# Relational Algebraic Ornaments

Hsiang-Shang Ko Jeremy Gibbons
Department of Computer Science, University of Oxford
{Hsiang-Shang.Ko, Jeremy.Gibbons}@cs.ox.ac.uk

## Abstract

Dependently typed programming is hard, because ideally dependently typed programs should share structure with their correctness proofs, but there are very few guidelines on how one can arrive at such integrated programs. McBride's *algebraic ornamentation* provides a methodological advancement, by which the programmer can derive a datatype from a specification involving a fold, such that a program that constructs elements of that datatype would be correct by construction. It is thus an effective method that leads the programmer from a specification to a dependently typed program. We enhance the applicability of this method by generalising algebraic ornamentation to a relational setting and bringing in relational algebraic methods, resulting in a hybrid approach that makes essential use of both dependently typed programming and *relational program derivation*. A dependently typed solution to the minimum coin change problem is presented as a demonstration of this hybrid approach. We also give a theoretically interesting "completeness theorem" of relational algebraic ornaments, which sheds some light on the expressive power of ornaments and inductive families.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

***Keywords*** dependently typed programming; inductive families; program derivation; algebra of programming; greedy algorithms

## 1. Introduction

Dijkstra famously described the situation where a program is written and then proved correct afterwards as "putting the cart before the horse", and argued that correctness proof and program should "grow hand in hand" such that the proof can guide the development of the program [8, 9]. Dependently typed programming is a promising step towards this goal: Traditional functional programming languages are enriched with expressive types capable of expressing strong specifications, and during program development the type system can offer more hints and guarantees. In particular, we are interested in programs that share structure with their correctness proofs, so the programs and their proofs can be devised and completed simultaneously [14]. We proposed to call such an approach *internalism*, suggesting that proofs are internalised in programs rather than being given separately [12].



*DTP '13*, September 24, 2013, Boston, MA, USA.

ACM 978-1-4503-2384-0/13/09...$15.00.
http://dx.doi.org/10.1145/2502409.2502413

In essence, internalism is no more than a refinement of the intuitive way in which we first learned to program, namely relying on the semantic understanding of programs to guide the development, but now aided by a more informative syntax. It is a novel refinement, though — the novelty lies in the idea that a program can carry a proof in its syntax implicitly. We programmers are unfamiliar with this idea, and consequently find it awkward to devise such programs. In this respect, dependently typed programming is indeed harder, not only because there are now proof obligations to be formally discharged, but also because we are still in search of program structures into which more of these proof obligations can be embedded.

To reduce the problem a little: since program structures are deeply influenced by their types (which is even more true for dependently typed programming), we need more help with designing types from which we can receive useful guidance when writing internalist programs. One important type structure we currently have for internalism is *inductive families* [10], which enables programs to carry inductive proofs implicitly. The use of inductive families is usually explained only through ad hoc examples, however, and a methodological treatment was absent until McBride proposed datatype *ornamentation* for organising related inductive families [15]. In particular, McBride proposed *algebraic* ornamentation and an associated construction method: Suppose that we wish to construct a function $f : A \to T$, where $T$ is some inductive type, that satisfies the specification $\mathit{fold}\ \phi \circ f \doteq g$ (where $\_\doteq\_$ denotes extensional equality) for some function $g : A \to B$ and algebra $\phi$. (McBride gave an example in which the function $f$ we wish to construct compiles arithmetic expressions into stack-machine programs. Compilation is correct if, for any arithmetic expression, the result of executing the stack-machine program compiled from that expression — which is done by a fold — matches the semantics of the expression.) Rather than directly implementing $f$ and then proving its correctness, we synthesise from the definitions of $T$ and $\phi$ an inductive family $T^\phi$ indexed by $B$ and construct a function $f^\phi : (a : A) \to T^\phi\ (g\ a)$, from which we can extract $f$ and a proof that it satisfies the specification. The definition of the inductive family $T^\phi$ is derived by algebraic ornamentation and carries the specification in its structure, which can guide the construction of $f^\phi$. This construction method is highly effective because it tells us how to arrive at a useful internalist type from a specification rather than improvising one and hoping that it will help. The specification targeted by the method does not seem too widely applicable, though. Can we solve more complicated specifications with this construction method?

A possible step forward is to consider the slightly more general specification $h \circ f \doteq g$: this specification can be solved with exactly the same construction method above if we can prove that $h \doteq \mathit{fold}\ \phi$. Effective ways of carrying out such proofs — especially when $h$ is written in point-free style — have been studied by the *program derivation* community [4]. Experience with program derivation suggests that we take one step further — to gen-

eralise to relations. With a relational language we can give more powerful yet concise specifications, and can naturally talk about program refinement with relational inclusion. And it seems natural to also generalise algebraic ornamentation to accept relational algebras, in the hope that relational program derivation can be smoothly integrated with dependently typed programming through relational algebraic ornamentation. We would start from a relational specification, manipulate the expression until it becomes a relational fold, and then invoke relational algebraic ornamentation to synthesise a datatype that offers useful type information during program construction and guarantees correctness with respect to the original specification once the program is constructed.

This paper reports on a successful initial experiment with the above hybrid construction method integrating relational program derivation and internalist programming, conducted in Agda [19, 20]. We import a part of the AoPA library [18] — which provides facilities for encoding relational derivations in Agda — to the ornament framework [7, 12, 15] and generalise McBride's algebraic ornamentation to the relational setting (Section 3). (The ornament framework is briefly recapped in Section 2.) Using relational algebraic ornamentation, we construct a program for the well-known greedy algorithm for the *minimum coin change problem* from its relational specification, and correctness of the program is jointly guaranteed by the proof encoded in the program and a variant of the *Greedy Theorem* given by Bird and de Moor [4, Chapter 10] (Section 5). It also turns out that relational algebraic ornamentation is theoretically interesting in its own right: we give a "completeness theorem" stating that *every ornament is a relational algebraic ornament up to isomorphism*, which sheds some light on the expressive power of ornaments and the computational meaning of inductive families (Section 4). We discuss this hybrid approach and the relationship between relational algebraic ornamentation and existing work on ornaments in Section 6.

The paper omits most low-level details of the datatype-generic constructions for clarity. Our Agda code is included in the ACM Digital Library as supplemental material.

## 2. Datatype descriptions and ornaments

In this section we give a high-level introduction to a *universe* [13] for *index-first datatypes* [6] and a language of *ornaments* for relating structurally similar datatypes, presenting only examples rather than datatype-generic definitions. For a full account, see our previous paper [12] and Dagand and McBride's work [7].

***Index-first datatypes and their descriptions.*** The notion of *index-first datatypes* was first introduced by Chapman et al. [6] and a notation for such datatypes was proposed by Dagand and McBride [7]. Below is our own adaptation [12] of Dagand and McBride's notation. For an index-first datatype, rather than listing its constructors along with their complete types like in Agda, we write the targeted types first and determine from the indices of the types which constructors they offer. For simple datatypes, this does not make a huge difference: For example, the following datatype declaration (which is prefixed by the keyword **indexfirst** to be distinguished from Agda datatype declarations) specifies that the type $\mathsf{Nat}$ of natural numbers offers either the $\mathsf{zero}$ constructor or the $\mathsf{suc}$ constructor, the latter requiring an argument of type $\mathsf{Nat}$, which is given a name $n$.

$$\textbf{indexfirst data}\ \mathsf{Nat} : \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Nat} \ni \mathsf{zero} \mid \mathsf{suc}\ (n : \mathsf{Nat})$$

Let $A : \mathsf{Set}$ in what follows.[1] The type $\mathsf{List}\ A$ of lists with elements of type $A$ is declared similarly.

$$\textbf{indexfirst data}\ \mathsf{List}\ A : \mathsf{Set}\ \textbf{where}$$
$$\mathsf{List}\ A \ni [\,]$$
$$\mid\ \_::\_\ (a : A)\ (as : \mathsf{List}\ A)$$

The power of index-first datatypes only shows up in non-trivially indexed datatype declarations, in which we can do pattern matching on the indices. For example, the datatype $\mathsf{Vec}'\ A$ of vectors, i.e., lists indexed by their length, can be declared as follows:

$$\textbf{indexfirst data}\ \mathsf{Vec}'\ A : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Vec}'\ A\ \mathsf{zero} \ni [\,]$$
$$\mathsf{Vec}'\ A\ (\mathsf{suc}\ n) \ni \_::\_\ (a : A)\ (as : \mathsf{Vec}'\ A\ n)$$

$\mathsf{Vec}'\ A$ has an index of type $\mathsf{Nat}$, on which we do pattern matching. When the index is $\mathsf{zero}$, the type $\mathsf{Vec}'\ A\ \mathsf{zero}$ only offers the nil constructor; when the index is $\mathsf{suc}\ n$ for some $n : \mathsf{Nat}$, the type $\mathsf{Vec}'\ A\ (\mathsf{suc}\ n)$ only offers the cons constructor, which takes a head element $a : A$ and a tail vector $as : \mathsf{Vec}'\ A\ n$ as arguments. Note that the representation of a vector is as efficient as that of a list — there is no information that needs to be stored in a cons node other than the head and the tail. In general, with index-first datatypes, we can directly express the *detagging* optimisation of inductive families proposed by Brady et al. [5]. In contrast, Agda's declaration of vectors,

$$\textbf{data}\ \mathsf{Vec}\ A : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{nil}\ \ : \mathsf{Vec}\ A\ \mathsf{zero}$$
$$\mathsf{cons} : (a : A)\ (m : \mathsf{Nat})\ (as : \mathsf{Vec}\ A\ m) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ m)$$

if read literally, means that the cons nodes need to store the length of the tail as well. This less efficient — but more generalisable — declaration of vectors can also be written in index-first style:

$$\textbf{indexfirst data}\ \mathsf{Vec}\ A : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Vec}\ A\ n \ni \mathsf{nil}\ (neq : n \equiv \mathsf{zero})$$
$$\mid\ \mathsf{cons}\ (a : A)\ (m : \mathsf{Nat})$$
$$(as : \mathsf{Vec}\ A\ m)\ (meq : n \equiv \mathsf{suc}\ m)$$

Besides the field $m$ storing the length of the tail, two more fields $neq$ and $meq$ are inserted, demanding explicit equality proofs about the indices. The vector datatype referred to in the rest of the paper will be this more generalisable version (rather than the optimised version $\mathsf{Vec}'$).

Under the bonnet, we assume that there is a universe

$$\mathsf{Desc} : (I : \mathsf{Set}) \to \mathsf{Set}_1$$

for index-first datatypes, which is parametrised by an index set $I : \mathsf{Set}$ of datatypes. Elements of $\mathsf{Desc}\ I$ are called *datatype descriptions*. The datatype declarations of $\mathsf{Nat}$, $\mathsf{List}\ A$, and $\mathsf{Vec}\ A$ given above are all assumed to be high-level presentations of datatype descriptions: $\mathsf{Nat}$ is considered to be trivially indexed by the one-element set $\top$ with sole element $\mathsf{tt}$, so its declaration corresponds to some code $NatD : \mathsf{Desc}\ \top$; $\mathsf{List}$ is also trivially indexed but has a parameter of type $\mathsf{Set}$, so its declaration corresponds to a family of codes $ListD : \mathsf{Set} \to \mathsf{Desc}\ \top$; $\mathsf{Vec}\ A$ is indexed by $\mathsf{Nat}$ and has a parameter of type $\mathsf{Set}$, so its declaration corresponds to some $VecD : \mathsf{Set} \to \mathsf{Desc}\ \mathsf{Nat}$. Datatype descriptions are decoded to actual types by the least fixed-point operator

$$\mu : \{I : \mathsf{Set}\} \to \mathsf{Desc}\ I \to (I \to \mathsf{Set})$$

so $\mathsf{Nat}$, $\mathsf{List}\ A$, and $\mathsf{Vec}\ A\ n$ are, in fact, sugared forms of $\mu\ NatD\ \mathsf{tt}$, $\mu\ (ListD\ A)\ \mathsf{tt}$, and $\mu\ (VecD\ A)\ n$. There are generic fold and induction operators parametrised by descriptions, so we can write datatype-generic programs and proofs that work for all datatypes encoded in the universe. Here we give the type of the generic fold operator: There is an operation on descriptions

$$\mathscr{F} : \{I : \mathsf{Set}\} \to \mathsf{Desc}\ I \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set})$$

[1] We treat quantification in text (like $A : \mathsf{Set}$ here) like introducing a module parameter, which subsequent definitions (like the declaration of $\mathsf{List}$) can refer to directly. To make it clear which parameters are referred to and how they are used, we include the (explicit) parameters as part of the definitions (like the declaration **data** $\mathsf{List}\ A : \mathsf{Set}$), deviating from Agda syntax.

which decodes a description to a base functor. Then the type of the generic fold operator is

$$\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathscr{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X)$$

where $\_\rightrightarrows\_$ is the type of families of functions between corresponding types in two type families:

$$\_\rightrightarrows\_ : \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}$$
$$X \rightrightarrows Y = \forall\ \{i\} \to X\ i \to Y\ i$$

For example, the base functor decoded from *ListD A* is[2]

$$\mathscr{F}\ (\mathit{ListD}\ A) : (\top \to \mathsf{Set}) \to (\top \to \mathsf{Set})$$
$$\mathscr{F}\ (\mathit{ListD}\ A)\ X\ \_ = \Sigma\ \mathsf{LTag}\ (\lambda\{\text{`}\mathsf{nil} \to \top; \text{`}\mathsf{cons} \to A \times X\ \mathsf{tt}\})$$

where LTag is a two-element set for marking the nil and cons cases:

```
data LTag : Set where
  `nil  : LTag
  `cons : LTag
```

and the fold operator on lists would essentially be

$$\begin{aligned}
&\mathit{fold}\ \{\top\}\ \{\mathit{ListD}\ A\} : \{X : \top \to \mathsf{Set}\} \to \\
&\qquad (\mathscr{F}\ (\mathit{ListD}\ A)\ X \rightrightarrows X) \to \\
&\qquad (\mu\ \ (\mathit{ListD}\ A) \quad \rightrightarrows X) \\
&\mathit{fold}\ f\ [\,] \quad\ = f\ (\text{`}\mathsf{nil}, \mathsf{tt}) \\
&\mathit{fold}\ f\ (a :: as) = f\ (\text{`}\mathsf{cons}, a, \mathit{fold}\ f\ as)
\end{aligned}$$

assuming that dependent pattern matching can be performed on the high-level presentations of index-first datatypes.

***Ornaments.*** The three datatypes Nat, List $A$, and Vec $A$ are evidently related: a list is a natural number whose cons nodes are decorated with elements of $A$, and a vector is a list enriched with length information. Such relationship can be seen by "overlaying" one datatype declaration on the other: for example, the declaration of List $A$ differs from that of Nat only in an extra field $(a : A)$ in the cons constructor, and the declaration of Vec $A$ differs from that of List $A$ in that (i) the index set is changed from $\top$ to Nat, (ii) the cons constructor has two extra fields, and (iii) the index of the recursive position is specified to be $m$. Such differences between datatype declarations are encoded as *ornaments*. Whenever there is an ornament between two datatypes, there is a forgetful function from the more informative datatype to the other, erasing information according to the ornament's specification of datatype differences. For example, we have a forgetful function from lists to natural numbers that discards elements associated with cons nodes — i.e., it computes the length of a list — and another one from vectors to lists which removes all length information from a vector and returns the underlying list.

Ornaments constitute the second underlying universe:

$$\mathsf{Orn} : \{I\ J : \mathsf{Set}\}\ (e : J \to I)\ (D : \mathsf{Desc}\ I)\ (E : \mathsf{Desc}\ J) \to \mathsf{Set}_1$$

An ornament $O$ : Orn $e\ D\ E$ specifies the difference between the more informative description $E$ and the basic description $D$, and is parametrised by an "index erasure" function $e$ from the index set of $E$ to that of $D$. The ornament gives rise to a forgetful function

$$\mathit{forget}\ O : \mu\ E \rightrightarrows (\mu\ D \circ e)$$

For example, there are families of ornaments

$$\mathit{NatD\text{-}ListD} : (A : \mathsf{Set}) \to \mathsf{Orn}\ !\ \mathit{NatD}\ (\mathit{ListD}\ A)$$

and

$$\mathit{ListD\text{-}VecD} : (A : \mathsf{Set}) \to \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ (\mathit{VecD}\ A)$$

(where $! = \mathit{const}\ \mathsf{tt}$) that encode the differences between the list-like datatypes. The function

$$\mathit{forget}\ (\mathit{NatD\text{-}ListD}\ A)\ \{\mathsf{tt}\} : \mathsf{List}\ A \to \mathsf{Nat}$$

[2] This is an abuse of Agda syntax to state the type of $\mathscr{F}\ (\mathit{ListD}\ A)$ and what it computes to, rather than a real Agda definition.

```
data InvImage {I J : Set} (e : J → I) : I → Set where
  ok : (j : J) → InvImage e (e j)

und : {I J : Set} {e : J → I} {i : I} → InvImage e i → J
und (ok j) = j   -- underlying value of an inverse image object

record _⋈_ {I J K : Set} (e : J → I) (f : K → I) : Set where
  constructor _,_
  field
    {i} : I
    j   : InvImage e i
    k   : InvImage f i

pull : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → I
pull = _⋈_.i
```

**Figure 1.** Definitions of inverse images and set-theoretic pullbacks.

computes the length of a list, and the function

$$\mathit{forget}\ (\mathit{ListD\text{-}VecD}\ A) : \forall\ \{n\} \to \mathsf{Vec}\ A\ n \to \mathsf{List}\ A$$

computes the underlying list of a vector.

***Ornamental descriptions.*** Ornaments arise between existing datatype descriptions. The typical scenario of using ornaments, however, is first modifying a base description into a more informative one and then specifying an ornament between the two descriptions. *Ornamental descriptions* are introduced to combine the two steps into one:

$$\mathsf{OrnDesc} : \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I)\ (D : \mathsf{Desc}\ I) \to \mathsf{Set}_1$$

An ornamental description

$$\mathit{OD} : \mathsf{OrnDesc}\ J\ e\ D$$

is like a new description of type Desc $J$, but is written relative to a base description $D$ such that not only can we extract the new description

$$\lfloor \mathit{OD} \rfloor : \mathsf{Desc}\ J$$

but we can also extract an ornament from the base description $D$ to the new description

$$\lceil \mathit{OD} \rceil : \mathsf{Orn}\ e\ D\ \lfloor \mathit{OD} \rfloor$$

An ornamental description is a convenient way to specify a new datatype that has an ornamental relationship with an existing one; it might be thought of as simultaneously denoting the new description and the ornament — the floor and ceiling brackets $\lfloor\_\rfloor$ and $\lceil\_\rceil$ are added to resolve ambiguity. For example, let $\_\leqslant_{A}\_ : A \to A \to \mathsf{Set}$ be an ordering on $A$ and declare a datatype of ordered lists (parametrised by $A$ and $\_\leqslant_{A}\_$) indexed by a lower bound under this ordering:

```
indexfirst data OrdList A _≤_ : A → Set where
  OrdList A _≤_ b
    ∋ nil
    | cons (a : A) (leq : b ≤ a) (as : OrdList A _≤_ a)
```

This datatype can be thought of as being decoded from an ornamental description

$$\mathit{OrdListOD}\ A\ \_\leqslant_{A}\_ : \mathsf{OrnDesc}\ A\ !\ (\mathit{ListD}\ A)$$

which inserts the field *leq* and refines the index of the recursive position to $a$. That is, the underlying description for OrdList is

$$\lfloor \mathit{OrdListOD}\ A\ \_\leqslant_{A}\_ \rfloor : \mathsf{Desc}\ A$$

(so OrdList $A\ \_\leqslant_{A}\_\ b$ desugars to $\mu\ \lfloor \mathit{OrdListOD}\ A\ \_\leqslant_{A}\_ \rfloor\ b$), and

$$\lceil \mathit{OrdListOD}\ A\ \_\leqslant_{A}\_ \rceil : \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ \lfloor \mathit{OrdListOD}\ A\ \_\leqslant_{A}\_ \rfloor$$

is the ornament from lists to ordered lists.

***Promotion predicates and isomorphisms.*** A more complete view of ornaments is given by a particular class of isomorphisms induced by ornaments. For example, consider the following datatype:

```
indexfirst data Ordered A _≤A_ : A → List A → Set where
  Ordered A _≤A_ b []          ∋ nil
  Ordered A _≤A_ b (a :: as)
    ∋ cons (leq : b ≤A a) (s : Ordered A _≤A_ a as)
```

A proof of $\mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b\ as$ consists of a series of inequality proofs and ensures that $as$ is ordered and bounded below by $b$, so $\mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b$ is a predicate that characterises ordered lists (with a lower bound). The Ordered predicate helps to formulate the following isomorphisms

$$\mathsf{OrdList}\ A\ \_{\leqslant_A}\_\ b \cong \Sigma[as : \mathsf{List}\ A]\ \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b\ as \quad (1)$$

for all $b : A$ — an ordered list bounded below by $b$ can be converted to/from a plain list and a proof that it is ordered and bounded below by $b$. In general, for any ornament $O : \mathsf{Orn}\ e\ D\ E$ (where $D : \mathsf{Desc}\ I$ and $E : \mathsf{Desc}\ J$) there is a *promotion predicate*

$$PromP\ O : \{i : I\}\ (j : \mathsf{InvImage}\ e\ i) \to \mu\ D\ i \to \mathsf{Set}$$

(where the definition of InvImage is shown in Figure 1) such that there is a family of *promotion isomorphisms*

$$\mu\ E\ (und\ j) \cong \Sigma[d : \mu\ D\ i]\ PromP\ O\ j\ d \quad (2)$$

for all $i : I$ and $j : \mathsf{InvImage}\ e\ i$. Moreover, the first half of the forward direction of the promotion isomorphisms is exactly *forget O*. For example, $\mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b$ is syntactic sugar for $PromP\ \lceil OrdListOD\ A\ \_{\leqslant_A}\_\rceil\ (\mathsf{ok}\ b)$, and the isomorphisms (1) are a specialisation of the promotion isomorphisms (2) for the ornament $\lceil OrdListOD\ A\ \_{\leqslant_A}\_\rceil$. The promotion isomorphisms are named as such because the right-to-left direction can be interpreted as promoting an element of the basic type (e.g., a list) to an element of the more informative type (e.g., an ordered list) provided that there is a proof that the promotion predicate (e.g., Ordered) is satisfied.

***Parallel composition of ornaments.*** At one point in Section 5 we will need *parallel composition* of ornaments. Let $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$, and $F : \mathsf{Desc}\ K$. Given two ornaments $O : \mathsf{Orn}\ e\ D\ E$ and $P : \mathsf{Orn}\ f\ D\ F$ (note the common description $D$) where $e : J \to I$ and $f : K \to I$, parallel composition of $O$ and $P$ is an ornamental description relative to $D$:

$$O \otimes P : \mathsf{OrnDesc}\ (e \bowtie f)\ pull\ D$$

where $e \bowtie f$ is the set-theoretic pullback of $e$ and $f$, and $pull : e \bowtie f \to I$ is the usual projection of pullbacks (definitions shown in Figure 1). Intuitively: both $O$ and $P$ encode modifications to the same base description $D$; parallel composition of $O$ and $P$ produces a new datatype $\lfloor O \otimes P \rfloor$ by committing all the modifications to $D$, and merges all the modifications into one ornament $\lceil O \otimes P \rceil$. For example, parallel composition of the ornament $\lceil OrdListOD\ A\ \_{\leqslant_A}\_\rceil$ from lists to ordered lists and the ornament *ListD-VecD A* from lists to vectors produces (i) a new datatype of *ordered vectors*

```
indexfirst data OrdVec A _≤A_ : A → Nat → Set where
  OrdVec A _≤A_ b n
    ∋ nil (neq : n ≡ zero)
    | cons (a : A) (leq : b ≤A a)
           (m : Nat) (as : OrdVec A _≤A_ a m)
           (meq : n ≡ suc m)
```

such that $\mathsf{OrdVec}\ A\ \_{\leqslant_A}\_\ b\ n$ desugars to

$$\mu\ \lfloor \lceil OrdListOD\ A\ \_{\leqslant_A}\_\rceil \otimes ListD\text{-}VecD\ A \rfloor\ (\mathsf{ok}\ b, \mathsf{ok}\ n)$$

and (ii) an ornament that gives rise to a forgetful function from ordered vectors to plain lists which retains only the list elements.

For a parallel composed ornament $\lceil O \otimes P \rceil$, the promotion isomorphisms (2) can be transformed into a more reusable form:

$$\mu\ \lfloor O \otimes P \rfloor\ (j,k) \cong \Sigma[d : \mu\ D\ i]\ PromP\ O\ j\ d \times PromP\ P\ k\ d \quad (3)$$

for all $i : I, j : \mathsf{InvImage}\ e\ i$, and $k : \mathsf{InvImage}\ f\ i$. For example, for ordered vectors we get

$$\mathsf{OrdVec}\ A\ \_{\leqslant_A}\_\ b\ n \cong \Sigma[as : \mathsf{List}\ A]\ \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b\ as \times \mathsf{Length}\ A\ n\ as$$

where Length $A$ is the promotion predicate for the ornament from lists to vectors, asserting that a list has a particular length:

```
indexfirst data Length A : Nat → List A → Set where
  Length A n []          ∋ nil (neq : n ≡ zero)
  Length A n (a :: as)
    ∋ cons (m : Nat) (l : Length A m as) (meq : n ≡ suc m)
```

## 3. Relational program derivation in Agda and relational algebraic ornamentation

In this section, we first introduce and formalise some basic notions in relational program derivation [4] by importing and generalising a small part of the AoPA library [18]. We then introduce *relational algebraic ornamentation*, which acts as a bridge between the two worlds of internalist programming and relational program derivation. At the end of this section is an example about the *Fold Fusion Theorem* [4, Section 6.2] and how the theorem translates to conversion functions between algebraically ornamented datatypes.

***Basic definitions for relational program derivation.*** One common approach to program derivation is by algebraic transformations of functional programs: one begins with a specification in the form of a functional program that expresses straightforward but possibly inefficient computation, and transforms it into an extensionally equal but more efficient functional program by applying algebraic laws and theorems. Using functional programs as the specification language means that specifications are directly executable, but the deterministic nature of functional programs can result in less flexible specifications. For example, when specifying an optimisation problem using a functional program that generates all feasible solutions and chooses an optimal one among them, the program would enforce a particular way of choosing the optimal solution, but such enforcement should not be part of the specification. To gain more flexibility, the specification language was later generalised to *relational programs*. With relational programs, we specify only the relationship between input and output without actually specifying a way to execute the programs, so specifications in the form of relational programs can be as flexible as possible. Though lacking a directly executable semantics, most relational programs can still be read computationally as potentially partial and nondeterministic mappings, so relational specifications largely remain computationally intuitive as functional specifications.

To emphasise the computational interpretation of relations, we will mainly model a relation between sets $A$ and $B$ as a function sending each element of $A$ to a *subset* of $B$. We define subsets by

$$\mathscr{P} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathscr{P}\ A = A \to \mathsf{Set}$$

That is, a subset $s : \mathscr{P}\ A$ is a characteristic function that assigns a type to each element of $A$, and $a : A$ is considered to be a member of $s$ if the type $s\ a : \mathsf{Set}$ is inhabited. We may regard $\mathscr{P}\ A$ as the type of computations that nondeterministically produce an element of $A$. A simple example is

$$any : \{A : \mathsf{Set}\} \to \mathscr{P}\ A$$
$$any = const\ \top$$

The subset $any : \mathscr{P}\,A$ associates the unit type $\top$ with every element of $A$. Since $\top$ is inhabited, *any* can produce any element of $A$. $\mathscr{P}$ cannot be made into a conventional monad because it is not an endofunctor, but it still has a monadic structure [3]: *return* and $\_\gg\!=\_$ are defined as

$$\begin{aligned}
&\mathit{return} : \{A : \mathsf{Set}\} \to A \to \mathscr{P}\,A\\
&\mathit{return} = _\equiv_\\
&_\gg\!=_ : \{A\,B : \mathsf{Set}\} \to \mathscr{P}\,A \to (A \to \mathscr{P}\,B) \to \mathscr{P}\,B\\
&_\gg\!=_\,\{A\}\,s\,f = \lambda b \to \Sigma[\,a : A\,]\ s\,a \times f\,a\,b
\end{aligned}$$

The subset $\mathit{return}\,a : \mathscr{P}\,A$ for some $a : A$ simplifies to $\lambda a' \to a \equiv a'$ (where $\_\equiv\_$ is propositional equality), so $a$ is the only member of the subset; if $s : \mathscr{P}\,A$ and $f : A \to \mathscr{P}\,B$, then the subset $s \gg\!= f : \mathscr{P}\,B$ is the union of all the subsets $f\,a : \mathscr{P}\,B$ where $a$ ranges over the elements of $A$ that belong to $s$, i.e., an element $b : B$ is a member of $s \gg\!= f$ exactly when there exists some $a : A$ belonging to $s$ such that $b$ is a member of $f\,a$.

We will mainly use relations between families of sets in this paper: if $X, Y : I \to \mathsf{Set}$ for some $I : \mathsf{Set}$, a relation from $X$ to $Y$ is defined as a family of relations from $X\,i$ to $Y\,i$ for every $i : I$.

$$\begin{aligned}
&_\rightsquigarrow_ : \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}_1\\
&X \rightsquigarrow Y = \forall\,\{i\} \to X\,i \to \mathscr{P}\,(Y\,i)
\end{aligned}$$

We can use the subset combinators to define relations. For example, the following combinator *fun* lifts a family of functions into a family of relations.

$$\begin{aligned}
&\mathit{fun} : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\} \to (X \rightrightarrows Y) \to (X \rightsquigarrow Y)\\
&\mathit{fun}\,f\,x = \mathit{return}\,(f\,x)
\end{aligned}$$

The identity relation is just the identity functions lifted to relations.

$$\begin{aligned}
&\mathit{idR} : \{I : \mathsf{Set}\}\,\{X : I \to \mathsf{Set}\} \to (X \rightsquigarrow X)\\
&\mathit{idR} = \mathit{fun}\,\mathit{id}
\end{aligned}$$

Composition of relations is easily defined with $\_\gg\!=\_$: computing $R \cdot S$ on input $x$ is first computing $S\,x$ and then feeding the result to $R$.

$$\begin{aligned}
&_\cdot_ : \{I : \mathsf{Set}\}\,\{X\,Y\,Z : I \to \mathsf{Set}\} \to\\
&\qquad (Y \rightsquigarrow Z) \to (X \rightsquigarrow Y) \to (X \rightsquigarrow Z)\\
&(R \cdot S)\,x = S\,x \gg\!= R
\end{aligned}$$

Or we may choose to define a relation pointwise, like

$$\begin{aligned}
&_\cap_ : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\} \to\\
&\qquad (X \rightsquigarrow Y) \to (X \rightsquigarrow Y) \to (X \rightsquigarrow Y)\\
&(R \cap S)\,x\,y = R\,x\,y \times S\,x\,y
\end{aligned}$$

This defines the meet of two relations. Unlike a function, which distinguishes between input and output, inherently a relation treats its domain and codomain symmetrically. This is reflected by the presence of the following *converse* operator:

$$\begin{aligned}
&_^\circ : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\} \to (X \rightsquigarrow Y) \to (Y \rightsquigarrow X)\\
&(R^\circ)\,y\,x = R\,x\,y
\end{aligned}$$

A relation can thus be "run backwards" simply by taking its converse. The nondeterministic and bidirectional nature of relations makes them a powerful and concise language for specifications, as will be demonstrated in Section 5.

Laws and theorems in relational program derivation are formulated with *relational inclusion*

$$\begin{aligned}
&_\subseteq_ : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\}\,(R\,S : X \rightsquigarrow Y) \to \mathsf{Set}\\
&R \subseteq S = \forall\,\{i\} \to (x : X\,i)\,(y : Y\,i) \to R\,x\,y \to S\,x\,y
\end{aligned}$$

or equivalence of relations, which is defined as two-way inclusion:

$$\begin{aligned}
&_\simeq_ : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\}\,(R\,S : X \rightsquigarrow Y) \to \mathsf{Set}\\
&R \simeq S = (R \subseteq S) \times (S \subseteq R)
\end{aligned}$$

We will also need *relators*, i.e., monotonic functors on relations with respect to relational inclusion.

$$\begin{aligned}
&\mathscr{R} : \{I : \mathsf{Set}\}\,(D : \mathsf{Desc}\,I)\,\{X\,Y : I \to \mathsf{Set}\} \to\\
&\qquad (X \rightsquigarrow Y) \to (\mathscr{F}\,D\,X \rightsquigarrow \mathscr{F}\,D\,Y)
\end{aligned}$$

If $R : X \rightsquigarrow Y$, the relation $\mathscr{R}\,D\,R : \mathscr{F}\,D\,X \rightsquigarrow \mathscr{F}\,D\,Y$ applies $R$ to the recursive positions of its input, leaving everything else intact. For example, if $D = \mathit{ListD}\,A$ (for some $A : \mathsf{Set}$), then $\mathscr{R}\,(\mathit{ListD}\,A)$ essentially specialises to

$$\begin{aligned}
&\mathscr{R}\,(\mathit{ListD}\,A) : \{X\,Y : I \to \mathsf{Set}\} \to\\
&\qquad (X \rightsquigarrow Y) \to (\mathscr{F}\,(\mathit{ListD}\,A)\,X \rightsquigarrow \mathscr{F}\,(\mathit{ListD}\,A)\,Y)\\
&\mathscr{R}\,(\mathit{ListD}\,A)\,R\,(\text{`}\mathsf{nil}\quad,\mathsf{tt}) = \mathit{return}\,(\text{`}\mathsf{nil},\mathsf{tt})\\
&\mathscr{R}\,(\mathit{ListD}\,A)\,R\,(\text{`}\mathsf{cons},a,x) = R\,x \gg\!= \lambda y \to \mathit{return}\,(\text{`}\mathsf{cons},a,y)
\end{aligned}$$

Among other properties, we can prove that $\mathscr{R}\,D$ preserves identity ($\mathscr{R}\,D\,\mathit{idR} \simeq \mathit{idR}$), composition ($\mathscr{R}\,D\,(R \cdot S) \simeq \mathscr{R}\,D\,R \cdot \mathscr{R}\,D\,S$), converse ($\mathscr{R}\,D\,(R^\circ) \simeq (\mathscr{R}\,D\,R)^\circ$), and is monotonic ($R \subseteq S$ implies $\mathscr{R}\,D\,R \subseteq \mathscr{R}\,D\,S$).

With relational inclusion, many concepts can be expressed in a surprisingly concise way. For example, a relation $R$ is a preorder if it is reflexive and transitive. In relational terms, these two conditions are expressed simply as $\mathit{idR} \subseteq R$ and $R \cdot R \subseteq R$, and are easily manipulable in calculations. Another important notion is *monotonic algebras* [4, Section 7.2]: an algebra $S : \mathscr{F}\,D\,X \rightsquigarrow X$ is *monotonic* on $R : X \rightsquigarrow X$ (usually an ordering) if

$$S \cdot \mathscr{R}\,D\,R \subseteq R \cdot S$$

which says that if two input values to $S$ have their recursive positions related by $R$ and are otherwise equal, then the output values would still be related by $R$. In the context of optimisation problems, monotonicity can be used to capture the *principle of optimality*, as will be shown in Section 5.

Having defined relations as nondeterministic mappings, it is straightforward to port the datatype-generic *fold* to relations:

$$\begin{aligned}
&(\!|_|\!) : \{I : \mathsf{Set}\}\,\{D : \mathsf{Desc}\,I\}\,\{X : I \to \mathsf{Set}\} \to\\
&\qquad (\mathscr{F}\,D\,X \rightsquigarrow X) \to (\mu\,D \rightsquigarrow X)
\end{aligned}$$

The definition of $(\!|\_|\!)$ is obtained by rewriting the definition of *fold* with the subset combinators. For example, the relational fold on lists would essentially be

$$\begin{aligned}
&(\!|_|\!)\,\{\top\}\,\{\mathit{ListD}\,A\} : \{X : \top \to \mathsf{Set}\} \to\\
&\qquad\qquad (\mathscr{F}\,(\mathit{ListD}\,A)\,X \rightsquigarrow X) \to\\
&\qquad\qquad (\mu\ \ (\mathit{ListD}\,A) \quad \rightsquigarrow X)\\
&(\!|R|\!)\,[\,] \qquad = R\,(\text{`}\mathsf{nil},\mathsf{tt})\\
&(\!|R|\!)\,(a :: as) = (\!|R|\!)\,as \gg\!= \lambda x \to R\,(\text{`}\mathsf{cons},a,x)
\end{aligned}$$

The functional and relational fold operators are related by the following lemma:

$$\begin{aligned}
&\mathit{fun}\text{-}\mathit{preserves}\text{-}\mathit{fold} :\\
&\quad \{I : \mathsf{Set}\}\,(D : \mathsf{Desc}\,I)\,\{X : I \to \mathsf{Set}\}\\
&\quad (f : \mathscr{F}\,D\,X \rightrightarrows X) \to \mathit{fun}\,(\mathit{fold}\,f) \simeq (\!|\mathit{fun}\,f|\!)
\end{aligned}$$

***Relational algebraic ornamentation.*** We now turn to relational algebraic ornamentation, the key construct that bridges internalist programming and relational program derivation. Let $R : \mathscr{F}\,(\mathit{ListD}\,A)\,X \rightsquigarrow X$ (where $X : \top \to \mathsf{Set}$) be a relational algebra for lists. We can define a datatype of "algebraic lists" as

$$\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{AlgList}\,A\,R : X\,\mathsf{tt} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{AlgList}\,A\,R\,x \ni \mathsf{nil}\,(\mathit{rnil} : R\,(\text{`}\mathsf{nil},\mathsf{tt})\,x)\\
&\qquad\qquad\qquad \mid\ \mathsf{cons}\,(a : A)\,(x' : X\,\mathsf{tt})\,(as : \mathsf{AlgList}\,A\,R\,x')\\
&\qquad\qquad\qquad\qquad (\mathit{rcons} : R\,(\text{`}\mathsf{cons},a,x')\,x)
\end{aligned}$$

There is an ornament from lists to algebraic lists which marks the fields *rnil*, $x'$, and *rcons* in AlgList as additional and refines the index of the recursive position to $x'$. The promotion predicate for this ornament is

$$\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{AlgListP}\,A\,R : X\,\mathsf{tt} \to \mathsf{List}\,A \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{AlgListP}\,A\,R\,x\,[\,] \qquad \ni \mathsf{nil}\,(\mathit{rnil} : R\,(\text{`}\mathsf{nil},\mathsf{tt})\,x)\\
&\quad \mathsf{AlgListP}\,A\,R\,x\,(a :: as) \ni \mathsf{cons}\,(x' : X\,\mathsf{tt})\\
&\qquad\qquad\qquad\qquad\qquad (p : \mathsf{AlgListP}\,A\,R\,x'\,as)\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{rcons} : R\,(\text{`}\mathsf{cons},a,x')\,x)
\end{aligned}$$

A simple argument by induction shows that $\mathsf{AlgListP}\ A\ R\ x\ as$ is in fact isomorphic to $(\!|R|\!)\ as\ x$ for any $as : \mathsf{List}\ A$ and $x : X\ \mathsf{tt}$. As a corollary, we have

$$\mathsf{AlgList}\ A\ R\ x \cong \Sigma[as : \mathsf{List}\ A]\ (\!|R|\!)\ as\ x \tag{4}$$

for any $x : X\ \mathsf{tt}$ by (2). That is, an algebraic list is exactly a plain list and a proof that the list folds to $x$ using the algebra $R$. The vector datatype is a special case of AlgList — to see that, define

$$\begin{aligned} &\mathit{length\text{-}alg} : \mathscr{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat} \\ &\mathit{length\text{-}alg}\ (\text{`}\mathsf{nil}\quad,\mathsf{tt}) \quad = \mathsf{zero} \\ &\mathit{length\text{-}alg}\ (\text{`}\mathsf{cons}, a, n) \ = \mathsf{suc}\ n \end{aligned}$$

and take $R = \mathit{fun\ length\text{-}alg}$. From (4) we have the isomorphisms

$$\mathsf{Vec}\ A\ n \cong \Sigma[as : \mathsf{List}\ A]\ (\!|\mathit{fun\ length\text{-}alg}|\!)\ as\ n$$

for all $n : \mathsf{Nat}$, from which we can derive

$$\mathsf{Vec}\ A\ n \cong \Sigma[as : \mathsf{List}\ A]\ \mathit{length}\ as \equiv n$$

by *fun-preserves-fold*, after defining $\mathit{length} = \mathit{fold\ length\text{-}alg}$.

The above can be generalised to all datatypes encoded by the Desc universe. Let $D : \mathsf{Desc}\ I$ be a description and $R : \mathscr{F}\ D\ X \rightsquigarrow X$ (where $X : I \to \mathsf{Set}$) an algebra. The (relational) *algebraic ornamentation* of $D$ with $R$ is an ornamental description

$$\mathit{algOrn}\ D\ R : \mathsf{OrnDesc}\ (\Sigma\ I\ X)\ \mathsf{proj}_1\ D$$

(where $\mathsf{proj}_1 : \Sigma\ I\ X \to I$). Its definition is a slight generalisation of the one given by Dagand and McBride [7, supplementary code]. The promotion predicate for the ornament $\lceil \mathit{algOrn}\ D\ R \rceil$ is pointwise isomorphic to $(\!|R|\!)$, i.e.,

$$\mathit{PromP}\ \lceil \mathit{algOrn}\ D\ R \rceil\ (\mathsf{ok}\ (i,x))\ d \cong (\!|R|\!)\ d\ x \tag{5}$$

for all $i : I$, $x : X\ i$, and $d : \mu\ D\ i$. As a corollary, we have the following isomorphisms

$$\mu\ \lfloor \mathit{algOrn}\ D\ R \rfloor\ (i,x) \cong \Sigma[d : \mu\ D\ i]\ (\!|R|\!)\ d\ x \tag{6}$$

for all $i : I$ and $x : X\ i$ by (2). For example, taking $D = \mathit{ListD}\ A$, the type $\mathsf{AlgList}\ A\ R\ x$ can be thought of as the high-level presentation of $\mu\ \lfloor \mathit{algOrn}\ (\mathit{ListD}\ A)\ R \rfloor\ (\mathsf{tt}, x)$. Algebraic ornamentation is a very convenient method for adding new indices to inductive families, and most importantly, it says precisely what the new indices mean. The method was demonstrated by McBride [15] with a correct-by-construction compiler for a small language, and will be demonstrated again in Section 5.

***Example: the Fold Fusion Theorem.*** As a first example of bridging internalist programming with relational program derivation through algebraic ornamentation, let us consider the *Fold Fusion Theorem* [4, Section 6.2]: Let $D : \mathsf{Desc}\ I$ be a description, $R : X \rightsquigarrow Y$ a relation, and $S : \mathscr{F}\ D\ X \rightsquigarrow X$ and $T : \mathscr{F}\ D\ Y \rightsquigarrow Y$ be algebras. If $R$ is a homomorphism from $S$ to $T$, i.e.,

$$R \cdot S \simeq T \cdot \mathscr{R}\ D\ R$$

which is referred to as the *fusion condition*, then we have

$$R \cdot (\!|S|\!) \simeq (\!|T|\!)$$

The above is, in fact, a corollary of two variations of Fold Fusion that replace relational equivalence in the statement of the theorem with relational inclusion. One of the variations is

$$R \cdot S \subseteq T \cdot \mathscr{R}\ D\ R \quad \text{implies} \quad R \cdot (\!|S|\!) \subseteq (\!|T|\!)$$

This can be used with (6) to derive a conversion between algebraically ornamented datatypes:

$$\begin{aligned} &\mathit{algOrn\text{-}fusion\text{-}}\subseteq\ D\ R\ S\ T : \\ &\quad R \cdot S \subseteq T \cdot \mathscr{R}\ D\ R \to \\ &\quad \{i : I\}\ (x : X\ i) \to \mu\ \lfloor \mathit{algOrn}\ D\ S \rfloor\ (i,x) \to \\ &\quad (y : Y\ i) \to R\ x\ y \to \mu\ \lfloor \mathit{algOrn}\ D\ T \rfloor\ (i,y) \end{aligned}$$

The other variation of Fold Fusion simply reverses the direction of inclusion:

$$R \cdot S \supseteq T \cdot \mathscr{R}\ D\ R \quad \text{implies} \quad R \cdot (\!|S|\!) \supseteq (\!|T|\!)$$

which translates to the conversion

$$\begin{aligned} &\mathit{algOrn\text{-}fusion\text{-}}\supseteq\ D\ R\ S\ T : \\ &\quad R \cdot S \supseteq T \cdot \mathscr{R}\ D\ R \to \\ &\quad \{i : I\}\ (y : Y\ i) \to \mu\ \lfloor \mathit{algOrn}\ D\ T \rfloor\ (i,y) \to \\ &\quad \Sigma[x : X\ i]\ \mu\ \lfloor \mathit{algOrn}\ D\ S \rfloor\ (i,x)\ \times\ R\ x\ y \end{aligned}$$

For a simple example, suppose that we need a "bounded" vector datatype, i.e., lists indexed with an upper bound on their length. A quick thought might lead to this definition

$$\begin{aligned} &\mathsf{BVec} : \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set} \\ &\mathsf{BVec}\ A\ m = \\ &\quad \mu\ \lfloor \mathit{algOrn}\ (\mathit{ListD}\ A)\ (\mathit{geq} \cdot \mathit{fun\ length\text{-}alg}) \rfloor\ (\mathsf{tt}, m) \end{aligned}$$

where $\mathit{geq} = \lambda\ x\ y \to x \leqslant y : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$ maps a natural number $x$ to any natural number that is at least $x$. The isomorphisms (6) specialise for BVec to

$$\mathsf{BVec}\ A\ m \cong \Sigma[as : \mathsf{List}\ A]\ (\!|\mathit{geq} \cdot \mathit{fun\ length\text{-}alg}|\!)\ as\ m$$

But is BVec really the bounded vectors? Indeed it is, because we can deduce

$$\mathit{geq} \cdot (\!|\mathit{fun\ length\text{-}alg}|\!) \simeq (\!|\mathit{geq} \cdot \mathit{fun\ length\text{-}alg}|\!)$$

by Fold Fusion (where $(\!|\mathit{fun\ length\text{-}alg}|\!)$ is equivalent to *fun length* by *fun-preserves-fold*). The fusion condition is

$$\mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \simeq \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \cdot \mathscr{R}\ (\mathit{ListD}\ A)\ \mathit{geq}$$

The left-to-right inclusion is easily calculated as follows:

$$\begin{aligned} & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \\ \subseteq\ & \{\, \mathit{idR}\ \text{identity} \,\} \\ & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \cdot \mathit{idR} \\ \subseteq\ & \{\, \text{relator preserves identity} \,\} \\ & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \cdot \mathscr{R}\ (\mathit{ListD}\ A)\ \mathit{idR} \\ \subseteq\ & \{\, \mathit{geq}\ \text{reflexive} \,\} \\ & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \cdot \mathscr{R}\ (\mathit{ListD}\ A)\ \mathit{geq} \end{aligned}$$

And from right to left:

$$\begin{aligned} & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \cdot \mathscr{R}\ (\mathit{ListD}\ A)\ \mathit{geq} \\ \subseteq\ & \{\, \mathit{fun\ length\text{-}alg}\ \text{monotonic on}\ \mathit{geq} \,\} \\ & \mathit{geq} \cdot \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \\ \subseteq\ & \{\, \mathit{geq}\ \text{transitive} \,\} \\ & \mathit{geq} \cdot \mathit{fun\ length\text{-}alg} \end{aligned}$$

Note that these calculations are good illustrations of the power of relational calculation despite their simplicity — they are straightforward symbolic manipulations, hiding details like quantifier reasoning behind the scenes. As demonstrated by the AoPA library [18], they can be faithfully formalised with preorder reasoning combinators in Agda and used to discharge the fusion conditions of *algOrn-fusion-*$\subseteq$ and *algOrn-fusion-*$\supseteq$. Hence we get two conversions, one of type

$$\mathsf{Vec}\ A\ n \to (n \leqslant m) \to \mathsf{BVec}\ A\ m$$

which relaxes a vector of length $n$ to a bounded vector whose length is bounded above by some $m$ that is at least $n$, and the other of type

$$\mathsf{BVec}\ A\ m \to \Sigma[n : \mathsf{Nat}]\ \mathsf{Vec}\ A\ n\ \times\ (n \leqslant m)$$

which converts a bounded vector whose length is at most $m$ to a vector of length precisely $n$ and guarantees that $n$ is at most $m$.

Theoretically, the conversions derived from Fold Fusion are actually more generally applicable than they seem, because *every ornament is an algebraic ornament up to isomorphism*. This we show next.

## 4. Completeness of relational algebraic ornaments

Consider the AlgList datatype in Section 3 again. The way it is refined relative to the plain list datatype looks canonical, in the

sense that any variation of the list datatype can be programmed as a special case of AlgList: we can choose whatever index set we want by setting the carrier of the algebra $R$; and by carefully programming $R$, we can insert fields into the list datatype that add more information or put restriction on fields and indices. For example, if we want some new information in the nil case, we can program $R$ such that $R\ (\text{`nil}, \mathsf{tt})\ x$ contains a field requesting that information; if, in the cons case, we need the targeted index $x$, the head element $a$, and the index $x'$ of the recursive position to be related in some way, we can program $R$ such that $R\ (\text{`cons}, a, x')\ x$ expresses that relationship.

The above observation leads to the following general theorem: Let $O : \mathsf{Orn}\ e\ D\ E$ be an ornament from $D : \mathsf{Desc}\ I$ to $E : \mathsf{Desc}\ J$. There is a *classifying algebra* for $O$

$$\mathit{clsAlg}\ O : \mathscr{F}\ D\ (\mathsf{InvImage}\ e) \rightsquigarrow \mathsf{InvImage}\ e$$

such that there are isomorphisms

$$\mu\ \lfloor \mathit{algOrn}\ D\ (\mathit{clsAlg}\ O) \rfloor\ (e\ j, \mathsf{ok}\ j) \cong \mu\ E\ j$$

for all $j : J$. That is, the algebraic ornamentation of $D$ using the classifying algebra derived from $O$ produces a datatype isomorphic to $\mu\ E$, so intuitively the algebraic ornament has the same content as $O$. We may interpret this theorem as saying that algebraic ornaments are "complete" for the ornament language: any relationship between datatypes that can be described by an ornament can be described up to isomorphism by an algebraic ornament.

The completeness theorem brings up a nice algebraic intuition about inductive families. Consider the ornament from lists to vectors, for example. This ornament specifies that the type $\mathsf{List}\ A$ is refined by the collection of types $\mathsf{Vec}\ A\ n$ for all $n : \mathsf{Nat}$. A list, say $a :: b :: [\,] : \mathsf{List}\ A$, can be reconstructed as a vector by starting in the type $\mathsf{Vec}\ A\ \mathsf{zero}$ as $[\,]$, jumping to the next type $\mathsf{Vec}\ A\ (\mathsf{suc}\ \mathsf{zero})$ as $b :: [\,]$, and finally landing in $\mathsf{Vec}\ A\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}))$ as $a :: b :: [\,]$. The list is thus *classified* as having length 2, as computed by the fold function *length*, and the resulting vector is a fused representation of the list and the classification proof. In the case of vectors, this classification is total and deterministic: every list is classified under one and only one index. But in general, classifications can be partial and nondeterministic. For example, promoting a list to an ordered list is classifying the list under an index that is a lower bound of the list. The classification process checks at each jump whether the list is still ordered; this check can fail, so an unordered list would "disappear" midway through the classification. Also there can be more than one lower bound for an ordered list, so the list can end up being classified under any one of them. Algebraic ornamentation in its original functional form can only capture part of this intuition about classification, namely those classifications that are total and deterministic. By generalising algebraic ornamentation to accept relational algebras, bringing in partiality and nondeterminacy, this idea about classification is captured in its entirety — a classification is just a relational fold computing the index that classifies an element. All ornaments specify classifications, and thus can be transformed into algebraic ornaments.

For more examples, let us first look at the classifying algebra for the ornament from natural numbers to lists. The base functor for natural numbers is

$$\mathscr{F}\ \mathit{NatD} : (\top \to \mathsf{Set}) \to (\top \to \mathsf{Set})$$
$$\mathscr{F}\ \mathit{NatD}\ X\ \_ = \Sigma\ \mathsf{LTag}\ (\lambda\{\text{`nil} \to \top; \text{`cons} \to X\ \mathsf{tt}\})$$

And the classifying algebra for the ornament *NatD-ListD A* is essentially

$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A) : \mathscr{F}\ \mathit{NatD}\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\text{`nil}\quad, \_\quad)\ (\mathsf{ok}\ \mathsf{tt}) = \top$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\text{`cons}, \mathsf{ok}\ t)\ (\mathsf{ok}\ \mathsf{tt}) = A \times (t \equiv \mathsf{tt})$$

The result of folding a natural number $n$ with this algebra is uninteresting, as it can only be $\mathsf{ok}\ \mathsf{tt}$. The fold, however, requires an element of $A$ for each successor node it encounters, so a proof that $n$ goes through the fold consists of $n$ elements of $A$. Another example is the ornament $OL = \lceil \mathit{OrdListOD}\ A\ \_\leqslant_A\_ \rceil$ from lists to ordered lists, whose classifying algebra is essentially

$$\mathit{clsAlg}\ OL : \mathscr{F}\ (\mathit{ListD}\ A)\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ OL\ (\text{`nil}\quad, \_\quad)\ (\mathsf{ok}\ b) = \top$$
$$\mathit{clsAlg}\ OL\ (\text{`cons}, a, \mathsf{ok}\ b')\ (\mathsf{ok}\ b) = (b \leqslant_A a) \times (b' \equiv a)$$

In the nil case, the empty list can be mapped to any $\mathsf{ok}\ b$ because any $b : A$ is a lower bound of the empty list; in the cons case, where $a : A$ is the head and $\mathsf{ok}\ b'$ is a result of classifying the tail, i.e., $b' : A$ is a lower bound of the tail, the list can be mapped to $\mathsf{ok}\ b$ if $b : A$ is a lower bound of $a$ and $a$ is exactly $b'$.

Perhaps the most important consequence of the completeness theorem (in its present form) is that it provides a new perspective on the expressive power of ornaments and inductive families. We showed in a previous paper [12] that every ornament induces a promotion predicate and a corresponding family of isomorphisms (which is restated as (2) in Section 2). But one question was untouched: can we determine (independently from ornaments) the range of predicates induced by ornaments? An answer to this question would tell us something about the expressive power of ornaments, and also about the expressive power of inductive families in general, since the inductive families we use are usually ornamentations of simpler algebraic datatypes from traditional functional programming. The completeness theorem offers such an answer: ornament-induced promotion predicates are exactly those expressible as relational folds (up to pointwise isomorphism). In other words, a predicate can be baked into a datatype by ornamentation if and only if it can be thought of as a nondeterministic classification of the elements of the datatype with a relational fold. This is more a guideline than a precise criterion, though, as the closest work about characterisation of the expressive power of folds discusses only functional folds [11] (however, we believe that those results generalise to relations too). But this does encourage us to think about ornamentation computationally and to design new datatypes with relational algebraic methods. We illustrate this point with a solution to the *minimum coin change problem* in the next section.

## 5. Example: the minimum coin change problem

Suppose that we have an unlimited number of 1-penny, 2-pence, and 5-pence coins, modelled by the following datatype:

```
data Coin : Set where
  1p 2p 5p : Coin
```

Given $n : \mathsf{Nat}$, the *minimum coin change problem* asks for the least number of coins that make up $n$ pence. We can give a relational specification of the problem with the following operator:

$$\mathit{min}\_\cdot\Lambda\_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}$$
$$(R : Y \rightsquigarrow Y)\ (S : X \rightsquigarrow Y) \to (X \rightsquigarrow Y)$$
$$(\mathit{min}\ R\ \cdot\Lambda\ S)\ x\ y = S\ x\ y \times (\forall\ y' \to S\ x\ y' \to R\ y'\ y)$$

An input $x : X\ i$ for some $i : I$ is mapped by $\mathit{min}\ R\ \cdot\Lambda\ S$ to $y : Y\ i$ if $y$ is a possible result in $S\ x : \mathscr{P}\ (Y\ i)$ and is the smallest such result under $R$, in the sense that any $y'$ in $S\ x : \mathscr{P}\ (Y\ i)$ must satisfy $R\ y'\ y$ (i.e., $R$ maps larger inputs to smaller outputs). Intuitively, we can think of $\mathit{min}\ R\ \cdot\Lambda\ S$ as consisting of two steps: the first step $\Lambda\ S$ computes the set of all possible results yielded by $S$, and the second step $\mathit{min}\ R$ chooses a minimum result from that set (nondeterministically). We use bags of coins as the type of solutions, and represent them as decreasingly ordered lists indexed with an upper bound. (This is a deliberate choice to make the derivation work, but one would naturally turn to this representation having attempted to apply the *Greedy Theorem*, which will be introduced shortly.) If we define the ordering on coins as

$$\_\leqslant_{\mathsf{C}}\_ : \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set}$$
$$c \leqslant_{\mathsf{C}} d \;=\; \mathit{value}\ c \leqslant \mathit{value}\ d$$

where the values of the coins are defined by

$$\mathit{value} : \mathsf{Coin} \to \mathsf{Nat}$$
$$\mathit{value}\ \mathsf{1p} \;=\; 1$$
$$\mathit{value}\ \mathsf{2p} \;=\; 2$$
$$\mathit{value}\ \mathsf{5p} \;=\; 5$$

then the datatype of coin bags we use is

$$\textbf{indexfirst data}\ \mathsf{CoinBag} : \mathsf{Coin} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{CoinBag}\ c \ni \mathsf{nil}$$
$$\mid\ \mathsf{cons}\ (d : \mathsf{Coin})\ (\mathit{leq} : d \leqslant_{\mathsf{C}} c)\ (b : \mathsf{CoinBag}\ d)$$

Down at the universe level, the (ornamental) description of CoinBag (relative to List Coin) is simply that of OrdList Coin ($\mathit{flip}\ \_\leqslant_{\mathsf{C}}\_$).

$$\mathit{CoinBagOD} : \mathsf{OrnDesc}\ \mathsf{Coin}\ !\ (\mathit{ListD}\ \mathsf{Coin})$$
$$\mathit{CoinBagOD} \;=\; \mathit{OrdListOD}\ \mathsf{Coin}\ (\mathit{flip}\ \_\leqslant_{\mathsf{C}}\_)$$

$$\mathit{CoinBagD} : \mathsf{Desc}\ \mathsf{Coin}$$
$$\mathit{CoinBagD} \;=\; \lfloor \mathit{CoinBagOD} \rfloor$$

$$\mathsf{CoinBag} : \mathsf{Coin} \to \mathsf{Set}$$
$$\mathsf{CoinBag} \;=\; \mu\ \mathit{CoinBagD}$$

The base functor for CoinBag is

$$\mathscr{F}\ \mathit{CoinBagD} : (\mathsf{Coin} \to \mathsf{Set}) \to (\mathsf{Coin} \to \mathsf{Set})$$
$$\mathscr{F}\ \mathit{CoinBagD}\ X\ c \;=$$
$$\Sigma\ \mathsf{LTag}\ (\lambda\{\text{`}\mathsf{nil} \to \top; \text{`}\mathsf{cons} \to \Sigma[d : \mathsf{Coin}]\ (d \leqslant_{\mathsf{C}} c) \times X\ d\})$$

The total value of a coin bag is the sum of the values of the coins in the bag, which is computed by a (functional) fold:

$$\mathit{total\text{-}value\text{-}alg} : \mathscr{F}\ \mathit{CoinBagD}\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}$$
$$\mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{nil}\ \ ,\_\ \ \ ) \;=\; 0$$
$$\mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{cons}, d, \_, n) \;=\; \mathit{value}\ d + n$$

$$\mathit{total\text{-}value} : \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}$$
$$\mathit{total\text{-}value} \;=\; \mathit{fold}\ \mathit{total\text{-}value\text{-}alg}$$

and the number of coins in a coin bag is also computed by a fold:

$$\mathit{size\text{-}alg} : \mathscr{F}\ \mathit{CoinBagD}\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}$$
$$\mathit{size\text{-}alg}\ (\text{`}\mathsf{nil}\ \ ,\_\ \ \ ) \;=\; 0$$
$$\mathit{size\text{-}alg}\ (\text{`}\mathsf{cons}, \_, \_, n) \;=\; 1 + n$$

$$\mathit{size} : \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}$$
$$\mathit{size} \;=\; \mathit{fold}\ \mathit{size\text{-}alg}$$

The specification of the minimum coin change problem can now be written as

$$\mathit{min\text{-}coin\text{-}change} : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathsf{CoinBag}$$
$$\mathit{min\text{-}coin\text{-}change} \;=$$
$$\mathit{min}\ (\mathit{fun}\ \mathit{size}^\circ \cdot \mathit{leq} \cdot \mathit{fun}\ \mathit{size}) \cdot \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value}^\circ)$$

where $\mathit{leq} \;=\; \mathit{geq}^\circ : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$ maps a natural number $n$ to any natural number that is at most $n$. Intuitively, given an input $n : \mathsf{Nat}$, the relation $\mathit{fun}\ \mathit{total\text{-}value}^\circ$ computes an arbitrary coin bag whose total value is $n$, so *min-coin-change* first computes the set of all such coin bags and then chooses from the set a coin bag whose size is smallest. Our goal, then, is to write a functional program $f : \mathit{const}\ \mathsf{Nat} \rightrightarrows \mathsf{CoinBag}$ such that $\mathit{fun}\ f \subseteq \mathit{min\text{-}coin\text{-}change}$, and then $f\ \mathsf{5p} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p}$ would be a solution — note that the type CoinBag 5p contains all coin bags, since 5p is the largest denomination and hence a trivial upper bound on the content of bags. Of course, we may guess what $f$ should look like, but its correctness proof is much harder. Can we construct the program and its correctness proof in a more manageable way?

***The plan.*** In traditional relational program derivation, we would attempt to refine *min-coin-change* to some simpler relational program and then to an executable functional program by applying algebraic laws and theorems. With algebraic ornamentation, however, there is a new possibility: if we can derive that, for some algebra $R : \mathscr{F}\ \mathit{CoinBagD}\ (\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathit{const}\ \mathsf{Nat}$,

$$(\!|R|\!)^\circ \subseteq \mathit{min\text{-}coin\text{-}change} \tag{7}$$

then we can manufacture a new datatype

$$\mathit{GreedySolutionOD} : \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathsf{proj}_1\ \mathit{CoinBagD}$$
$$\mathit{GreedySolutionOD} \;=\; \mathit{algOrn}\ \mathit{CoinBagD}\ R$$

$$\mathsf{GreedySolution} : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set}$$
$$\mathsf{GreedySolution}\ c\ n \;=\; \mu\ \lfloor \mathit{GreedySolutionOD} \rfloor\ (c, n)$$

and construct a function of type

$$\mathit{greedy} : (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedySolution}\ c\ n$$

from which we can assemble a solution

$$\mathit{sol} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p}$$
$$\mathit{sol} \;=\; \mathit{forget}\ \lceil \mathit{GreedySolutionOD} \rceil \circ \mathit{greedy}\ \mathsf{5p}$$

The program *sol* satisfies the specification because of the following argument: For any $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$, by (6) we have

$$\mathsf{GreedySolution}\ c\ n \;\cong\; \Sigma[b : \mathsf{CoinBag}\ c]\ (\!|R|\!)\ b\ n$$

In particular, since the first half of the left-to-right direction of the isomorphism is $\mathit{forget}\ \lceil \mathit{GreedySolutionOD} \rceil$, we have

$$(\!|R|\!)\ (\mathit{forget}\ \lceil \mathit{GreedySolutionOD} \rceil\ g)\ n$$

for any $g : \mathsf{GreedySolution}\ c\ n$. Substituting $g$ by $\mathit{greedy}\ \mathsf{5p}\ n$, we get

$$(\!|R|\!)\ (\mathit{sol}\ n)\ n$$

which implies, by (7),

$$\mathit{min\text{-}coin\text{-}change}\ n\ (\mathit{sol}\ n)$$

i.e., *sol* satisfies the specification. Thus all we need to do to solve the minimum coin change problem is (i) refine the specification *min-coin-change* to the converse of a fold, i.e., find the algebra $R$ in (7), and (ii) construct the internalist program *greedy*.

***Refining the specification.*** The key to refining *min-coin-change* to the converse of a fold lies in the following version of the *Greedy Theorem*, which is essentially a specialisation of Bird and de Moor's Theorem 10.1 [4]: Let $D : \mathsf{Desc}\ I$ be a description, $R : \mu\ D \rightsquigarrow \mu\ D$ a preorder, and $S : \mathscr{F}\ D\ X \rightsquigarrow X$ an algebra. Consider the specification

$$\mathit{min}\ R \cdot \Lambda\ ((\!|S|\!)^\circ)$$

That is, given an input value $x : X\ i$ for some $i : I$, we choose a minimum under $R$ among all those elements of $\mu\ D\ i$ that computes to $x$ through $(\!|S|\!)$. The Greedy Theorem states that, if the initial algebra $\alpha = \mathit{fun}\ \mathsf{con} : \mathscr{F}\ D\ (\mu\ D) \rightsquigarrow \mu\ D$ is monotonic on $R$ (where $\mathsf{con} : \mathscr{F}\ D\ (\mu\ D) \rightrightarrows \mu\ D$ is the datatype-generic constructor), i.e.,

$$\alpha \cdot \mathscr{R}\ D\ R \subseteq R \cdot \alpha$$

and there is a relation (ordering) $Q : \mathscr{F}\ D\ X \rightsquigarrow \mathscr{F}\ D\ X$ such that the *greedy condition*

$$\alpha \cdot \mathscr{R}\ D\ ((\!|S|\!)^\circ) \cdot (Q \cap (S^\circ \cdot S))^\circ \subseteq R^\circ \cdot \alpha \cdot \mathscr{R}\ D\ ((\!|S|\!)^\circ)$$

is satisfied, then we have

$$(\!|\ (\mathit{min}\ Q \cdot \Lambda\ (S^\circ))^\circ\ |\!)^\circ \subseteq \mathit{min}\ R \cdot \Lambda\ ((\!|S|\!)^\circ)$$

Here we offer an intuitive explanation of the Greedy Theorem, but the theorem admits an elegant calculational proof, which can be faithfully reprised in Agda. The monotonicity condition states that if $ds : \mathscr{F}\ D\ (\mu\ D)\ i$ for some $i : I$ is better than $ds' : \mathscr{F}\ D\ (\mu\ D)\ i$ under $\mathscr{R}\ D\ R$, i.e., $ds$ and $ds'$ are equal except that the recursive positions of $ds$ are all better than the corresponding recursive positions of $ds'$ under $R$, then $\mathsf{con}\ ds : \mu\ D\ i$ would be better than $\mathsf{con}\ ds' : \mu\ D\ i$ under $R$. This implies that, when solving the optimisation problem, better solutions to subproblems would lead to a better solution to the original problem, so the *principle of optimality* applies, i.e., to reach an optimal solution it suffices to find

optimal solutions to subproblems, and we are entitled to use the converse of a fold to find optimal solutions recursively. The greedy condition further states that there is an ordering $Q$ on the ways of decomposing the problem which has significant influence on the quality of solutions: Suppose that there are two decompositions $xs$ and $xs' : \mathscr{F}\ D\ X\ i$ of some problem $x : X\ i$ for some $i : I$, i.e., both $xs$ and $xs'$ are in $S^\circ\ x : \mathscr{P}\ (\mathscr{F}\ D\ X\ i)$, and assume that $xs$ is better than $xs'$ under $Q$. Then for any solution resulting from $xs'$ (computed by $\alpha \cdot \mathscr{R}\ D\ ((\!|S|\!)^\circ)$) there always exists a better solution resulting from $xs$, so ignoring $xs'$ would only rule out worse solutions. The greedy condition thus guarantees that we will arrive at an optimal solution by always choosing the best decomposition, which is done by $min\ Q \cdot \Lambda\ (S^\circ) : X \rightsquigarrow \mathscr{F}\ D\ X$.

Back to the coin changing problem: By *fun-preserves-fold*, the specification *min-coin-change* is equivalent to

$$min\ (fun\ size^\circ \cdot leq \cdot fun\ size) \cdot \Lambda\ ((\!|fun\ total\text{-}value\text{-}alg|\!)^\circ)$$

which matches the form of the generic specification given in the Greedy Theorem, so we try to discharge the two conditions of the theorem. The monotonicity condition reduces to monotonicity of *fun size-alg* on *leq*, and can be easily proved either by relational calculation or pointwise reasoning. As for the greedy condition, an obvious choice for $Q$ is an ordering that leads us to choose the largest possible denomination, so we go for

$$\begin{aligned}
&Q : \mathscr{F}\ CoinBagD\ (const\ \mathsf{Nat}) \rightsquigarrow \mathscr{F}\ CoinBagD\ (const\ \mathsf{Nat})\\
&Q\ (\text{`}\mathsf{nil}\quad, \_\quad) = return\ (\text{`}\mathsf{nil}, \mathsf{tt})\\
&Q\ (\text{`}\mathsf{cons}, d, _) =\\
&\quad (_ \leqslant_{\mathsf{C}} _\ d) \gg\!= \lambda e \to any \gg\!= \lambda r \to return\ (\text{`}\mathsf{cons}, e, r)
\end{aligned}$$

where, in the cons case, the output is required to be also a cons node, and the coin at its head position must be one that is no smaller than the coin $d$ at the head position of the input. It is non-trivial to prove the greedy condition by relational calculation. Here we offer instead a brute-force yet conveniently expressed case analysis by dependent pattern matching, which also serves as an example of algebraic ornamentation. Define a new datatype $\mathsf{CoinBag}' : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}$ by composing two algebraic ornaments on *CoinBagD* in parallel:

$$\begin{aligned}
&CoinBag'OD : \mathsf{OrnDesc}\ (\mathsf{proj}_1 \bowtie \mathsf{proj}_1)\ pull\ CoinBagD\\
&CoinBag'OD = \lceil algOrn\ CoinBagD\ (fun\ total\text{-}value\text{-}alg) \rceil \otimes\\
&\qquad\qquad \lceil algOrn\ CoinBagD\ (fun\ size\text{-}alg) \rceil\\
&\mathsf{CoinBag}' : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathsf{CoinBag}'\ c\ n\ l = \mu\ \lfloor CoinBag'OD \rfloor\ (\mathsf{ok}\ (c,n), \mathsf{ok}\ (c,l))
\end{aligned}$$

By (3), (5), and *fun-preserves-fold*, $\mathsf{CoinBag}'$ is characterised by the isomorphisms

$$\begin{aligned}
\mathsf{CoinBag}'\ c\ n\ l \cong\ &\Sigma[b : \mathsf{CoinBag}\ c]\\
&(total\text{-}value\ b \equiv n) \times (size\ b \equiv l) \qquad (8)
\end{aligned}$$

for all $c : \mathsf{Coin}$, $n : \mathsf{Nat}$, and $l : \mathsf{Nat}$. Hence a coin bag of type $\mathsf{CoinBag}'\ c\ n\ l$ contains $l$ coins that are no larger than $c$ and sum up to $n$ pence. We can give the following types to the two constructors of $\mathsf{CoinBag}'$:

$$\begin{aligned}
&\mathsf{bnil}' \quad : \forall\ \{c\} \to \mathsf{CoinBag}'\ c\ 0\ 0\\
&\mathsf{bcons}' : \forall\ \{c\ n\ l\} \to (d : \mathsf{Coin}) \to d \leqslant_{\mathsf{C}} c \to\\
&\qquad \mathsf{CoinBag}'\ d\ n\ l \to \mathsf{CoinBag}'\ c\ (value\ d + n)\ (1 + l)
\end{aligned}$$

The greedy condition then essentially reduces to this lemma:

$$\begin{aligned}
&greedy\text{-}lemma :\\
&\quad (c\ d : \mathsf{Coin}) \to c \leqslant_{\mathsf{C}} d \to\\
&\quad (m\ n : \mathsf{Nat}) \to value\ c + m \equiv value\ d + n \to\\
&\quad (l : \mathsf{Nat})\ (b : \mathsf{CoinBag}'\ c\ m\ l) \to\\
&\quad \Sigma[l' : \mathsf{Nat}]\ \mathsf{CoinBag}'\ d\ n\ l' \times (l' \leqslant l)
\end{aligned}$$

That is, given a problem (i.e., a value to be represented by coins), if $c : \mathsf{Coin}$ is a choice of decomposition (i.e., the first coin used) no better than $d : \mathsf{Coin}$ (recall that we prefer larger denominations), and $b : \mathsf{CoinBag}'\ c\ m\ l$ is a solution of size $l$ to the remaining subproblem $m$ resulting from choosing $c$, then there is a solution to the remaining subproblem $n$ resulting from choosing $d$ whose size $l'$ is no greater than $l$. We define two *views* [16] — or "customised pattern matching" — to aid the analysis. The first view analyses a proof of $c \leqslant_{\mathsf{C}} d$ and exhausts all possibilities of $c$ and $d$,

$$\begin{aligned}
&\textbf{data}\ \mathsf{CoinOrderedView} : \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{1p1p} : \mathsf{CoinOrderedView\ 1p\ 1p}\\
&\quad \mathsf{1p2p} : \mathsf{CoinOrderedView\ 1p\ 2p}\\
&\quad \mathsf{1p5p} : \mathsf{CoinOrderedView\ 1p\ 5p}\\
&\quad \mathsf{2p2p} : \mathsf{CoinOrderedView\ 2p\ 2p}\\
&\quad \mathsf{2p5p} : \mathsf{CoinOrderedView\ 2p\ 5p}\\
&\quad \mathsf{5p5p} : \mathsf{CoinOrderedView\ 5p\ 5p}\\
&view\text{-}ordered\text{-}coin :\\
&\quad (c\ d : \mathsf{Coin}) \to c \leqslant_{\mathsf{C}} d \to \mathsf{CoinOrderedView}\ c\ d
\end{aligned}$$

where the covering function *view-ordered-coin* is written by standard pattern matching on $c$ and $d$. The second view analyses some $b : \mathsf{CoinBag}'\ c\ n\ l$ and exhausts all possibilities of $c$, $n$, $l$, and the first coin in $b$ (if any). The view datatype $\mathsf{CoinBag}'\mathsf{View}$ is shown in Figure 2, and the covering function

$$\begin{aligned}
&view\text{-}CoinBag' :\\
&\quad \forall\ \{c\ n\ l\}\ (b : \mathsf{CoinBag}'\ c\ n\ l) \to \mathsf{CoinBag}'\mathsf{View}\ b
\end{aligned}$$

is again written by standard pattern matching. Given these two views, *greedy-lemma* can be split into eight cases by first exhausting all possibilities of $c$ and $d$ with *view-ordered-coin* and then analysing the content of $b$ with *view-CoinBag'*. Figure 3 shows the case-split tree generated semi-automatically by Agda; the detail is explained as follows:

- At goal 0 (and, similarly, goals 3 and 7), the input bag is $b : \mathsf{CoinBag}'\ \mathsf{1p}\ n\ l$, and we should produce a $\mathsf{CoinBag}'\ \mathsf{1p}\ n\ l'$ for some $l' : \mathsf{Nat}$ such that $l' \leqslant l$. This is easy because $b$ itself is a suitable bag.
- At goal 1 (and, similarly, goals 2, 4, and 5), the input bag is of type $\mathsf{CoinBag}'\ \mathsf{1p}\ (1 + n)\ l$, i.e., the coins in the bag are no larger than 1p and the total value is $1 + n$. The bag must contain 1p as its first coin; let the rest of the bag be $b : \mathsf{CoinBag}'\ \mathsf{1p}\ n\ l''$. At this point Agda can deduce that $l$ must be $1 + l''$. Now we can return $b$ as the result after the upper bound on its coins is relaxed from 1p to 2p, which is done by
  $$\begin{aligned}
  &relax : \forall\ \{c\ n\ l\}\ (b : \mathsf{CoinBag}'\ c\ n\ l) \to\\
  &\qquad \forall\ \{d\} \to c \leqslant_{\mathsf{C}} d \to \mathsf{CoinBag}'\ d\ n\ l
  \end{aligned}$$
- The remaining goal 6 is the most interesting one: The input bag has type $\mathsf{CoinBag}'\ \mathsf{2p}\ (3 + n)\ l$, which in this case contains two 2-pence coins, and the rest of the bag is $b : \mathsf{CoinBag}'\ \mathsf{2p}\ k\ l''$. Agda deduces that $n$ must be $1 + k$ and $l$ must be $2 + l''$. We thus need to add a penny to $b$ to increase its total value to $1 + k$, which is done by
  $$\begin{aligned}
  &add\text{-}penny :\\
  &\quad \forall\ \{c\ n\ l\} \to \mathsf{CoinBag}'\ c\ n\ l \to \mathsf{CoinBag}'\ c\ (1 + n)\ (1 + l)
  \end{aligned}$$
  and relax the bound of *add-penny b* from 2p to 5p.

Throughout the proof, Agda is able to keep track of the total value and the size of bags and make deductions, so the case analysis is done with little overhead. The greedy condition can then be discharged by pointwise reasoning, using (8) to interface with *greedy-lemma*. We conclude that the Greedy Theorem is applicable, and obtain

$$(\!|(min\ Q \cdot \Lambda\ (fun\ total\text{-}value\text{-}alg^\circ))^\circ|\!)^\circ \subseteq min\text{-}coin\text{-}change$$

We have thus found the algebra

$$R = (min\ Q \cdot \Lambda\ (fun\ total\text{-}value\text{-}alg^\circ))^\circ$$

which will help us to construct the final internalist program.

```
data CoinBag′View : {c : Coin} {n : Nat} {l : Nat} → CoinBag′ c n l → Set where
  empty : {c : Coin} → CoinBag′View {c} {0} {0} bnil′
  1p1p  : {m l : Nat} {lep : 1p ⩽C 1p} (b : CoinBag′ 1p m l) → CoinBag′View {1p} {1 + m} {1 + l} (bcons′ 1p lep b)
  1p2p  : {m l : Nat} {lep : 1p ⩽C 2p} (b : CoinBag′ 1p m l) → CoinBag′View {2p} {1 + m} {1 + l} (bcons′ 1p lep b)
  2p2p  : {m l : Nat} {lep : 2p ⩽C 2p} (b : CoinBag′ 2p m l) → CoinBag′View {2p} {2 + m} {1 + l} (bcons′ 2p lep b)
  1p5p  : {m l : Nat} {lep : 1p ⩽C 5p} (b : CoinBag′ 1p m l) → CoinBag′View {5p} {1 + m} {1 + l} (bcons′ 1p lep b)
  2p5p  : {m l : Nat} {lep : 2p ⩽C 5p} (b : CoinBag′ 2p m l) → CoinBag′View {5p} {2 + m} {1 + l} (bcons′ 2p lep b)
  5p5p  : {m l : Nat} {lep : 5p ⩽C 5p} (b : CoinBag′ 5p m l) → CoinBag′View {5p} {5 + m} {1 + l} (bcons′ 5p lep b)
```

**Figure 2.** The view datatype on CoinBag′.

```
greedy-lemma : (c d : Coin) → c ⩽C d → (m n : Nat) → value c + m ≡ value d + n →
               (l : Nat) (b : CoinBag′ c m l) → Σ[ l′ : Nat ] CoinBag′ d n l′ × (l′ ⩽ l)
greedy-lemma c  d  c⩽d m  n  eq   l  b with view-ordered-coin c d c⩽d
greedy-lemma .1p .1p _  .n n  refl l  b CoinBag′ 1p n l | 1p1p = { Σ[ l′ : Nat ] CoinBag′ 1p n l′ × (l′ ⩽ l) }0
greedy-lemma .1p .2p _  .(1 + n) n  refl l  b | 1p2p with view-CoinBag′ b
greedy-lemma .1p .2p _  .(1 + n) n  refl .(1 + l″) _ | 1p2p | 1p1p {.n} {l″} b CoinBag′ 1p n l″ =
                                                { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ⩽ 1 + l″) }1
greedy-lemma .1p .5p _  .(4 + n) n  refl l  b | 1p5p with view-CoinBag′ b
greedy-lemma .1p .5p _  .(4 + n) n  refl _  _ | 1p5p | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _  .(4 + n) n  refl _  _ | 1p5p | 1p1p _ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _  .(4 + n) n  refl _  _ | 1p5p | 1p1p _ | 1p1p _ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _  .(4 + n) n  refl .(4 + l″) _ | 1p5p | 1p1p _ | 1p1p _ | 1p1p _ | 1p1p {.n} {l″} b CoinBag′ 1p n l″ =
                                                { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ⩽ 4 + l″) }2
greedy-lemma .2p .2p _  .n n  refl l  b CoinBag′ 2p n l | 2p2p = { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ⩽ l) }3
greedy-lemma .2p .5p _  .(3 + n) n  refl l  b | 2p5p with view-CoinBag′ b
greedy-lemma .2p .5p _  .(3 + n) n  refl _  _ | 2p5p | 1p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _  .(3 + n) n  refl _  _ | 2p5p | 1p2p _ | 1p1p b with view-CoinBag′ b
greedy-lemma .2p .5p _  .(3 + n) n  refl .(3 + l″) _ | 2p5p | 1p2p _ | 1p1p _ | 1p1p {.n} {l″} b CoinBag′ 1p n l″ =
                                                { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ⩽ 3 + l″) }4
greedy-lemma .2p .5p _  .(3 + n) n  refl _  _ | 2p5p | 2p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _  .(3 + n) n  refl .(2 + l″) _ | 2p5p | 2p2p _ | 1p2p {.n} {l″} b CoinBag′ 2p n l″ =
                                                { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ⩽ 2 + l″) }5
greedy-lemma .2p .5p _  .(4 + k) .(1 + k) refl .(2 + l″) _ | 2p5p | 2p2p _ | 2p2p {k} {l″} b CoinBag′ 2p k l″ =
                                                { Σ[ l′ : Nat ] CoinBag′ 5p (1 + k) l′ × (l′ ⩽ 2 + l″) }6
greedy-lemma .5p .5p _  .n n  refl l  b CoinBag′ 5p n l | 5p5p = { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ⩽ l) }7
```

**Figure 3.** Cases of *greedy-lemma*, generated semi-automatically by Agda's interactive case-split mechanism. Shown in the (shaded) interaction points are their goal types, and the types of some pattern variables are shown in subscript beside them.

***Constructing the internalist program.*** As planned, we synthesise a new datatype by ornamenting CoinBag using the algebra *R*:

```
GreedySolutionOD : OrnDesc (Coin × Nat) proj1 CoinBagD
GreedySolutionOD = algOrn CoinBagD R

GreedySolution : Coin → Nat → Set
GreedySolution c n = μ ⌊GreedySolutionOD⌋ (c , n)
```

The two constructors of GreedySolution can be given the following types:

```
gnil : ∀ {c n} →
       total-value-alg (‘nil , tt) ≡ n →
       (∀ ns → total-value-alg ns ≡ n → Q ns (‘nil , tt)) →
       GreedySolution c n

gcons :
  ∀ {c n} → (d : Coin) (d⩽c : d ⩽C c) →
  ∀ {n′} → total-value-alg (‘cons , d , d⩽c , n′) ≡ n →
       (∀ ns → total-value-alg ns ≡ n → Q ns (‘cons , d , d⩽c , n′)) →
       GreedySolution d n′ → GreedySolution c n
```

Before we proceed to construct the internalist program

```
greedy : (c : Coin) (n : Nat) → GreedySolution c n
```

let us first simplify the two constructors of GreedySolution. Each of the two constructors has two additional proof obligations coming from the algebra *R*: For gnil, since *total-value-alg* (‘nil , tt) reduces to 0, we may just specialise *n* to 0 and discharge the equality proof obligation. For the second proof obligation, *ns* is necessarily (‘nil , tt) if *total-value-alg ns* ≡ 0, and indeed *Q* maps (‘nil , tt) to (‘nil , tt), so the second proof obligation can be discharged as well. We thus obtain a simpler constructor defined using gnil:

```
gnil′ : ∀ {c} → GreedySolution c 0
```

For gcons, again since *total-value-alg* (‘cons , *d* , *d*⩽*c* , *n*′) reduces to *value d* + *n*′, we may just specialise *n* to *value d* + *n*′ and discharge the equality proof obligation. For the second proof obliga-

tion, any *ns* that satisfies *total-value-alg ns* $\equiv$ *value* $d + n'$ must be of the form $(\text{`cons}, e, e \leqslant c, m')$ for some $e : \mathsf{Coin}$, $e \leqslant c : e \leqslant_{\mathsf{C}} c$, and $m' : \mathsf{Nat}$ since the right-hand side *value* $d + n'$ is nonzero, and $Q$ maps *ns* to $(\text{`cons}, d, d \leqslant c, n')$ if $e \leqslant_{\mathsf{C}} d$, so $d$ should be the largest "usable" coin if this proof obligation is to be discharged. We say that $d : \mathsf{Coin}$ is *usable* with respect to some $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$ if $d$ is bounded above by $c$ and can be part of a solution to the problem for $n$ pence:

```
UsableCoin : Nat → Coin → Coin → Set
UsableCoin n c d =
  (d ⩽C c) × (Σ[n' : Nat] value d + n' ≡ n)
```

Now we can define a new constructor using gcons:

```
gcons' :
  ∀ {c} → (d : Coin) → d ⩽C c →
  ∀ {n'} →
  ((e : Coin) → UsableCoin (value d + n') c e → e ⩽C d) →
  GreedySolution d n' → GreedySolution c (value d + n')
```

which requires that $d$ is the largest usable coin with respect to $c$ and *value* $d + n'$. We are thus directed to implement a function *maximum-coin* that computes the largest usable coin with respect to any $c : \mathsf{Coin}$ and nonzero $n : \mathsf{Nat}$,

```
maximum-coin :
  (c : Coin) (n : Nat) → n > 0 →
  Σ [d : Coin] UsableCoin n c d ×
               ((e : Coin) → UsableCoin n c e → e ⩽C d)
```

which takes some theorem proving but is overall a typical Agda exercise in dealing with natural numbers and ordering. Now we can implement the greedy algorithm as the internalist program

```
greedy : (c : Coin) (n : Nat) → GreedySolution c n
greedy c n = <-rec P f n c
  where
    P : Nat → Set
    P n = (c : Coin) → GreedySolution c n
    f : (n : Nat) → ((n' : Nat) → n' < n → P n') → P n
    f  n          rec c with compare-with-zero n
    f .0          rec c | is-zero = gnil'
    f  n          rec c | above-zero n>z
                        with maximum-coin c n n>z
    f .(value d + n') rec c | above-zero n>z
                            | d, (d⩽c, n', refl), guc =
                              gcons' d d⩽c guc (rec n' { }8 d)
```

where the combinator

```
<-rec : (P : Nat → Set) →
        ((n : Nat) → ((n' : Nat) → n' < n → P n') → P n) →
        (n : Nat) → P n
```

is for well-founded recursion on $\_<\_$, and the function

```
compare-with-zero : (n : Nat) → ZeroView n
```

is a covering function for the view

```
data ZeroView : Nat → Set where
  is-zero    : ZeroView 0
  above-zero : {n : Nat} → n > 0 → ZeroView n
```

At goal 8, Agda deduces that $n$ is *value* $d + n'$ and demands that we prove $n' <$ *value* $d + n'$ in order to make the recursive call, which is easily discharged since *value* $d > 0$.

## 6. Discussion

The relational framework of this paper heavily borrows techniques from the AoPA library [18]. AoPA deals with non-dependently typed programs only, whereas to work with indexed datatypes we need to move to indexed families of relations; to work with the ornamental universe we parametrise the relational fold with a description, making it fully datatype-generic, whereas AoPA has only specialised versions for lists and binary trees; we defined $min\_ \cdot \Lambda\_$ as a single operator (which happens to be the *shrinking* operator proposed by Mu and Oliveira [17]) to avoid the struggle with predicativity that AoPA had. All of the above are not fundamental differences between our work and AoPA, though — the two differ essentially in methodology: In AoPA, dependent types merely provide a foundation on which relational program derivations can be expressed and checked by machines but the programs remain non-dependently typed, whereas in this paper relational program derivation is a tool for obtaining non-trivial inductive families that effectively guide program development as advertised as the strength of internalist dependently typed programming. In short, the focus of AoPA is on traditional relational program derivation (expressed in a dependently typed language), whereas our emphasis is on internalist programming (aided by relational program derivation).

Let us contemplate the roles played by internalist programming and relational program derivation in the case study in Section 5. As mentioned in Section 1, internalist programming provides us with a more informative syntax in which programs can encode more semantic information, including correctness proofs. We can thus write programs that directly explain their own meaning. An example is the final program for the greedy algorithm in Section 5, whose structure carries an implicit inductive proof; the program constructs not merely a list of coins, but a bag of coins chosen according to a particular optimisation strategy (i.e., *min* $Q \cdot \Lambda$ (*fun total-value-alg* $^\circ$)). Internalist programming alone has limited power, however, because internalist programs share structure with their correctness proofs, but we cannot expect to have such coincidences all the time. In particular, there is no hope of integrating a correctness proof into a program when the structure of the proof is more complicated than that of the program. For example, we can not imagine how to integrate a correctness proof for the full specification of the minimum coin change problem into a program for the greedy algorithm. In essence, we have two kinds of proofs to deal with: the first kind follow program structure and can be embedded in internalist programs, and the second kind are general proofs of full specifications, which do not necessarily follow program structure and which fall outside the scope of internalism. To exploit the power of internalism as much as possible, we need ways to reduce the second kind of proof obligations to the first kind — note that such reduction involves not only constructing proof transformations but also determining what internalist proofs are sufficient for establishing proofs of full specifications. It turns out that relational program derivation is exactly one way in which we can construct such proof transformations systematically from specifications. In relational program derivation, we identify important forms of relational programs (i.e., relational composition, recursion schemes, and various other combinators), and formulate algebraic laws and theorems in terms of these forms. By applying the laws and theorems, we massage a relational specification into a known form which corresponds to a proof obligation that can be expressed in an internalist type, enabling transition to internalist programming. For example, we now know that, by relational algebraic ornamentation, a relational fold can be turned into an inductive family for internalist programming. Thus, given a relational specification, we might seek to massage it into a relational fold when that possibility is pointed out by known laws and theorems (e.g., the Greedy Theorem). To sum up, we get a hybrid methodology that effectively leads us from relational specifications towards correct-by-construction programs, providing hints in the form of relational algebraic laws and theorems and internalist type information throughout the development.

There is work to be done to improve practicality of the methodology besides notational support, especially interoperability between internalist programming and relational program derivation. Relational program derivation was developed in a non-dependently typed setting, but we need new operators and theorems to deal with indexed datatypes. For example, the version of the Greedy Theorem in this paper works for the minimum coin change problem because, luckily, the problem is simple enough — given $n : \mathsf{Nat}$ we know that an optimal solution necessarily resides in $(\mathit{fun\ total\text{-}value}^{\circ})\ \{\mathsf{5p}\}\ n : \mathscr{P}(\mathsf{CoinBag\ 5p})$. But in general we do not know the index in the type of an optimal solution in advance; so we need to optimise across indices, which requires a non-trivial revision of the Greedy Theorem. We also need more mechanisms that allow us to make smooth transitions between relational program derivation and dependently typed programming, relational algebraic ornamentation being one example. Non-examples include (i) the wrapping of the dependently typed function *greedy-lemma* into a proof of the relational greedy condition, which is done by painstakingly dissecting the proof terms generated by relational operators and is very tedious since the proof terms are difficult to understand, and (ii) the definition of the classifying algebra, which can hardly be manipulated by relational algebraic methods, making the completeness theorem much less useful in practice. These difficulties arise because the encoding of relational program derivation and the constructions for internalist programming were conceived separately and did not take the other side into account. Most likely, an integrated design is needed to narrow the gap.

***Relationship with existing work on ornaments.*** There is a dual to the completeness theorem in Section 4: every relational algebra is isomorphic (in a suitable category of relational algebras) to the classifying algebra for the algebraic ornament using the algebra. More precisely: Let $D : \mathsf{Desc}\ I$ be a description and $R : \mathscr{F}\ D\ X \rightsquigarrow X$ an algebra (where $X : I \to \mathsf{Set}$). There is a family of isomorphisms between $X\ i$ and $\mathsf{InvImage}\ \mathsf{proj}_1\ i$ for every $i : I$ (where $\mathsf{proj}_1 : \Sigma\ I\ X \to I$); call the forward direction of this family of isomorphisms $h : X \rightrightarrows \mathsf{InvImage}\ \mathsf{proj}_1$. Then we have

$$\mathit{fun}\ h \cdot R \simeq \mathit{clsAlg}\ \lceil \mathit{algOrn}\ D\ R \rceil \cdot \mathscr{R}\ D\ (\mathit{fun}\ h)$$

Together with the completeness theorem, we see that *algOrn* and *clsAlg* are, in some sense, inverse to each other up to isomorphism. This suggests that if we properly set up descriptions and ornaments as a category, *algOrn* and *clsAlg* can be extended to functors between the category of $D$-algebras and the slice category of ornaments over $D$, and their inverse relationship can be extended to an equivalence between the two categories. A preliminary attempt failed, however, due to a defect in the design of the ornament language in our and Dagand and McBride's work [7, 12] which prevents ornaments from being formed between obviously structurally related datatypes. We plan to make the ornament language closer to *containers* [1, 2], in the hope that, in particular, we can establish the equivalence of categories sketched above, which would show that ornaments and relational algebras are essentially the same thing.

Finally, there is a practical concern: algebraic ornamentation does not make full use of index-first datatypes and can produce redundant representations. (For example, the vector datatype produced by algebraic ornamentation is the one containing equality proof obligations rather than the optimised one, $\mathsf{Vec}'$, in Section 2.) With algebraic ornamentation, we determine what the targeted index can be from the data, but for index-first datatypes the natural notion is *coalgebraic* ornamentation, because we determine what data can be offered from the targeted index. This, however, is a further justification of the move to relational algebras, because relational algebras are bidirectional and hence coalgebras coincide with the converses of algebras. In future work, we will attempt to construct a suitable universe for relational algebras with which a relational algebra (or part of it) can be marked as coalgebraic, so its coalgebraic structure can be exploited by algebraic ornamentation to generate efficient index-first representations.

## Acknowledgements

The first author is supported by a University of Oxford Clarendon Fund Scholarship, and both authors by the UK Engineering and Physical Sciences Research Council project *Reusability and Dependent Types* (EP/G034516/1). We are grateful to Conor McBride for the inspiring discussions at the *Reusability and Dependent Types* project meetings and for generous suggestions on this paper, and to the anonymous reviewers for their valuable comments.

## References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] T. Altenkirch and P. Morris. Indexed containers. In *Logic in Computer Science*, LICS'09, pages 277–285. IEEE, 2009.

[3] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer-Verlag, 2010.

[4] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[5] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, 2004.

[6] J. Chapman, P.-É. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP'10, pages 3–14. ACM, 2010.

[7] P.-É. Dagand and C. McBride. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM, 2012.

[8] E. W. Dijkstra. Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612, 1974.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4): 440–465, 1994.

[11] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1): 146–160, 2001.

[12] H.-S. Ko and J. Gibbons. Modularising inductive families. *Progress in Informatics*, 10:65–88, 2013. doi:10.2201/NiiPi.2013.10.5.

[13] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[14] C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170, 2004.

[15] C. McBride. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*, 2011.

[16] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[17] S.-C. Mu and J. N. Oliveira. Programming from Galois connections. *Journal of Logic and Algebraic Programming*, 81(6):680–704, 2012.

[18] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of Programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579, 2009.

[19] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[20] U. Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.