reuse. The enriching direction of zero-cost data reuse follows the same pattern as the forgetful direction, by first proving an extensional identity (`l2vId`), and then using it to define a zero-cost version (`l2v!`) via φ:

```
l2vId ◀ ∀ A : ★. Π xs : List A. l2v xs ≃ xs
   = elimVec β (λ x,ih. ρ ih - β).

l2v! ◀ ∀ A : ★. Π xs : List A. Vec A (len xs)
  = λ xs. φ (l2vId xs) - (l2v xs) {xs}.
```

And similarly, `l2v!` converts any list (`xs`) to a vector at zero-cost:

```
|l2v!| = (λ xs. xs)
```

Zero-Cost Forgetful Program Reuse. For zero-cost forgetful program reuse of vector append, we prove the following extensional identity: Applying the conversion `appV2appL` to any implementation of vector append (`f`), *and* both list arguments, is equal to applying vector append (`f`) to both list argument that have been zero-cost converted to vectors (via `l2v!`).

```
appV2appLId ◀ Π f : AppV. ∀ A : ★. Π xs,ys : List A.
  appV2appL f xs ys ≃ f (l2v! xs) (l2v! ys)
  = λ f,xs,ys.          // v2l (f (l2v xs) (l2v ys)) ≃ f xs ys
  ρ (l2vId xs) -        // v2l (f xs (l2v ys))
  ρ (l2vId ys) -        // v2l (f xs ys)
  ρ (v2lId (f xs ys)) - // f xs ys ≃ f xs ys
  β
```

The right-side of the equality in the goal begins with `f xs ys`, because the zero-cost conversions `l2v! xs` and `l2v! ys` definitionally reduce to `xs` and `ys`, respectively. We rewrite twice (for `xs` and `ys`) by the extensional identity lemma for `l2v` (using `l2vId`). Then, we rewrite once (for `f xs ys`) by the extensional identity lemma for `v2l` (using `v2lId`), after which our goal is solvable by reflexivity (β).

We define the zero-cost conversion `appV2appL!` using φ and the identity lemma `appV2appLId` applied to the vector append argument (`f`) and both list arguments (`xs` and `ys`):

```
appV2appL! ◀ AppV ⟶ AppL
  = λ f,xs,ys. φ (appV2appLId f xs ys) -
  (appV2appL f xs ys) {f (l2v! xs) (l2v! ys)}.
```

The erased zero-cost conversion appV2appL! also definitionally reduces to the identity function:

```
|appV2appL!| = λ f. λ xs,ys. |f (l2v! xs) (l2v! ys)|
  = λ f. λ xs,ys. f xs ys
  = λ f. f
```

The l2v! zero-cost conversions reduce to applications of the identity function. Then, the body of the λ f abstraction η-contracts to f, such that the entire expression reduces to the identity function.

Zero-Cost Enriching Program Reuse. The zero-cost enriching program reuse of list append requires the following extensional identity: Applying the conversion appL2appV to any implementation of list append (f), a proof of the length distributivity premise (p), and both vector arguments, is equal to applying list append (f) to both vector argument that have been zero-cost converted to lists (via v2l!).

```
appL2appVId ◀ Π f : AppV. Π q : LenDistAppL f.
  ∀ A : ★. ∀ n,m : Nat. Π xs : Vec A n. Π ys : Vec A m.
  appL2appV f q xs ys ≃ f (v2l! xs) (v2l! ys)
  = λ f,q,xs,ys.        // l2v (f (v2l xs) (v2l ys)) ≃ f xs ys
  ρ (v2lId xs) -        // l2v (f xs (v2l ys)) ≃ f xs ys
  ρ (v2lId ys) -        // l2v (f xs ys) ≃ f xs ys
  ρ (l2vId (f xs ys)) - // f xs ys ≃ f xs ys
  β
```

Once again, the zero-cost conversion (appL2appV!) is defined in terms of the linear time conversion (appL2appV), φ, and the extensional identity (appL2appVId):

```
appL2appV! ◀ Π f : AppL. LenDistAppL f ⇒ AppV
  = λ f. Λ q. λ xs,ys.
  φ (appL2appVId f q xs ys) -
  (appL2appV f q xs ys) {f (v2l! xs) (v2l! ys)}.
```

The implication (⇒) to the right of the premise (LenDistAppL appL) of appL2appV is syntax for a non-dependent implicit (or, erased) product (i.e. a ∀ with no dependency on the quantified variable). The fact that the zero-cost conversion appL2appV! uses an erased premise (compared to the non-erased premise in the linear time conversion appL2appV) is *crucial*, allowing appL2appV! to also erase to the identity function:

```
|appL2appV!| = (λ f. λ xs,ys. |f (v2l! xs) (v2l! ys)|)
  = λ f. λ xs,ys. f xs ys
  = λ f. f
```

The implicit abstraction Λ q is discarded by erasure, allowing the erasure |appL2appV!| to η-contract to the identity function (similar to how |appV2appL!| reduces).

4 GENERIC PROGRAM & PROOF REUSE

Section 3.2 gives a zero-cost solution to the problem of linear time data and program reuse problem presented in Section 3.1. However, the reused definitions in Section 3.2 are *manually* derived. Beginning with this section, and for the remainder of this paper, we solve the problem of zero-cost reuse *generically*.

In Section 4.1 we review the type of dependent identity functions (`IdDep`), which captures a pattern appearing in the manual zero-cost solution to reuse (Section 3.2). The `IdDep` type is the dependent generalization of the non-dependent `Id` type introduced by Firsov et al. [2018]. Section 4.2 generically solves the problem of *forgetful* program and proof reuse, which corresponds to defining `IdDep`-closed combinators for the type of non-dependent functions (for program reuse), and the type of dependent functions (for proof reuse). Section 4.3 defines 2 additional combinators to generically solve the problem of *enriching* program and proof reuse.

4.1 Type of Dependent Identity Functions

As explained in Section 2.2, an (erased) term may have several possible types in a Curry-style theory. Of particular importance to our work is that the identity function, represented by the untyped lambda term $(\lambda\, x.\, x)$, can have many possible types. We have seen several examples of this in Section 3.1, where the zero-cost conversions `v2l!`, `l2v!`, `appV2appL!`, and `appL2appV!` all erase to the identity function. Thus, it makes sense to define a *type of dependent identity functions* for any domain $A : \star$ and codomain $B : A \to \star$. We informally denote the type of dependent identity functions by $(a : A) \leq B\, a$. Inhabitance of the type $(a : A) \leq B\, a$ represents the existence of a term $\mathcal{F}$, such that $|\mathcal{F}| = (\lambda\, x.\, x)$, and the existence of a typing derivation for the judgement $\Gamma \vdash \mathcal{F} : \Pi a : A.\, B\, a$.

Section 3.1 manually defines zero-cost conversions using a proof that the linear time conversion is (after erasure) an identity operation. Hence, the zero-cost conversion depends on 2 parts:

(1) The linear time conversion.
(2) A proof that the linear time conversion is extensionally an identity function.

Now we formally derive the type of dependent identity functions $(a : A) \leq B\, a$ in Cedille as `IdDep A B`, which abstractly represents both zero-cost conversion parts as a dependent function (Π) returning a dependent pair (`Sigma`):[4]

```
IdDep ◀ Π A : ★. Π B : A → ★. ★
  = λ A,B. Π a : A. Sigma (B a) (λ b. b ≃ a).
```

The type `IdDep A B` is defined when `A` is a type and `B` is a family of types indexed by `A`. Inhabitants of `IdDep A B` take elements of `(a : A)` to elements of `(b : B a)`, and a proof that `b` is propositionally equal to `a` (using the heterogeneous equality type ≃ from Figure 1). We can represent the 2 parts more explicitly by deriving an introduction rule that takes the (conversion) function `f` and its extensional identity proof as arguments:

```
intrIdDep ◀ ∀ A : ★. ∀ B : A → ★.
  Π f : (Π a : A. B a). (Π a : A. f a ≃ a) → IdDep A B
  =  λ f,q,a. pair (f a) (q a).
```

In practice, it is more convenient to introduce elements of `IdDep` directly in terms of the underlying ΠΣ representation, rather than using `intrIdDep`.

Now we define the crucial elimination rule `elimIdDep`, which exposes the witness $\mathcal{F}$ at type `Π a : A. B a`, whose erasure is the identity function:

```
elimIdDep ◀ ∀ A : ★. ∀ B : A → ★. IdDep A B → Π a : A. B a
  = λ c,a. φ (proj2 (c a)) - (proj1 (c a)) {a}.
```

The elimination rule `elimIdDep` uses φ to return the input `a`, originally at type `A`, at type `(B a)` using the extensional identity proof `(proj2 (c a))`, where `c : IdDep A B`. From the erasure rules

[4] The dependent pair type `Sigma` can be derived in Cedille just like the inductive `List` and `Vec` types, as explained in Section 2.3.

of CDLE (in Figure 2), it follows that for any dependent identity function `c` of type `IdDep A B`, `|elimIdDep c| = |ℱ| = (λ a. a)`.

Finally, notice how the definition of `elimIdDep` abstracts out a part (i.e. the use of φ and the extensional identity proof) of the zero-cost conversion definitions (`v2l!`, `l2v!`, `appV2appL!`, and `appL2appV!`) from Section 3.2. In subsequent sections we define `IdDep`-closed combinators, taking `IdDep` inputs and producing an `IdDep` output. Because the combinators always return an `IdDep`, well typed combinator definitions guarantee the existence of zero-cost conversions (whose witness we can always produce by applying `elimIdDep`).

We will also use non-dependent identity function counterparts `Id`, `intrId`, and `elimId` (of `IdDep`, `intrIdDep`, and `elimIdDep`, respectively), where `A : ★`, but also `B : ★` (rather than `B : A → ★`). These are trivially derivable from the dependent versions, so we omit their definitions. Note that our derived non-dependent `Id` type is isomorphic to the `Id` typed introduced by Firsov et al. [2018].

Recall our informal notation of type `IdDep A B` as $(a : A) \leq B\ a$. The informal notation is inspired by Miquel [2001], who uses a non-dependent version of this notation ($A \leq B$) for a subtyping judgement derivable in a Curry-style theory with implicit products. Indeed, our `Id A B` is inhabited when `A` is a subtype of `B`, and correspondingly all of our combinators can also be understood as internalized subtyping inference rules (we discuss the relationship with subtyping further in our related work, Section 6.1). When there is an identity function from `A` to `B` (i.e. `Id A B`), and both `A` and `B` be are functions, it becomes confusing to talk about domains and codomains (e.g. "domain" could refer to the identity function domain `A`, or the domain of the non-identity function `A`, or the domain of the non-identity function `B`). To avoid confusion, and inspired by the relationship with subtyping, we refer to the domain of an identity function as the *subtype* and the codomain as the *supertype* (thus, we can non-ambiguously refer to the domain and codomain of the subtype, and the same for the supertype).

4.2 Forgetful Reuse

Now we define generic solutions to the problem of forgetful program and proof reuse as `IdDep`-closed combinators for the non-dependent and dependent function types, respectively. As a demonstration of using our generic solution, we redo the `appV` reuse example from Section 3.2 in terms of our combinators (we also provide an additional example of reusing the proof of vector append associativity).

The examples in this section assume the existence of identity functions (i.e. values of type `IdDep`) to convert between lists and vectors:

```
v2l ◀ ∀ A : ★. ∀ n : Nat. Id (Vec A n) (List A)
l2v ◀ ∀ A : ★. IdDep (List A) (λ xs. Vec A (len xs))
```

We delay the task of defining `v2l` and `l2v` to Section 5, where we define both identity functions as examples of using our generic data reuse combinators.

4.2.1 Program Reuse Combinator. All the names of our combinators are short descriptions of their return types. For example, below, `allArr2arr` has return type `Id (∀ i : I. X i → X' i) (Y → Y')`. Mnemonically, `allArr2arr` returns an identity function from `allArr` (∀ →) to (2) `arr` (→). We define `allArr2arr` as:

```
allArr2arr ◀ ∀ I : ★. ∀ X : I → ★. ∀ X' : I → ★. ∀ Y : ★. ∀ Y' : ★.
  Π r : Y → I.
  Π c1 : IdDep Y (λ y. X (r y)).
  Π c2 : ∀ i : I. Id (X' i) Y'.
  Id (∀ i : I. X i → X' i) (Y → Y')
```

```
  = λ r,c1,c2,f. pair (λ y. elimId
    (c2 -(r y))
    (f -(r y) (elimIdDep c1 y))
  ) β.
```

The combinator `allArr2arr` is a generic solution to forgetful *non-dependent function* reuse (or, forgetful *program* reuse). For example, it can solve a problem like the one below, where black boxes (■) represent arbitrary (not necessarily the same) types:

```
Id (∀ n : Nat. Vec A n → ■) (List A → ■)
```

The domain of the subtype is an indexed type and the domain of the supertype is a non-indexed type. For example, if we were to solve the problem above with `allArr2arr`, we would set index type `I` to `Nat`, the type family `X` to `Vec A`, and the non-indexed type `Y` to `List A`. The codomains of the subtype and supertype (i.e. the black boxes, or `X'` and `Y'`, respectively) cannot depend on the explicit domain arguments (i.e. `X` and `Y`), which is why we say that `allArr2arr` solves the problem of non-dependent function reuse. However, the codomain of the subtype (`X'`) can depend on the implicit index argument (of type `I`). This covers all the implicit arguments of `allArr2arr`, and now we explain the explicit arguments:

- The argument `r` is the *refinement function*, computing an index of type `I` from the non-indexed type `Y`, e.g. `len : List A → Nat`.
- The argument `c1` is the *contravariant dependent identity function between domains*. It enriches the non-indexed supertype domain `y : Y`, e.g. `xs : List A`. to the indexed subtype domain `X (r y)`, e.g. `Vec A (len xs)`. The index is the refinement of the non-indexed input `y`, e.g. `(len xs)`).
- The argument `c2` is the *covariant non-dependent identity function between codomains*. It forgets the indexed subtype codomain `X' i` as the non-indexed supertype codomain `Y'`.

Notice that `c2` is parameterized by the index type `i : I`, e.g. `Nat`, rather than the non-indexed type `y : Y`, e.g. `List A`. In the implementation of `allArr2arr`, we generate the type index `I` by automatically applying the refinement function (`r`) to a value of the non-indexed type (`y : Y`), making our combinator easier to use.

4.2.2 Program Reuse Example. Now we demonstrate zero-cost forgetful program reuse of vector append to define list append, in terms of `allArr2arr`. We produce an identity function (`Id`) from `AppV` to `AppL`, called `appV2appL` below:

```
appV2appL ◀ Id AppV AppL
  = // Id (∀ A : ★. ■) (∀ A : ★. ■)
  copyType (Λ A.
  // Id (∀ n : Nat. Vec A n → ■) (List A → ■)
  allArr2arr (len -A) (l2v -A) (Λ n.
  // Id (∀ m : Nat. Vec A n → ■) (List A → ■)
  allArr2arr (len -A) (l2v -A) (Λ m.
  // Id (Vec A (add n m)) (List A)
  v2l -A -(add n m)
  ))).
```

Our example includes goal types in comments, where each goal above illustrates the part of the problem solved by a combinator application below. The black boxes (■) hide the parts of the goals that are not relevant to what is being solved by the combinators below. We begin by handling the impredicative quantification `∀ A : ★.`, which is present in both `AppV` and `AppL`, using the easy to define auxiliary definition `copyType` from Figure 3. Next, we apply `allArr2arr` twice to handle contravariantly enriching both arguments from lists to vectors. In these applications, `r` is the length function (`len`), and `c1` is the enriching data reuse function `l2v`. Additionally, `c2` becomes the remainder of the `appV2appL` definition, giving us access to index arguments `n` and `m`. Finally, we covariantly forget the return type from a list to a vector using the forgetful data reuse function `v2l`.

Note that `appV2appL`, above, simultaneously captures the linear time conversion function and the extensional identity proof from Section 3.2 (i.e. the former `appV2appL` and `appV2appLId`). We can recover the actual zero-cost conversion by applying `elimId` to our identity function:

```
appV2appL! ◀ AppV → AppL = elimId appV2appL.
```

Previously, we used a bang (`!`) suffix as a *syntactic* convention for defining a zero-cost conversion. Now, we can also think of the elimination rule of identity functions (`elimId`) as a bang *operator*, because applying it to any `Id` results in a zero-cost conversion. From now on we omit defining the actual zero-cost conversions (like `appV2appL!`), because they can always be recovered by applying the elimination rule for the type of identity functions.

4.2.3 Proof Reuse Combinator. The combinator `allPi2pi` is a generic solution to forgetful *dependent function* reuse (or, forgetful *proof* reuse). For example, it can solve a problem like the one below:

```
Id (∀ n : Nat. Π xs : Vec A n. ■) (Π xs : List A. ■)
```

The subtype codomain may depend on subtype (vector) domain, and the supertype codomain may depend on the supertype (list) domain. The definition of `allPi2pi` follows:

```
allPi2pi ◀ ∀ I : ★. ∀ X : I → ★. ∀ X' : Π i : I. X i → ★.
  ∀ Y : ★. ∀ Y' : Y → ★. Π r : Y → I.
  Π c1 : IdDep Y (λ y. X (r y)).
  Π c2 : ∀ i : I. Id (X i) Y.
  Π c3 : ∀ i : I. Π x : X i. Id (X' i x) (Y' (elimId (c2 -i) x)).
  Id (∀ i : I. Π x : X i. X' i x) (Π y : Y. Y' y)
  = λ r,c1,c2,c3,f. pair (λ y. elimId
    (c3 -(r y) (elimIdDep c1 y))
    (f -(r y) (elimIdDep c1 y))
  ) β.
```

Compared to `allArr2arr`, the `I`, `X`, `Y`, `r`, and `c1` arguments are the same. However, now the subtype codomain `X'` may depend on its indexed domain `X i`, and the supertype codomain `Y'` may depend on its non-indexed domain `Y`. Now we explain the remaining explicit arguments to `allPi2pi`:

- The argument `c2` is the *covariant dependent identity function between domains*. It forgets the indexed subtype domain `X i`, e.g. `Vec A n`, as the non-indexed subtype domain `Y`, e.g. `List A`. We use this additional covariant function between domains (compared to the contravariant version `c1`) in the type of the `c3` argument.
- The argument `c3` is the *covariant non-dependent identity function between codomains*. It forgets the indexed subtype codomain `X' i x` as the non-indexed supertype codomain `Y' (elimId (c2 -i) x)`. The `Y` index to `Y'` is zero-cost converted from `x` using the extra covariant argument `c2`.

The need for the conversion above, and hence the additional covariant argument `c2`, stems from the convenience of argument `c3` being parameterized by index `I` and indexed type `X i`, rather than non-indexed type `Y`. In other words, it is needed for argument `c3` to assume that its parameter has already been zero-cost converted via `elimIdDep c1 y` (as performed in the implementation of allPi2pi). Recall that allArr2arr already automated the refinement r y, and now allPi2pi additionally automates the zero-cost conversion elimIdDep c1 y, because the codomains of allPi2pi can depend on the domains.

4.2.4 Proof Reuse Example. As an example of zero-cost proof reuse, we demonstrate how to prove associativity of list append from the associativity of vector append.[5] First, we create type synonyms for the theorem of list append associativity (AssocL) and vector append associativity (AssocV), parameterized by a definition of list append (AppL) and vector append (AppV), respectively:

```
AssocL ◀ AppL → ★ = λ appL.
  ∀ A : ★. Π xs,ys,zs : List A.
  appL (appL xs ys) zs ≃ appL xs (appL ys zs)).

AssocV ◀ AppV → ★ = λ appV.
  ∀ A : ★. ∀ n : Nat. Π xs : Vec A n.
  ∀ m : Nat. Π ys : Vec A m. ∀ o : Nat. Π zs : Vec A o.
  appV (appV xs ys) zs ≃ appV xs (appV ys zs)).
```

Next, we reuse any proof of AssocV to prove AssocL at zero-cost:

```
assocV2assocL ◀ ∀ appV : AppV.
  Id (AssocV appV) (AssocL (elimId appV2appL appV))
  // Id (∀ A : ★. ■) (∀ A : ★. ■)
  = copyType (Λ A.
  // Id (∀ n : Nat. Π xs : Vec A n. ■) (Π xs : List A. ■)
  allPi2pi (len -A) (l2v -A) (v2l -A) (Λ n. λ xs.
  // Id (∀ m : Nat. Π ys : Vec A m. ■) (Π ys : List A. ■)
  allPi2pi (len -A) (l2v -A) (v2l -A) (Λ m. λ ys.
  // Id (∀ o : Nat. Π zs : Vec A o. ■) (Π zs : List A. ■)
  allPi2pi (len -A) (l2v -A) (v2l -A) (Λ o. λ zs.
  // Id (appV (appV xs ys) zs ≃ appV xs (appV ys zs)))
  //    (appV (appV xs ys) zs ≃ appV xs (appV ys zs)))
  id
  )))).
```

Notice that the identity function assocV2assocL is parameterized by any implementation (appV) of the type of vector append (AppV). We apply the type synonym for vector append associativity (AssocV) directly to vector append (appV), but the type synonym for list append associativity (AssocL) expects an implementation of list append (i.e. a value of type AppL). Hence, we apply AssocL to the result of zero-cost converting appV to a list append, via elimId appV2appL appV, which uses our previously defined identity function appV2appL.

Once again, we begin solving assocV2assocL by copying the type parameter A via copyType. Next, we apply allPi2pi to handle the 3 primary arguments to the theorem. Compared to allArr2arr, allPi2pi receives the additional covariant argument c2, e.g. (λ n. v2l -A -n). Also, the final c3 argument gets to abstract over the indexed type, in addition to the index, e.g. λ n,xs. The final goal is solvable by the auxiliary identity combinator for identity functions (id from Figure 3). Before erasure, the supertype of the final goal has instances of elimId appV2appL appV, instead

[5] The concept of proof reuse being "zero-cost" may seem odd, as systems like Coq erase proofs during program extraction. However, proofs in intentional type theory may sometimes have computational content we wish to preserve, and hence it can be valuable to zero-cost reuse such proofs.

of `appV`. Similarly, before erasure, the final goal has instances of `elimId v2l xs`, instead of `xs` (and the same for `ys` and `zs`). However, because these are zero-cost conversions, after erasure (as depicted in the comment above) the goal is simply solvable by `id`.

4.3 Enriching Reuse

Now we generically solve enriching program and proof reuse as `IdDep`-closed combinators for the non-dependent and dependent function types, respectively. Each forgetful program and proof reuse (Section 4.2) combinator returns a *non-dependent* identity function (`Id`). In contrast, each enriching version returns a *dependent* identity function (`IdDep`), where the dependency is used to define the *premise* necessary for enrichment. We demonstrate our enriching combinators by redoing the `appL` enriching program reuse example from Section 3.2.

4.3.1 Program Reuse Combinator. The combinator `arr2allArrP` is a generic solution to *enriching* non-dependent function reuse (or, enriching *program* reuse). Recall (from Section 3.2) that (in general) enriching program reuse must be performed modulo a premise required for the enrichment to be possible. For example, `arr2allArrP` can solve a problem like the one below:

```
IdDep (List A → ■) (λ f. (Π xs : List A. ■) ⇒ ∀ n : Nat. Vec A n → ■)
```

Program enrichment returns a dependent identity function (`IdDep`). An additional implicit (erased) *premise* argument (to the left of ⇒) appears in the supertype, and the premise has a dependent domain whose type is equal to the subtype's domain (e.g. `List A`). The codomain of the premise can depend on the subtype (e.g. `f`), in addition to the domain of the premise (`xs`). The definition of `arr2allArrP` follows:

```
arr2allArrP ◀ ∀ Y : ★. ∀ Y' : ★. ∀ P : Y → Y' → ★.
  ∀ I : ★. ∀ X : I → ★. ∀ X' : I → ★. Π r : Y → I.
  Π c1 : ∀ i : I. Id (X i) Y.
  Π c1' : ∀ i : I. Π x : X i. i ≃ r (elimId (c1 i) x).
  Π c2 : Π y : Y. IdDep Y' (λ y'. P y y' ⇒ X' (r y)).
  IdDep (Y → Y') (λ f. (Π y : Y. P y (f y)) ⇒ ∀ i : I. X i → X' i)
  = λ r,c1,c1',c2,f. pair (Λ p,i. λ x. elimIdDep
    (ρ (c1' -i x) - (c2 (elimId (c1 -i) x)))
    (f (elimId (c1 -i) x))
    -(p (elimId (c1 -i) x))
  ) β.
```

Enriching `arr2allArrP` shares the following implicit arguments with forgetful `allArr2arr` (from Section 4.2): `Y`, `Y'`, `I`, `X`, and `X'`, as well as the following explicit arguments: `r` and `c1`. However, the premise `P` appears as an additional implicit argument, which may depend on both the domain (`Y`) and codomain (`Y'`) of the subtype. Now we explain the differing explicit arguments:

- The argument `c1'` is the *index preservation property*. It requires the index `i` of the supertype domain (`x : X i`) to equal the refinement (using refinement function `r`) of the zero-cost converting `x` (using the identity function `c1`), e.g. `n ≃ len (elimId c1 xs)`, where `xs : Vec A n`.
- The argument `c2` is the *covariant dependent identity function between codomains*. It enriches the non-indexed subtype codomain `Y'` as the indexed supertype codomain `X' i`. The enrichment codomain also gets an additional implicit premise argument `P`, which may depend on both the subtype domain and codomain.

Notice that `c2` is parameterized by the non-indexed type `y : Y`, e.g. `List A`, rather than the index `i : I`, e.g. `Nat`. This makes `arr2allArrP` easier to use, as the implementation automatically rewrites by `c1'` (the index preservation property)! We point out the consequence of this automatic rewrite in the following example.

4.3.2 Program Reuse Example. Below, we redo the enriching reuse of list append example from Section 3.2. While our forgetful function type combinators attack 2 pieces at a time (the domains of the supertype and subtype), the enriching function type combinators attack 3 (the additional piece being the premise, whose type is duplicated from the subtype domain).

```
appL2appV ◀ IdDep AppL (λ appL. LenDistAppL appL ⇒ AppV)
  // IdDep (∀ A : ★. ■) (λ x. (∀ A : ★. ■) ⇒ ∀ A : ★. ■)
  = copyTypeP (Λ A.
  // IdDep (List A → ■) (λ f. (Π xs : List A. ■) ⇒ ∀ n : Nat. Vec A n → ■)
  arr2allArrP (len -A) (v2l -A) (v2lPresLen -A) (λ xs.
  // IdDep (List A → ■) (λ g. (Π ys : List A. ■) ⇒ ∀ m : Nat. Vec A m → ■)
  arr2allArrP (len -A) (v2l -A) (v2lPresLen -A) (λ ys.
  // IdDep (List A) (λ zs. len zs ≃ add (len xs) (len ys)
  //                        ⇒ Vec A (add (len xs) (len ys)))
  subst (l2v -A)
  ))).
```

We begin with the auxiliary `copyTypeP` combinator (from Figure 3), which is a version of `copyTypeP` that also handles the premise (i.e. the 3rd piece). Next, we use our enriching combinator `arr2allArrP` to handle both inductive arguments of append. This leaves leaves us with the goal in the final comment, above. Finally, we discharge the premise by rewriting, via the auxiliary combinator `subst` (from Figure 3). As an argument, `subst` takes the identity function to apply after rewriting, which is the enriching data reuse `l2v` in this example.

The final goal type includes the sum of the lengths of both input vectors, rather than the sum of two vector indices. This convenience is a result of the automatic rewriting performed in the implementation of our `arr2allArrP` combinator! In contrast, the manual definition of `appL2appV` in Section 3.2 needed to manually rewrite by `v2lPresLen` for both append inputs.

4.3.3 Proof Reuse Combinator. We include the definition of the enriching proof reuse combinator (`pi2allPiP`), for reference, below. We do not describe it in detail, as the extensions to handle the dependent arguments in codomains `X'` and `Y'`, as well as the additional `c1'` through `c3` arguments, follow the same pattern as `allPi2pi` from Section 4.2.

```
pi2allPiP ◀ ∀ Y : ★. ∀ Y' : Y → ★. ∀ P : Π y : Y. Y' y → ★.
  ∀ I : ★. ∀ X : I → ★. ∀ X' : Π i : I. X i → ★. Π r : Y → I.
  Π c1 : ∀ i : I. Id (X i) Y.
  Π c1' : ∀ i : I. Π x : X i. i ≃ r x.
  Π c2 : IdDep Y (λ y. X (r y)).
  Π c3 : Π y : Y. IdDep (Y' y) (λ y'. P y y' ⇒ X' (r y) (elimIdDep c2 y)).
  IdDep (Π y : Y. Y' y)
    (λ f. (Π y : Y. P y (f y)) ⇒ ∀ i : I. Π x : X i. X' i x)
  = λ r,c1,c1',c2,c3,f. pair (Λ p,i. λ x. elimIdDep
    (ρ (c1' -i x) - (c3 (elimId (c1 -i) x)))
    (f (elimId (c1 -i) x))
    -(p (elimId (c1 -i) x))
  ) β.
```

```
id ◀ ∀ A : ★. Id A A = λ a. pair a β.

copyType ◀ ∀ F : ★ → ★. ∀ G : ★ → ★.
  (∀ A : ★. Id (F A) (G A)) →
   Id (∀ A : ★. F A) (∀ A : ★. G A)
  = λ c,xs. pair (Λ A. elimId (c -A) (xs -A)) β.

copyTypeP ◀ ∀ F : ★ → ★. ∀ P : Π A : ★. F A → ★. ∀ G : ★ → ★.
  (∀ A : ★. IdDep (F A) (λ xs. P A xs ⇒ G A)) →
  IdDep (∀ A : ★. F A) (λ xs. (∀ A : ★. P A (xs -A)) ⇒ ∀ A : ★. G A)
  = λ c,xs. pair (Λ p,A. elimIdDep (c -A) (xs -A) -(p -A)) β.

subst ◀ ∀ Y : ★. ∀ I : ★. ∀ X : I → ★.
  ∀ r : Y → I. ∀ i : I.
  IdDep Y (λ y. X (r y)) →
  IdDep Y (λ y. r y ≃ i ⇒ X i)
  = λ c,y. pair (Λ q. ρ ς q - elimIdDep c y) β.
```

Fig. 3. Auxiliary identity combinator, combinators to copy a shared impredicative quantification, and combinator to rewrite by an equality constraint.

We also omit the example of enriching proof reuse of list append associativity. It is very similar to the forgetful proof reuse example of vector append associativity, because `v2lPresLen` becomes an additional argument, making the premise the trivial `Unit` type.

5 GENERIC DATA REUSE

In this section we give the *generic* zero-cost solution to the problem of linear time data reuse presented in Section 3.1, and manually solved in Section 3.2. In Section 5.1, we review a type of least fixed points, used to generically encode datatypes. Section 5.3 covers generic forgetful data reuse, and Section 5.3 covers generic enriching data reuse.

5.1 Type of Least Fixed Points

Section 2.3 reviews the work by Stump [2018] to manually derive induction principles for Church-encoded datatypes via intersecting (using ι) with an inductivity predicate. Firsov and Stump [2018] solved the same problem generically, by deriving a least fixed point type for any *functor*, composed of 4 pieces:

(1) An object mapping (`F ◀ ★ → ★`).
(2) An arrow mapping (`fmap ◀ ∀ X,Y : ★. (X → Y) → F X → F Y`).
(3) The identity functor law.
(4) The composition functor law.

Firsov et al. [2018] improved the solution by deriving a least fixed point type that only requires 2 pieces:

(1) A type scheme (`F ◀ ★ → ★`).
(2) An identity mapping (`imap ◀ ∀ X,Y : ★. Id X Y → Id (F X) (F Y)`).

In type theory, the type scheme `F` is the same as the object mapping of the functor. However, the identity mapping (`imap`) is a restriction of the arrow map (`fmap`), which only requires the user to lift an identity function (`Id` from Section 4.1) between 2 types (`X` and `Y`) to an identity

```
nilLF ◀ ∀ A,X : ★. ListF A X
consLF ◀ ∀ A,X : ★. A ➝ X ➝ ListF A X
elimListF ◀ ∀ A,X : ★. ∀ P : ListF A X ➝ ★.
  P nilLF ➝
  (Π x : A. Π xs : X. P (consLF x xs)) ➝
  Π xs : ListF A X. P xs

nilVF ◀ ∀ A : ★. ∀ X : Nat ➝ ★. VecF A X zero
consVF ◀ ∀ A : ★. ∀ X : Nat ➝ ★. ∀ n : Nat. A ➝ X n ➝ VecF A X (suc n)
elimVecF ◀ ∀ A : ★. ∀ X : Nat ➝ ★. ∀ P : Π n : Nat. VecF A X n ➝ ★.
  P zero nilVF ➝
  (∀ n : Nat. Π a : A. Π x : X n. P (suc n) (consVF x xs)) ➝
  ∀ n : Nat. Π xs : VecF A X n. P n xs
```

Fig. 4. Constructors and eliminators for list and vector schemes (`ListF` and `VecF`)

function between the scheme `F` applied to the same 2 types. Deriving a concrete datatype in terms of the generic encoding of Firsov et al. [2018] takes less effort (compared to using the encoding of Firsov and Stump [2018]), because `imap` is less onerous to define, and no laws need to be proved.

Furthermore, the class of datatypes representable by the Firsov et al. [2018] encoding expands to include infinitary types and positive (not merely strictly-positive) types. Firsov et al. [2018] is an "efficient" lambda-encoding (using Mendler-style F-algebras), in the sense that inductive types support a constant time "predecessor" operation (e.g. `pred` for `Nat`, and `tail` for `List`), using only linear space in the encoding. Expert readers may have noticed that the tail `xs : List` (where `List` is Church-encoded) in the cons case of `elimList` from Section 2.3 is erased (i.e. quantified using ∀ rather than Π), hence computations cannot be defined with (unerased) access to the tail of the list. Deriving induction for a concrete `List` type encoded via the work of Firsov et al. [2018], and using Mendler-style F-algebras, solves this problem (allowing unerased quantification over the tail via Π, accessible in constant time).

In this work we generically solve zero-cost *data reuse* by defining combinators for the fixpoint type of Firsov et al. [2018], whose type is:

```
IdMapping ◀  (★ ➝ ★) ➝ ★ = λ F. ∀ X,Y : ★. Id X Y ➝ Id (F X) (F Y).

Fix ◀ Π F : ★ ➝ ★. IdMapping F ➝ ★
```

This work derives the non-indexed fixpoint (`Fix`) in terms of an indexed fixpoint (`IFix`), over indexed schemes and index-preserving identity mappings (`IIdMapping`). The non-indexed fixpoint is the trivial case where the index is the `Unit` type (having the single inhabitant `unit`). Below, we only give the type of the indexed fixpoint `IFix`, and its implementation is a straightforward generalization of the non-indexed version by Firsov et al. [2018]:

```
IIdMapping ◀ Π I : ★. ((I ➝ ★) ➝ (I ➝ ★)) ➝ ★ = λ I,F.
  ∀ X,Y : I ➝ ★. (∀ i : I. Id (X i) (Y i)) ➝
    ∀ i : I. Id (F X i) (F Y i).

IFix ◀ Π I : ★. Π F : (I ➝ ★) ➝ (I ➝ ★).
  Π imap : IIdMapping I F. I ➝ ★
```

5.2 Data Schemes and Identity Mappings

The examples in the remainder of this section will demonstrate how data reuse combinators reduce the problem of defining an identity function between fixpoints, to defining an identity function between schemes. This is a much simpler problem, because schemes are essentially sums-of-products, which do *not* have inductive arguments.

Our later examples will refer to the scheme for lists (`ListF`), and the scheme for vectors (`VecF`), whose Church-encodings appear below:

```
ListF ◀ ★ → ★ → ★ = λ A,X.
  ∀ C : ★. C → (A → X → C) → C.

VecF ◀ ★ → (Nat → ★) → Nat → ★ = λ A,X,n.
  ∀ C : ★.
  (n ≃ zero ⇒ C) →
  (∀ m : Nat. n ≃ suc m ⇒ A → X m → C)
  → C.
```

Importantly, the impredicatively quantified return type (`∀ C : ★`) does not appear in the recursive tail position of the cons case of either scheme. Intead, the non-indexed `X` appears there for `ListF`, while the indexed `X m` appears there for `VecF`.

Another important detail is that the natural number `n` in `VecF` is a parameter, rather than an index, because `C` is merely a type (★), rather than a family (`Nat → ★`).

Finally, notice that the nil and cons cases of `VecF` have additional *implicit* index equality constraint arguments. Because both the natural number (in the cons case of `VecF`) and the equality constraints are implicit arguments, the constructors of `ListF` and `VecF` erase to the same underlying untyped terms. For reference, the type signatures for the `ListF` and `VecF` constructors and eliminators appear in Figure 4. We assume an intersection-type encoding of `ListF` and `VecF`, using the same technique as in Section 2.3, to make it possible to define the eliminators.[6]

Next, we define the identity mappings `imapL` (for `ListF`) and `imapV` (for `VecF`), whose definitions only differ by which eliminator is used:

```
imapL ◀ ∀ A : ★. IdMapping (ListF A) = λ f. elimListF
  (pair nilLF β)
  (λ x,xs. pair (consLF x (elimId f xs) β)).

imapV ◀ ∀ A : ★. IIdMapping Nat (VecF A) = λ f. elimVecF
  (pair nilVF β)
  (λ x,xs. pair (consVF x (elimId f xs) β)).
```

The returned value is `Sigma`-type the codomain of `Id` from Section 4.1, where the first component is the supertype and second component is the equality witness. For both `imapL` and `imapV`, we mostly rebuild the term with constructors. The interesting subterm is the tail argument (`elimId f xs`) of the cons rebuilding (for both `consLF` and `consVF`). In the nil cases, the second component of the pair (constructing `Sigma`) is obviously reflexivity (β) when rebuilding `nilLF` with itself and `nilVF` with itself. However, in the cons cases, the second component is also β. This is because the *identity* function being mapped (`f`) is erased when zero-cost converting (i.e. `|elimId f xs| = xs`). Hence, β is evidence of the cons rebuilding cases because `|consLF x (elimId f xs)| = |consLF| x xs`, and `|consVF x (elimId f xs)| = |consVF| x xs`.

[6] There is no predecessor problem to worry about when deriving schemes with induction principles (or, eliminators), because schemes contain no inductive occurrences.

5.3 Forgetful Reuse

5.3.1 Data Reuse Combinator. The combinator `ifix2fix` is a generic solution to forgetful *fixpoint* reuse (or, forgetful *data* reuse). For example, it can solve a problem like the one below:

```
Id (IFix Nat (VecF A) (imapV A) n) (Fix (ListF A) (imapL A))
```

Above, the subtype is an indexed fixpoint and the supertype is a non-indexed fixpoint (hence, this the forgetful direction of data reuse). The type of `ifix2fix` follows:

```
ifix2fix ◀ ∀ I : ★. ∀ F : (I → ★) → (I → ★). ∀ G : ★ → ★.
  Π imapF : IIdMapping I F.
  Π imapG : IdMapping G.
  Π c : ∀ X : I → ★. ∀ Y : ★.
    (∀ i : I. Id (X i) Y) → ∀ i : I. Id (F X i) (G Y).
  ∀ i : I. Id (IFix I F imapF i) (Fix G imapG)
```

If we were to solve the problem above with `ifix2fix`, we would set index type `I` to `Nat`, the indexed scheme `X` to `VecF A`, and the non-indexed scheme `Y` to `ListF A`. This covers all the implicit arguments of `ifix2fix`, and now we explain the explicit arguments:

- The argument `imapF` is the *index-preserving identity mapping* for the indexed scheme `F`, e.g. `imapV A` for `VecF A`.
- The argument `imapG` is the *identity mapping* for the non-indexed scheme `G`, e.g. `imapL A` for `ListF A`.
- The argument `c` is the *identity algebra*. It forgets the indexed subtype scheme (`F X i`) as the non-indexed supertype scheme (`G Y`), while assuming how to forget the abstract indexed subtype (`X`) as the abstract non-indexed supertype (`Y`).

The type of `ifix2fix` is reminiscent of standard patterns appearing in generic programming using fixpoint encodings of datatypes. If you define a *non-recursive* identity function between schemes, where the “recursive” positions `X i` are abstract, and you have access to an abstract forgetful identity function (from `X i` to `Y`), you are rewarded with a *recursive* identity function between fixpoints of those schemes.

We omit the implementation of `ifix2fix` combinator since the exact details depend on a particular encoding of Mendler-style fixed points. Intuitively, the identity function from `IFix` to `Fix` is developed by using the generic dependent elimination of `IFix` to apply the `c` argument on each inductive level of the value. The premise of `c`, namely (`∀ i : I. Id (X i) Y`), is the inductive hypothesis of the dependent elimination.

5.3.2 Data Reuse Example. Now we demonstrate zero-cost forgetful reuse of vector data as list data. First, we establish type synonyms for the list and vector types, derived generically as the fixpoints of their schemes and identity mappings:

```
List ◀ ★ → ★ = λ A. Fix (ListF A) (imapL A).
Vec ◀ ★ → Nat → ★ = λ A,n. IFix Nat (VecF A) (imapV A) n.
```

Next, we define an identity function (`v2l`) from `Vec A n` to `List A` by applying `ifix2fix` to the identity mappings and an identity algebra. For legibility, we provide the identity algebra (`vf2lf`) as a standalone definition:

```
vf2lf ◀ ∀ A : ★. ∀ X : Nat → ★. ∀ Y :  ★.
  Π c : ∀ n : Nat. Id (X n) Y.
  ∀ n : Nat. Id (VecF A X n) (ListF A Y)
  = λ c. elimVecF
  (pair nilLF β)
```

```
    (λ x,xs. pair (consLF x (elimId c xs) β)).

  v2l ◀ ∀ A : ★. ∀ n : Nat. Id (Vec A n) (List A) =
    ifix2fix imapV imapL vf2lf.
```

The identity algebra `vf2lf` is defined by constructing an identity function, and the construction is very similar to how we defined the identity mappings `imapL` and `imapV` in Section 5.2. This time, the conversion changes the types (by going from indexed scheme `VecF` to scheme `ListF`), but β still suffices as equality in both cases because the constructors of both schemes erase to the same untyped terms. More concretely, `|nilVF| = |nilLF|` and `|consVF x xs| = |consLF x (elimId c xs)|`. In the cons case, `xs` has (abstract vector) type `X n`, but this is zero-cost converted via `c` to (abstract list) type `Y`. Hence, because we know that `|consVF| = |consLF|`, it follows that:

```
  |consVF x xs| = |consVF| x xs = |consLF| x xs = |consLF x (elimId c xs)|
```

5.4 Mendler-Style Algebras

In generic developments using fixpoint-encodings of datatypes, it is common to define non-dependent functions as the fold of an algebra. Our generic enriching data reuse combinator (in Section 5.5) requires an algebra argument (which is folded in the dependent type signature of the combinator). However, because our fixpoint type is defined using a Mendler-style encoding [Firsov et al. 2018], our enriching combinator must take a *Mendler-style algebra*. Below, we give the definition of a Mendler-style algebra (`AlgM`), and we include the more familiar Church-style algebra (`AlgC`) for reference:

```
  AlgC ◀ (★ → ★) → ★ → ★ = λ F,X. F X → X.
  AlgM ◀ (★ → ★) → ★ → ★ = λ F,X. ∀ R : ★. Π rec : R → X. F R → X.
```

Mendler algebras (`AlgM`) exploit parametricity to abstractly hide inductive data via impredicative quantification (`∀ R : ★`). However, a *recursion function* (`Π rec : R → X`) is provided to explicitly make recursive calls on the hidden data. [7]

Below, we give an example of defining the list length function (`len`) as the fold of a Mendler-style length algebra (`lenAlgM`). We also provide the type of the Mendler-style `foldM` function for reference.

```
  lenAlgM ◀ ∀ X : ★. AlgM (ListF X) Nat
    = λ rec. elimListF zero (λ x,xs. suc (rec xs)).
  len ◀ ∀ A : ★. List A → Nat = foldM lenAlgM.

  foldM ◀ ∀ F : ★ → ★. ∀ imap : IdMapping F. ∀ X : ★.
    AlgM F X → Fix F imap → X
```

The length algebra (`lenAlgM`) case-splits (using `elimListF`) on the scheme (`ListF`) of the generically encoded list. The nil case returns `zero`, and the cons case returns the `suc`(essor) of the result of applying the recursion function (`rec`) to the abstract recursive data (`xs : R`). Our example in Section 5.5 uses both the length algebra (`lenAlgM`) and the length function (`len`) defined as its fold.

5.5 Enriching Reuse

The combinator we define in this section (`fix2ifix`) generically solves data enrichment, going from a non-indexed to an indexed type, when the index can be computed as a total function from the non-indexed type (e.g. going from `List` to `Vec` via the total function `len : List A → Nat`).

[7] Mendler-style data hiding and explicit recursion is one of the ingredients used by [Firsov et al. 2018] to define constant-time predecessor functions.

5.5.1 Data Reuse Combinator. Next, we define the combinator `fix2ifix`, which is a generic solution to enriching *fixpoint* reuse (or, enriching *data* reuse). For example, it can solve a problem like the one below:

```
IdDep (Fix (ListF A) (imapL A))
  (λ xs. IFix Nat (VecF A) (imapV A) (foldM lenAlgM xs))
```

Notice that `fix2ifix` must return a *dependent* identity function, because the index of the output vector is computed as the length (`len`) of the input list (`xs`). The type of `fix2ifix` follows:

```
fix2ifix ◀ ∀ I : ★. ∀ F : (I → ★) → (I → ★). ∀ G : ★ → ★.
  Π imapF : IIdMapping I F.
  Π imapG : IdMapping G.
  Π ralg : AlgM G I.
  Π c : ∀ X : I → ★. ∀ Y : ★. Π r : Y → I.
    IdDep Y (λ y. X (r y)) →
    IdDep (G Y) (λ xs. F X (ralg -Y r xs)).
  IdDep (Fix G imapG)
    (λ x. IFix I F imapF (foldM ralg x))
```

Both `fix2ifix` and `ifix2fix` (from Section 5.3) share the same implicit arguments, namely `I`, `F`, and `G`, and they also share the explicit `imapF` and `imapG` arguments. However, `fix2ifix` has the following differing explicit arguments:

- The argument `ralg` is the *refinement algebra* for the non-indexed scheme `G`, e.g. `lenAlgM A` for `ListF A`.
- The argument `c` is the *dependent identity algebra*. It enriches the non-indexed subtype scheme (`xs : G Y`) to the indexed supertype scheme (`F X (ralg r xs)`), while assuming how to enrich the abstract non-indexed subtype (`y : Y`) as the abstract indexed supertype (`X (r y)`).

Similar to the `c` of forgetful `ifix2fix`, the `c` of enriching `fix2ifix` requires a *non-recursive* identity function between schemes, while assuming access to an identity function between abstract "recursive" positions. However, the identity function in `c` for `fix2ifix` are *dependent*. Hence, the index of the assumed supertype (`X`) is computed from the non-indexed subtype (`y : Y`) by applying an abstract *refinement function* (`r`). Correspondingly, the index of the produced supertype (`F X`) is computed from the non-indexed subtype (`xs : G Y`) by applying the *refinement algebra* (`ralg r`), while using `r` for the `rec`(ursive) function of the Mendler-style algebra.

The implementation of `fix2ifix`, just like `ifix2fix`, applies `c` to each inductive level. The outcome is also similar, as `fix2ifix` allows the user to define a *non-recursive* identity algebra, and it produces a *recursive* identity function between fixpoints. The primary difference is that `fix2ifix` results in a dependent identity function. Hence, the index in the dependent result is computed by folding the the Mendler-style algebra (`ralg`) over the inductive input `x`.

5.5.2 Data Reuse Example. Now we demonstrate zero-cost enriching reuse of list data as vector data. The dependent identity function (`l2v`) from `xs : List A` to `Vec A (len xs)` is defined by applying `fix2ifix` to the identity mappings and the dependent identity algebra `lf2vf`:

```
lf2vf ◀ ∀ A : ★. ∀ X : Nat → ★. ∀ Y : ★.
 Π f : Y → Nat.
 IdDep Y (λ y. X (f y)) →
 IdDep (ListF A Y ) (λ xs. VecF A X (lenAlgM f xs))
  = λ c. elimListF
  (pair nilVF β)
  (λ x,xs. pair (consVF x (elimIdDep c xs) β)).
```

```
l2v ◀ ∀ A : ★. IdDep (List A) (λ xs. Vec A (len xs))
  = fix2ifix imapV imapL lenAlgM lf2vf.
```

The Mendler-style algebra used by `lf2vf` is our previously defined length algebra (`lenAlgM`). The definition of `lf2vf` is essentially the same as `vf2lf` from Section 5.3, but now we eliminate a list scheme and produce vector scheme constructors. Because the vector and list scheme constructors erase to the same terms, the argument for why reflexivity (β) suffices as identity evidence stays the same. Another difference is that the abstract tail is computed as a *dependent* elimination (`elimIdDep`, rather than `elimId`). However, the dependent elimination is also erased (`|elimIdDep c xs| = xs`).

6 RELATED WORK

6.1 Subtyping

Miquel [2001] shows that in a Curry-style type theory with implicit products, the subtyping *judgement* can be derived as follows:

$$\Gamma \vdash X \leq Y \triangleq \Gamma, x : X \vdash x : Y$$

The `Id` type can be seen as the internalization of this judgement, with `IdDep` a corresponding dependent version (i.e. our informal syntax $(x : X) \leq Y\ x$, not covered by Miquel). Miquel also showed that the subsumption rule of subtyping is admissible in the theory with the derived judgement, and our elimination rule `elimId` corresponds to its internalization. Finally, all of our combinators can also be translated to admissible subtyping rules in his theory. Miquel covers several admissible subtyping rules, but not ones corresponding to our primary forgetful and enriching combinators for program, proof, and data reuse. Our data reuse combinators may be of particular interest to the subtyping community, as they corresponds to Mendler-style datatype-generic subtyping rules.

Inspired by the internalized subtyping judgement of Miquel, Barras and Bernardo [2008] show how to derive zero-cost forgetful data reuse conversions for Church-encoded datatypes. This work was extended by Diehl and Stump [2018] to the enriching direction. In Section 3.2, we derive zero-cost data reuse in terms of a linear time conversion and its extensional identity proof, using ϕ. In contrast, the zero-cost conversions of Barras and Bernardo [2008] and Diehl and Stump [2018] require no extensional identity proof, as the conversions erase to the identity function by a clever exploitation of η-equality, without needing a rule like ϕ. Our work can be seen as the generic version of their manual zero-cost reuse. When working generically with abstract combinator definitions, an abstraction like `IdDep` is necessary, and hence also a rule like ϕ (used to eliminate it).

6.2 Coercible in Haskell

Breitner et al. describe a GHC extension to Haskell (available starting with GHC 7.8) for a type class `Coercible a b`, which allows casting from `a` to `b` when such a cast is indeed the identity function [Breitner et al. 2016]. The motivation is to support retyping of data defined using Haskell's `newtype` statement, which is designed to give programmers the power to erect abstraction barriers that cannot be crossed outside of the module defining the `newtype`. Within such a module, however, `Coercible a b` and the associated function `coerce : a -> b` allow programmers to apply zero-cost casts to change between a `newtype` and its definition.

Coercible had to be added as primitive to GHC, along with a rather complex system of *roles* specifying how coercibility of application of type constructors follows from coercibility of arguments to those constructors. In contrast, in the present work, we have shown how to derive zero-cost coercions within the existing type theory of Cedille (via IdDep, also derived in Cedille, which is the dependent equivalent of Coercible). On the other hand, much of the complexity of Coercible in GHC arises from (1) how it interoperates with programmer-specified abstraction (via newtype) and (2) the need to resolve Coercible a b class constraints automatically, similarly to other class constraints in Haskell. The present work does not address either issue. However, the present work does allow for dependent casts between indexed variants of datatypes, which Coercible does not cover (because Coercible is only equivalent to our non-dependent Id).

6.3 Dependent Interoperability

The field of dependent interoperability is concerned with reusing code between non-dependent and dependent implementations of datatypes and functions. The goal is to support interaction between non-dependent and dependent languages, like extracted OCaml and Coq. The most similar work to ours in this field is that of Dagand et al. [2016]. Inspired by Homotopy Type Theory (HoTT), Dagand et al. [2016] formalize partial equivalence types, simultaneously representing the forgetful and enriching directions of reuse.

They also develop combinators that are closed with respect to their partial equivalence type. For example, their HODepEquiv combinator is quite like our forgetful program reuse combinator allArr2arr. However, their work primarily focuses on the forgetful direction of reuse for total functions, as partial functions can be reused by inserting dynamic checks and failures using their *partial* equivalence type. In contrast, we emphasize the *total* reuse of functions in the enriching direction (like arr2allArrP), using premises to make the total functions possible. Because they are primarily interested in program reuse, not proof reuse, they do not provide dependent versions of their reuse combinators. Additionally, they only provide combinators function types, not fixpoint types, as their work assumes manual solutions to the problem of data reuse.

The class of datatypes reusable in their setting is larger, because isomorphic datatypes, with different representations, can be related. In contrast, our work requires the erasures of the constructors of related types to be the same untyped terms. However, for the price of a smaller class of reusable types, we gain the ability to perform conversions at zero-cost.

Finally, Dagand et al. [2016] automates the assembly of combinators to reuse programs by registering them as instances of Coq's type class mechanism. Cedille does not currently have type classes, but we could employ the same automation strategy if type classes get added to Cedille in the future.

6.4 Ornaments

Ornaments [McBride 2011] are used to define refined version of types (e.g. Vec) from unrefined types (e.g. List) by "ornamenting" the unrefined type with extra index information. In contrast, our work establishes a relationship between Vec and List after-the-fact, by defining forgetful and enriching IdDep values between the types. By *defining* vectors as natural-number-ornamented lists, ornaments can be used to calculate the "patch" type necessary to adapt a function from one type to another type [Dagand and McBride 2012]. For example, ornaments could calculate that LenDistAppL is the premise necessary to adapt appL from lists to vectors (appV).

Although ornaments can be used to derive conversions between types in an ornamental relationship [Ko and Gibbons 2013; McBride 2011], they take linear time, rather than constant time (i.e. the conversions are not zero-cost). Besides refining the indices of existing datatypes, ornaments also allow data to be added to existing datatypes. For example, vectors can be index-refined lists, but lists can also be natural numbers with elements added. Our work only covers the index refinement aspect of ornaments.

6.5 Type Theory in Color

Type Theory in Color (TTC) [Bernardy and Guilhem 2013] generalizes the concept of erased arguments of types to various colors, which may be erased optionally and independently according to modalities in the type theory. In the vector datatype declaration, the index data can be colored. If a vector is passed to a function expecting a list (whose modality enforces the lack of the index data color), then a forgetful zero-cost conversion (using our parlance) is performed.

Lists can also be used as vectors, via an enriching zero-cost conversion in the other direction. This works due to a mechanism to interpret lists as a predicate on natural numbers. The list predicate is generated as the erasure of its colored elements (like ornaments, colors can add data in addition to refining indices), which results in refining lists by the length *function*.

Our work can be used to define an enriching zero-cost conversion from natural numbers to the datatype of finite sets (`Fin`). This is not possible with colors, because `Fin` is indexed by successor (`suc`) in both of its constructors, which would require generating a predicate on the natural numbers from a non-deterministic function (or *relation*). Colors allow zero-cost conversions to be generated and *implicitly* applied because colors erase *types*, as well as values, whereas implicit products only erase values (e.g. Λ is erased, but not $\forall$). Thus, while zero-cost conversions need to be *explicitly* crafted and applied in our setting, we are able to *define* zero-cost conversions (like taking natural numbers to finite sets) for which there is no unique solution.

7 EXTENSIONS AND FUTURE WORK

7.1 Auxiliary Combinators

Our program and proof reuse combinators expect index arguments to appear next to their indexed types in type signatures. For this reason, our combinators would not be directly applicable if we wrote the type signature of vector append with the natural number indices of both vector arguments at the beginning, followed by both vectors. However, it is straightforward to define an auxiliary combinator that flips argument order, which Dagand et al. [2016] do for their partial type equivalence abstraction.

If a subsequent indexed type argument depends on the *same* index as a previous argument, rather than a new one, it also prevents our combinators from being applicable. Consider the artificial example of vector append where both input vectors must be the same length. This can be solved via a straightforward auxiliary combinator that introduces a new index quantification, along with an equality that constrains the new index to equal the old index.

7.2 Index-Index Combinators

In this paper we only considered relating non-indexed types (e.g. list) to indexed types (e.g. vector). In general, we may want to relate an indexed type to a *less indexed* type, like relating vectors to ordered vectors in the introduction. Our combinators straightforwardly generalize to 2 indexed types, $\mathtt{X} : \mathtt{I} \rightarrow \star$ and $\mathtt{Y} : \mathtt{J} \rightarrow \star$, along with a function that translates the more refined index to the less refined index (of type $\mathtt{I} \rightarrow \mathtt{J}$). In fact, our formalization is defined using this more general representation, and we derive the combinators of this paper using the `Unit` type as the index type `J` of family `Y`.

7.3 Dependent Premise Combinators

The class of datatypes enrichable by our data reuse combinator `fix2ifix` requires the index (`I`) to be computable from the non-indexed type as a *total* function (the refining function `r`). This excludes reusing datatypes where the index can be computed as a *partial* function from the non-indexed type, plus extra data. We believe this would be possible by defining an enhanced data enrichment combinator with an *erased premise*, that the supertype could depend on. We leave this extension to future work, although we have already defined a dependent-premise version of enriching program and proof reuse.

8 CONCLUSION

We have demonstrated how to reuse programs, proofs, and types at zero-cost, in both the forgetful and enriching directions. We achieve this generically via combinators over the type of dependent identity functions (`IdDep`). Because partially applying the elimination rule of `IdDep` results in the term erasing to an identity function, any conversion making use of the result of `elimIdDep` requires no runtime overhead.

REFERENCES

Bruno Barras and Bruno Bernardo. 2008. The implicit calculus of constructions as a programming language with dependent types. *Foundations of Software Science and Computational Structures* (2008), 365–379.

Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-theory in Color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2500365.2500577

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *J. Funct. Program.* 26 (2016), e15.

Pierre-Evariste Dagand and Conor McBride. 2012. Transporting Functions Across Ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/2364527.2364544

Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial Type Equivalences for Verified Dependent Interoperability. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 298–310. https://doi.org/10.1145/2951913.2951933

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Larry Diehl and Aaron Stump. 2018. Zero-Cost Coercions for Program and Proof Reuse. (2018). arXiv:1802.00787

Denis Firsov, Richard Blair, and Aaron Stump. 2018. Efficient Mendler-Style Lambda-Encodings in Cedille. (2018). Under review for the International Conference on Interactive Theorem Proving as of March 16, 2018.

Denis Firsov and Aaron Stump. 2018. Generic Derivation of Induction for Impredicative Encodings in Cedille. In *Certified Programs and Proofs (CPP)*, June Andronick and Amy Felty (Eds.).

Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*. 166–181.

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

Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*. 86–95.

Conor McBride. 2011. Ornamental algebras, algebraic ornaments. (2011).

Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications (TLCA)*, Samson Abramsky (Ed.). 344–359.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology.

Aaron Stump. 2017. The calculus of dependent lambda eliminations. *J. Funct. Program.* 27 (2017), e14.

Aaron Stump. 2018. From Realizability to Induction via Dependent Intersection. *Ann. Pure Appl. Logic* (2018). to appear.

The Coq Development Team. 2008. *The Coq Proof Assistant Reference Manual.* http://coq.inria.fr

J. B. Wells. 1999. Typability and Type Checking in System F are Equivalent and Undecidable. *Ann. Pure Appl. Logic* 98, 1-3 (1999), 111–156.