REFINING INDUCTIVE TYPES

ROBERT ATKEY, PATRICIA JOHANN, AND NEIL GHANI

University of Strathclyde, UK
e-mail address: Robert.Atkey@cis.strath.ac.uk

University of Strathclyde, UK
e-mail address: Patricia.Johann@cis.strath.ac.uk

University of Strathclyde, UK
e-mail address: Neil.Ghani@cis.strath.ac.uk

Abstract. Dependently typed programming languages allow sophisticated properties of data to be expressed within the type system. Of particular use in dependently typed programming are indexed types that refine data by computationally useful information. For example, the $\mathbb{N}$-indexed type of vectors refines lists by their lengths. Other data types may be refined in similar ways, but programmers must produce purpose-specific refinements on an *ad hoc* basis, developers must anticipate which refinements to include in libraries, and implementations must often store redundant information about data and their refinements. In this paper we show how to generically derive inductive characterisations of refinements of inductive types, and argue that these characterisations can alleviate some of the aforementioned difficulties associated with *ad hoc* refinements. Our characterisations also ensure that standard techniques for programming with and reasoning about inductive types are applicable to refinements, and that refinements can themselves be further refined.

1. Introduction

One of the key aims of current research in functional programming is to reduce the *semantic gap* between what programmers know about computational entities and what the types of those entities can express. One particularly promising approach to closing this gap is to *index* types by extra information that can be used to express properties of their elements. For example, most functional languages support a standard list data type parameterised over the type of the data lists contain, but for some applications it is also convenient to be able to state the length of a list in its type. This makes it possible, for instance, to ensure that the list argument to the `tail` function has non-zero length — i.e., is non-empty — and that the lengths of the two list arguments to `zip` are the same. Without this kind of static enforcement of preconditions, functions such as these must be able to signal erroneous arguments — perhaps using an error monad, or a built-in exception facility — and their clients must be able to handle the cases in which an error is raised.

A data type that equips each list with its length can be defined in the dependently typed language Agda 2 [34] using the following declaration:

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

```
data Vector (B : Set) : Nat -> Set where
  nil  : Vector B zero
  cons : {n : Nat} -> B -> Vector B n -> Vector B (succ n)
```

This declaration[1] inductively defines, for each choice of element type `B`, a data type `Vector B` that is indexed by natural numbers and has two constructors: `nil`, which constructs a vector of data with type `B` of length zero (here represented by the data constructor `zero` for the natural numbers), and `cons`, which constructs from an index `n`, an element of `B`, and a vector of data with type `B` of length `n`, a new vector of data with type `B` of length `n`+1 (here represented by the application `succ n` of the data constructor `succ` for the natural numbers to `n`). The inductive type `Vector B` can be used to define functions on lists with elements of type `B` that are "length-aware" in a way that functions processing data of standard list types cannot be. For example, it allows length-aware `tail` and `zip` functions to be given via the following Agda 2 types and definitions:

```
tail : {B : Set} -> {n : Nat} -> Vector B (succ n) -> Vector B n
tail (cons b bs) = bs

zip  : {B C : Set} -> {n : Nat} ->
         Vector B n -> Vector C n -> Vector (B x C) n
zip  nil         nil        = nil
zip (cons b bs) (cons c cs) = cons (b , c) (zip bs cs)
```

Examples such as these suggest that indexing types by computationally relevant information has great potential. However, for this potential to be realised we must better understand how indexed types can be constructed. Moreover, since we want to ensure that all the techniques that have been developed for structured programming with and principled reasoning about inductive types[2] — such as those championed in the Algebra of Programming [8] literature — are applicable to the resulting indexed types, we also want these types to be inductive. This paper therefore asks the following fundamental question:

> *Can elements of one inductive type be systematically augmented with computationally relevant information to construct an indexed inductive type that captures the computationally relevant information in their indices? If so, how?*

That is, how can we *refine* an inductive type to get a new type, called a *refinement*, that associates to each element of the original type its index, and how can we ensure that the resulting refinement is inductive?

1.1. **A Naive Solution.** One straightforward way to refine an inductive type is to use a refinement function to compute the index for each of its elements and then to associate these indices to their corresponding elements. To refine lists by their lengths, for example, we would start with the standard list data type, which has the following Agda 2 declaration[3]:

[1] The `{X : S}` notation indicates that there is an implicit parameter of type `S`, named `X`. When applying a function with an implicit argument, Agda 2 will attempt to infer a suitable value for it.

[2] Recall that an inductive data type is one that an be represented as the least fixed point μF of a functor F on a category suitable for interpreting the types in a language.

[3] Agda 2 allows overloading of constructor names, so we reuse the constructor names `nil` and `cons` from the `Vector` type defined above.

```
data List (B : Set) : Set where
  nil  : List B
  cons : B -> List B -> List B
```

We would then define the following function `length` by structural recursion on elements of `List B`:

```
length : {B : Set} -> List B -> Nat
length nil         = zero
length (cons _ bs) = succ (length bs)
```

From these we would construct the following refinement of lists by the function `length`, using a subset type:

$$\texttt{ListWithLength B n} \cong \{\texttt{bs} : \texttt{List B} \mid \texttt{length bs} = \texttt{n}\} \tag{1.1}$$

(alternatively, we could have also used a Σ-type to hold the list `bs` and the proof that `length bs` $=$ `n`.) Note that this construction is *global* in that both the data type and the collection of indices exist *a priori*, and the refinement is obtained by assigning, *post facto*, an appropriate index to each data type element. It also suffers from a serious drawback: the resulting refinement — `ListWithLength B` here — is not presented as an inductive type, so the naive solution is not a solution to the fundamental question posed above. (In addition, the refinement `ListWithLength B` does not obviously have anything to do with the type `Vector B`.) So the question remains: how do we get the inductive type `Vector B` from the inductive type `List B`?

1.2. **A Better Solution.** When the given refinement function is computed by structural recursion (i.e., by the fold) over the data type to be refined — as is the case for the function `length` above and is often the case in practice — then we can give an alternative construction of refinements that provides a comprehensive answer to the fundamental question raised above. In this case we can construct, for each inductive type μF and each F-algebra α whose fold computes the desired refinement function, a functor F^α whose least fixed point μF^α is the desired refinement. This construction is the central contribution of the paper. Our characterisation of the refinement of μF by α as the inductive type μF^α allows the entire arsenal of structured programming techniques based on initial algebras to be brought to bear on the resulting refinement. By contrast with the construction in (1.1) above, our characterisation is also *local*, in that the indices of recursive substructures are readily available *at the time a structurally recursive program is written*, rather than needing to be computed by inversion at run time from the index of the input data structure to the program.

For each functor F and F-algebra α, the functor F^α that we construct is intimately connected with the generic structural induction rule for the inductive type μF, as presented by Hermida and Jacobs [24] and by Ghani, Johann, and Fumex [22]. This is perhaps not surprising: structural induction proves properties of functions defined by structural recursion on elements of inductive types. If the values of such functions are abstracted into the indices of associated indexed inductive types, then their computation need no longer be performed during inductive proofs. In essence, work has been shifted away from computation and onto data. Refinement can thus be seen as supporting reasoning by structural induction "up to" the index of a term.

1.3. **The Structure of this Paper.** The remainder of this paper is structured as follows. In Section 2 we introduce inductive types and recall their representation as carriers of initial algebras of functors. We first recall that, for any functor F, the collection of F-algebras forms a category, and then give a key theorem due to Hermida and Jacobs [24] relating different F-algebras and, thereby, different refinements of μF. In Section 3 we define the fibrational framework for refinements with which we work in this paper, and introduce the important idea of the lifting of a functor. In Section 4 we show how liftings can be used to refine inductive types, prove the correctness of our construction of refinements, and illustrate our construction with some simple examples. In Section 5 we show how to refine inductive types that are themselves already indexed, thus extending our construction to allow refinement of the whole range of indexed inductive types available in dependently typed languages. In Section 6 we further extend our basic refinement technique to allow partial refinement, in which indexed types are constructed from inductive types not all of whose elements have indices. Our motivating example here is that of expressions that can fail to be well-typed. Indeed, we refine the type of possibly ill-typed expressions by a type checker to yield the indexed inductive type of well-typed expressions. In Section 7 we extend the basic notion of refinement in yet another direction to allow refinement by paramorphisms — also known as primitive recursive functions — and their generalisation zygomorphisms. Perhaps surprisingly, this takes us from the world of indexed inductive types to indexed induction-recursion, in which inductive types and recursive functions are defined simultaneously. In Section 8 we conclude and discuss related and future work.

Throughout this paper, we adopt a semantic approach based on category theory because it allows a high degree of abstraction and economy. More specifically, we develop our theory in the abstract setting of fibrations [26]. Nevertheless, we specialise to the families fibration over the category of sets in order to improve accessibility and give concrete intuitions; Section 3 gives the necessary definitions and background. Moreover, carefully using only the abstract structure of the families fibration allows us to expose crucial structure that might be lost were a specific programming notation to be used. This structure both simplifies our proofs and facilitates the iteration of our construction detailed in Section 5. It also highlights the commonalities between the various constructions we present. In particular, each of the refinement processes we discuss produces functors of the form $J \circ \hat{F}$, where $\hat{F}$ is the lifting of the functor F defining the data type μF to be refined. We are currently investigating whether this observation leads to a more general theory of refinement, as well as its potential use in structuring an implementation. A type-theoretic, rather than categorical, answer to the fundamental question this paper addresses has already been given by McBride [32] using his notion of ornaments for data types (see Section 8).

1.4. **Differences from the Previously Published Version.** This paper is a revised and expanded version of the FoSSaCS 2011 conference version [4]. Additional explanations have been provided throughout, examples have been expanded, and some of the material has been reordered for clarity. Section 2.2, which explains in more detail the connection between initial algebras and the indexed inductive types present in systems such as Agda 2, is entirely new. Section 7, which discusses the connection between refinement by zygomorphisms and indexed inductive-recursive definitions, is also completely new, and represents significant further development of our basic refinement technique.

2. Inductive Types and F-algebras

A data type is *inductive (in a category $\mathcal{C}$)* if it is the least fixed point μF of an endofunctor F on $\mathcal{C}$, in a sense to be made precise in Section 2.1 below. For example, if Set denotes the category of sets and functions, $\mathbb{Z}$ is the set of integers, and $+$ and $\times$ denote the coproduct and product, respectively, then $\mu F_{\texttt{Tree}}$ for the endofunctor $F_{\texttt{Tree}}X = \mathbb{Z} + X \times X$ on Set represents the following data type of binary trees with integer data at the leaves:

```
data Tree : Set where
    leaf : Integer -> Tree
    node : (Tree x Tree) -> Tree
```

2.1. F-algebras.

Our precise understanding of inductive types comes from the categorical notion of an F-algebra. If $\mathcal{C}$ is a category and F is an endofunctor on $\mathcal{C}$, then an *F-algebra* is a pair $(A, \alpha : FA \to A)$ comprising an object A of $\mathcal{C}$ and a morphism $\alpha : FA \to A$ in $\mathcal{C}$. The object A is called the *carrier* of the F-algebra, and the morphism α is called its *structure map*. We usually refer to an F-algebra solely by its structure map $\alpha : FA \to A$, since the carrier is present in the type of this map.

An *F-algebra morphism* from $\alpha : FA \to A$ to $\alpha' : FB \to B$ is a morphism $f : A \to B$ of $\mathcal{C}$ such that $f \circ \alpha = \alpha' \circ Ff$. An F-algebra $\alpha : FA \to A$ is *initial* if, for any F-algebra $\alpha' : FB \to B$, there exists a unique F-algebra morphism from α to α'. If it exists, the initial F-algebra is unique up to isomorphism, and Lambek's Lemma further ensures that the[4] initial F-algebra is an isomorphism. Its carrier is thus the least fixed point μF of F. We write $in_F : F(\mu F) \to \mu F$ for the initial F-algebra, and $(\!|\alpha|\!)_F : \mu F \to A$ for the unique morphism from $in_F : F(\mu F) \to \mu F$ to any F-algebra $\alpha : FA \to A$. We write $(\!|-|\!)$ for $(\!|-|\!)_F$ when F is clear from context. Of course, not all functors have initial algebras. For instance, the functor $FX = (X \to 2) \to 2$ on Set does not have an initial algebra.

In light of the above, the data type `Tree` can be interpreted as the carrier of the initial $F_{\texttt{Tree}}$-algebra. In functional programming terms, a function $\alpha : \mathbb{Z} + A \times A \to A$ is an $F_{\texttt{Tree}}$-algebra, and the function $(\!|\alpha|\!) : \texttt{Tree} \to A$ induced by the initiality property is exactly the application to α of the standard iteration function `fold` for trees (actually, the application of `fold` to an "unbundling" of α into replacement functions, one for each of $F_{\texttt{Tree}}$'s constructors). More generally, for each functor F, the function $(\!|-|\!)_F : (FA \to A) \to \mu F \to A$ is the standard iteration function for μF.

2.2. Indexed Inductive Types as F-Algebras.

Indexed types can be inductive, and this gives rise to the notion of an *indexed inductive type*. Such a type is also called an *inductive family* of types [18]. Indexed inductive types can be seen as initial F-algebras for endofunctors F on categories of indexed sets. For example, if B is a set of elements, then we can define a functor $F_{\texttt{Vector}_B}$ on the category of $\mathbb{N}$-indexed sets whose least fixed point represents the inductive data type `Vector B` from introduction. The two constructors `nil` and `cons` are reflected in the definition of $F_{\texttt{Vector}_B}$ as a coproduct, the individual arguments to each constructor are reflected as products within each summand of this coproduct, and the implicit equality constraints on the indices are reflected as explicit equality constraints.

[4] We identify isomorphic entities when convenient. When doing so, we write $=$ in place of $\cong$.

We define

$$\begin{array}{rcl} F_{\mathtt{Vector}_B} & : & (\mathbb{N} \to \mathrm{Set}) \to (\mathbb{N} \to \mathrm{Set}) \\ F_{\mathtt{Vector}_B}\ X & = & \lambda n.\{* \mid n = 0\} + \{(n_1 : \mathbb{N}, a : B, x : X n_1) \mid n = n_1 + 1\} \end{array}$$

where the notation $\{* \mid n = 0\}$ denotes the set $\{*\}$ when $n = 0$ and the empty set otherwise. The carrier of the initial algebra $in_{F_{\mathtt{Vector}_B}} : F_{\mathtt{Vector}_B}(\mu F_{\mathtt{Vector}_B}) \to \mu F_{\mathtt{Vector}_B}$ of this functor consists of the $\mathbb{N}$-indexed family $\mu F_{\mathtt{Vector}_B}$ of sets of vectors with elements from B, together with a function $in_{F_{\mathtt{Vector}_B}}$ that "bundles together" the constructors `nil` and `cons`. In Section 4.2 below we show how $F_{\mathtt{Vector}_B}$ can be *derived* from the functor $F_{\mathtt{List}_B}$ whose least fixed point is the inductive type of lists with elements from B, together with the algebra *lengthalg* whose fold is the standard length function on lists.

In general, X-indexed inductive types can be understood as initial algebras of functors $F : (X \to \mathrm{Set}) \to (X \to \mathrm{Set})$. In Section 3 below we will see how the collection of categories of indexed sets can be organised into the *families fibration*, in which we carry out the constructions giving rise to our framework for refinement.

2.3. **Categories of F-algebras.** If F is an endofunctor on $\mathcal{C}$, we write Alg_F for the category whose objects are F-algebras and whose morphisms are F-algebra morphisms between them. Identities and composition in Alg_F are taken directly from $\mathcal{C}$. The existence of initial F-algebras is equivalent to the existence of initial objects in the category Alg_F.

In Theorems 3.3 and 6.2 below, we will have an initial object in one category of algebras and want to show that applying a functor to it gives the initial object in another category of algebras. We will use adjunctions to do this. Recall that an *adjunction* $\mathcal{C} \underset{R}{\overset{L}{\leftarrows \bot \rightarrow}} \mathcal{D}$ between two categories $\mathcal{C}$ and $\mathcal{D}$ consists of a left adjoint functor L, a right adjoint functor R, and an isomorphism natural in A and X between the set $\mathcal{C}(LA, X)$ of morphisms in $\mathcal{C}$ from LA to X and the set $\mathcal{D}(A, RX)$ of morphisms in $\mathcal{D}$ from A to RX. We say that the functor L is *left adjoint* to R, and that the functor R is *right adjoint* to L, and we write $L \dashv R$. To lift adjunctions to categories of algebras, we will make much use of the following theorem of Hermida and Jacobs [24]:

Theorem 2.1. *If $F : \mathcal{C} \to \mathcal{C}$ and $G : \mathcal{D} \to \mathcal{D}$ are functors, $L \dashv R$, and $F \circ L \cong L \circ G$ is a natural isomorphism, then the adjunction $\mathcal{C} \underset{R}{\overset{L}{\leftarrows \bot \rightarrow}} \mathcal{D}$ lifts to an adjunction $\mathrm{Alg}_F \underset{R'}{\overset{L'}{\leftarrows \bot \rightarrow}} \mathrm{Alg}_G$.*

In the setting of the theorem, if G has an initial algebra, then so does F since left adjoints preserve colimits and in particular initial objects. To compute the initial F-algebra in concrete situations we need to know that $L'(k : GA \to A) = Lk \circ p_A$, where p is (one half of) the natural isomorphism between $F \circ L$ and $L \circ G$. Then the initial F-algebra is given by applying L' to the initial G-algebra, i.e., $in_F = L'(in_G)$, and hence $\mu F = L'(\mu G)$.

3. A Framework for Refinement

We develop our theoretical framework for refinement in the setting of fibrational models of extensional Martin-Löf type theory, which is a key theory underlying dependently typed programming. Since the concepts and terminology of fibrational category theory will not be familiar to most readers, we have taken care to formulate each of our definitions and main

theorems in the families fibration. The families fibration gives the archetypal semantics of Martin-Löf type theory, in which indexed types are interpreted directly as indexed sets. In this section we define the families fibration, and identify the parts of its structure that we require for the rest of the paper. As readers who are familiar with the categorical notion of fibration will observe, the terminology and structure that we identify comes from fibred category theory. We take care to identify the particular properties of the families fibration that are required for our results to hold, and refer to the literature for the formulation of these properties in the general setting.

3.1. **The Families Fibration.** As is customary, we model indexed types in the category Fam(Set). An object of Fam(Set) is a pair (A, P) comprising a set A and a function $P : A \to \mathrm{Set}$; such a pair is called a *family* of sets. We denote a family (A, P) as $P : A \to \mathrm{Set}$ when convenient, or simply as P when A can be inferred from context. A morphism $(f, f^{\sim}) : (A, P) \to (B, Q)$ of Fam(Set) is a pair of functions $f : A \to B$ and $f^{\sim} : \forall a.\ Pa \to Q(fa)$. From a programming perspective, a family (A, P) is an A-indexed type P, where Pa represents the collection of data with index a. An alternative, logical, view is that (A, P) is a predicate representing a property P of data of type A, and that Pa represents the collection of proofs that a has property P. When Pa is inhabited, P is said to *hold* for a. When Pa is empty, P is said *not to hold* for a. We will freely switch between the programming and logical interpretations of families when providing intuition for our formal development below.

The *families fibration* $U : \mathrm{Fam(Set)} \to \mathrm{Set}$ is the functor mapping each family (A, P) to A and each morphism $(f, f^{\sim})$ to f. The category Set is referred to as the *base category* of the families fibration and Fam(Set) is referred to as its *total category*. For each set A, the category $\mathrm{Fam(Set)}_A$ consists of families (A, P) and morphisms $(f, f^{\sim})$ between them such that $f = id_A$. Such a morphism is said to be a *vertical morphism*. Similarly, a *vertical natural transformation* is a natural transformation each of whose components is a vertical morphism. We say that an object or morphism in $\mathrm{Fam(Set)}_A$ is *over* A with respect to the families fibration, and call $\mathrm{Fam(Set)}_A$ the *fibre* of the families fibration over A. A function $f : A \to B$ contravariantly generates a *reindexing functor* $f^* : \mathrm{Fam(Set)}_B \to \mathrm{Fam(Set)}_A$ for the families fibration that maps (B, Q) to $(A, Q \circ f)$.

3.2. **Truth and Comprehension.** Each fibre $\mathrm{Fam(Set)}_A$ has a terminal object $(A, \lambda a : A.\ 1)$, where 1 is the canonical singleton set. In light of the logical reading of families above, this object is called the *truth predicate* for A. The mapping of objects to their truth predicates extends to a functor $\top : \mathrm{Set} \to \mathrm{Fam(Set)}$, called the *truth functor* for the families fibration. In addition, for each family (A, P) we can define the *comprehension* of (A, P), denoted $\{(A, P)\}$, to be $\Sigma a : A.Pa$, i.e., $\{(a, p) \mid a \in A, p \in Pa\}$. The comprehension $\{(A, P)\}$ packages elements $a \in A$ with proofs $p \in Pa$. The mapping of families to their comprehensions extends to a functor $\{-\} : \mathrm{Fam(Set)} \to \mathrm{Set}$, called the *comprehension functor* for the families fibration. Overall, we have the following pleasing collection of adjoint relationships:

$$\begin{array}{ccc} \mathrm{Fam(Set)} & & \\ {\scriptstyle U}\downarrow \dashv \top \dashv & \searrow & {\scriptstyle \{-\}} \\ \mathrm{Set} & \swarrow & \end{array} \tag{3.1}$$

The families fibration U is thus a *comprehension category with unit* [25, 26]. Like every comprehension category with unit, U supports a natural transformation $\pi : \{-\} \to U$ such that $\pi_{(A,P)}(a, p) = a$ for all (a, p) in $\{(A, P)\}$, projecting out the A component from a comprehension. In fact, U is a *full comprehension category with unit*, i.e., the functor from $\mathrm{Fam}(\mathrm{Set})$ to $\mathrm{Set}^{\to}$ induced by π is full and faithful. Here, $\mathrm{Set}^{\to}$ is the *arrow category* of Set. Its objects are morphisms of Set and its morphisms from $f : X \to Y$ to $f' : X' \to Y'$ are pairs (α_1, α_2) of morphisms in Set such that $f' \circ \alpha_1 = \alpha_2 \circ f$. Fullness means that the action of π on morphisms is surjective, and faithfulness means that it is injective. Fullness will be used in the proof of Theorem 5.1 below, when we consider refinements of indexed types.

3.3. **Indexed Coproducts.** For each function $f : A \to B$ and family (A, P), we can form the family $\Sigma_f(A, P) = (B, \lambda b.\ \Sigma_{a \in A}.\ (b = fa) \times Pa)$, called the *indexed coproduct of* (A, P) *along* f. The mapping of each family to its indexed coproduct along f extends to a functor $\Sigma_f : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_B$ which is left adjoint to the reindexing functor f^* for the families fibration. In the abstract setting of fibrations, a fibration with the property that each re-indexing functor f^* has a left adjoint Σ_f is called a *bifibration*, and the functors Σ_f are called *op-reindexing* functors. A bifibration that is also a full comprehension category with unit is called a *full cartesian Lawvere category* [25]. The families fibration is a full cartesian Lawvere category.

The functors Σ_f are often subject to the Beck-Chevalley condition for coproducts, which is well-known to hold for the families fibration. This condition ensures that, in certain circumstances, op-reindexing commutes with re-indexing [26]. It is used in the proof of Lemma 3.1.

At several places below we make essential use of the fact that the families fibration has very strong coproducts, i.e., that in the diagram

$$\begin{array}{ccc} \{(A,P)\} & \xrightarrow{\{\psi\}} & \{\Sigma_f(A,P)\} \\ {\scriptstyle \pi_{(A,P)}}\downarrow & & \downarrow{\scriptstyle \pi_{\Sigma_f(A,P)}} \\ A & \xrightarrow{\quad f \quad} & B \end{array} \tag{3.2}$$

where ψ is the obvious map of families of sets over f, $\{\psi\}$ is an isomorphism. This notion of very strong coproducts naturally generalises the usual notion of strong coproducts [26], and imposes a condition that is standard in models of type theory.

3.4. **Indexed Products.** For each function $f : A \to B$ and family (A, P) we can also form the family $\Pi_f(A, P) = (B, \lambda b.\ \Pi_{a \in A}.(b = fa) \to Pa)$, called the *indexed product of* (A, P) *along* f. The mapping of each family to its indexed product along f extends to a functor $\Pi_f : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_B$ which is right adjoint to the reindexing functor f^* for the families fibration. Altogether we have the following collection of relationships for each function $f : A \to B$:

$$\mathrm{Fam}(\mathrm{Set})_B \underset{\Pi_f}{\overset{\Sigma_f}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot \; \bot}}}} \mathrm{Fam}(\mathrm{Set})_A \quad \text{(with } f^* \text{ between)}$$

Like its counterpart for indexed coproducts, the Beck-Chevalley condition for indexed products is often required and indeed it holds in the families fibration. We do not make use of this condition in this paper.

3.5. **Liftings.** The relationship between inductive types and their refinements can be given in terms of liftings of functors. A *lifting* of a functor $F : \mathrm{Set} \to \mathrm{Set}$ is a functor $\hat{F} : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Fam}(\mathrm{Set})$ such that $F \circ U = U \circ \hat{F}$. A lifting is *truth-preserving* if there is a natural isomorphism $\top \circ F \cong \hat{F} \circ \top$. Truth-preserving liftings for all polynomial functors — i.e., for all functors built from identity functors, constant functors, coproducts, and products — were given by Hermida and Jacobs [24]. Truth-preserving liftings were established for arbitrary functors by Ghani *et al.* [22]. Their truth-preserving lifting $\hat{F}$ is defined on objects by

$$\begin{array}{rcl} \hat{F}(A,P) & = & (FA, \lambda x.\ \{y : F\{(A,P)\} \mid F\pi_{(A,P)}y = x\}) \\ & = & \Sigma_{F\pi_{(A,P)}} \top(F\{(A,P)\}) \end{array} \tag{3.3}$$

Reading this definition logically, we can say that $\hat{F}(A,P)$ holds for $x \in FA$ if P holds for every $a \in A$ "inside" x. Thus $\hat{F}$ is a generic definition of the *everywhere* modality, as defined for containers by Altenkirch and Morris [3]. This can be seen clearly by considering the action of the lifting in (3.3) on polynomial functors:

$$\begin{array}{lcl} \widehat{Id}(A,P) & = & (A,P) \\ \widehat{K_B}(A,P) & = & \top B = (B, \lambda x.\ 1) \\ (\widehat{F+G})(A,P) & = & \left(FA + GA,\ \lambda a.\mathsf{case}\ a\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{inl}\ x \Rightarrow \hat{F}(A,P)x \\ \mathsf{inr}\ y \Rightarrow \hat{G}(A,P)y \end{array} \right.\right) \\ (\widehat{F \times G})(A,P) & = & (FA \times GA,\ \lambda(a,b).\ \hat{F}(A,P)a \times \hat{G}(A,P)b) \end{array}$$

The identity functor on Set does not contribute any new information to proofs that a property holds for a given data element, so its lifting is the identity functor on Fam(Set). For any B, the constantly B-valued functor K_B on Set does not contribute any inductive information to proofs, so its lifting is the truth predicate $\top B$ for B. The lifting of a coproduct of functors splits into two possible cases, depending on the value being analysed. And a product of functors contributes proof information from each of its components. Lifting is defined generically in terms of the functor F, and so it is possible to compute the lifting of non-polynomial functors such as the the finite powerset functor. Ghani, Johann and Fumex [22] give further examples of the lifting $\hat{F}$ applied to non-polynomial functors.

Below, in Lemmas 3.1 and 3.2 and Sections 4, 5, 6 and 7, we will be interested in the restriction of the lifting $\hat{F}$ to fibres over particular sets A. Given an object (A,P) of $\mathrm{Fam}(\mathrm{Set})_A$, $\hat{F}(A,P)$ is an object of $\mathrm{Fam}(\mathrm{Set})_{FA}$. Therefore, if we restrict the domain of $\hat{F}$ to $\mathrm{Fam}(\mathrm{Set})_A$, we get a functor $\hat{F}_A : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_{FA}$. The subscript A on $\hat{F}_A$ indicates that we have restricted the domain to $\mathrm{Fam}(\mathrm{Set})_A$.

The final expression in (3.3) is given in terms of the constructions of Sections 3.2 and 3.3, so the definition of a lifting makes sense in any full cartesian Lawvere category.

Under certain conditions, the lifting $\hat{F}$ for any functor F is well-behaved with respect to reindexing and op-reindexing. We make this observation precise in two lemmas that will be used in our development of both our basic (Section 4) and partial refinement techniques (Section 6). To state the first, we need the notion of a pullback; this notion will also be used in Sections 5, 6, and 7 below. The *pullback* of the morphisms $f : X \to Z$ and $g : Y \to Z$

consists of an object W and two morphisms $i : W \to X$ and $j : W \to Y$ such that $g \circ j = f \circ i$. We indicate pullbacks diagrammatically by

$$\begin{array}{ccc} W & \xrightarrow{j} & Y \\ {\scriptstyle i}\downarrow & \lrcorner & \downarrow{\scriptstyle g} \\ X & \xrightarrow[f]{} & Z \end{array}$$

Moreover, for any W', $i' : W' \to X$, and $j' : W' \to Y$ such that $g \circ j' = f \circ i'$, there exists a unique morphism $u : W' \to W$ such that $i \circ u = i'$ and $j \circ u = j'$. When it exists, the pullback of f and g is unique up to (unique) isomorphism. All container functors [1], and hence all functors modelling strictly positive types, preserve pullbacks.

We can now state our lemmas.

Lemma 3.1. *For any functor $F : \mathrm{Set} \to \mathrm{Set}$ that preserves pullbacks, lifting commutes with reindexing, i.e., for all functions $f : A \to B$, there exists a vertical natural isomorphism $\hat{F}_A \circ f^* \cong (Ff)^* \circ \hat{F}_B$.*

Lemma 3.2. *For any functor $F : \mathrm{Set} \to \mathrm{Set}$, lifting commutes with op-reindexing, i.e., for all functions $f : A \to B$, there exists a vertical natural isomorphism $\hat{F}_B \circ \Sigma_f \cong \Sigma_{Ff} \circ \hat{F}_A$.*

More generally, Lemma 3.1 holds in any full cartesian Lawvere category satisfying the Beck-Chevalley condition for coproducts, whereas Lemma 3.2 holds in any full cartesian Lawvere category with very strong coproducts.

Since $\hat{F}$ is an endofunctor on $\mathrm{Fam}(\mathrm{Set})$, the category $\mathrm{Alg}_{\hat{F}}$ of $\hat{F}$-algebras exists. The families fibration $U : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Set}$ extends to a fibration $U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$, called the *algebras fibration* induced by U. Concretely, the action of U^{Alg} is the same as that of U, so that $U^{\mathsf{Alg}}(k : \hat{F}P \to P) = (Uk : FUP \to UP)$ on objects and $U^{\mathsf{Alg}}(h : (k_1 : \hat{F}P \to P) \to (k_2 : \hat{F}Q \to Q)) = Uh$ on morphisms. Moreover, writing $\top^{\mathsf{Alg}}$ and $\{-\}^{\mathsf{Alg}}$ for the truth and comprehension functors for U^{Alg}, respectively, the adjoint relationships from Diagram 3.1 all lift to give $U^{\mathsf{Alg}} \dashv \top^{\mathsf{Alg}} \dashv \{-\}^{\mathsf{Alg}}$. The two adjunctions here follow from Theorem 2.1 using the fact that $\hat{F}$ is a truth-preserving lifting. That left adjoints preserve initial objects can now be used to establish the following fundamental result, originally from Hermida and Jacobs [24], and generalised by Ghani *et al.* [22]:

Theorem 3.3. *$\top(\mu F)$ is the carrier $\mu\hat{F}$ of the initial $\hat{F}$-algebra.*

Theorem 3.3 can be generalised to any full cartesian Lawvere category. As shown by Hermida and Jacobs, and by Ghani *et al.*, it can be used to give a generic structural induction rule for any functor F having an initial algebra.

4. From Liftings to Refinements

In this section we show that the refinement of an inductive type μF by an F-algebra $\alpha : FA \to A$ — i.e., the family

$$(A, \lambda a : A.\ \{x : \mu F \mid (\!|\alpha|\!)x = a\}) \tag{4.1}$$

generalising the refinement in (1.1) — is inductively characterised as μF^α, where $F^\alpha : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_A$ is given by

$$F^\alpha(A, P) = (A, \lambda a.\ \{x : F\{(A, P)\} \mid \alpha(F\pi_{(A,P)}x) = a\}) \tag{4.2}$$

That is, $F^\alpha(A,P)$ is obtained by first building the FA-indexed type $\hat{F}(A,P)$ from Equation 3.3, and then restricting membership to those elements whose α-values are correctly computed from those of their immediate subterms. More generally, we can express F^α in terms of the constructions of Section 3 as

$$F^\alpha = \Sigma_\alpha \circ \hat{F}_A \tag{4.3}$$

Before we prove that the above construction of F^α is correct, we show that it yields the refinement of lists by the length function given in (1.1).

Example 1. The inductive type of lists of elements with type B can be specified by the functor $F_{\mathtt{List}_B}X = 1 + B \times X$. Writing Nil for the left injection and Cons for the right injection into the coproduct $F_{\mathtt{List}_B}X$, the $F_{\mathtt{List}_B}$-algebra $\mathit{lengthalg} : F_{\mathtt{List}_B}\mathbb{N} \to \mathbb{N}$ that computes the lengths of lists is

$$\begin{array}{lcl} \mathit{lengthalg}\ \mathsf{Nil} & = & 0 \\ \mathit{lengthalg}\ (\mathsf{Cons}(b,n)) & = & n+1 \end{array}$$

In the families fibration, we can calculate the refinement of $\mu F_{\mathtt{List}_B}$ by the algebra $\mathit{lengthalg}$ as follows:

$$\begin{array}{lll} & F^{\mathit{lengthalg}}_{\mathtt{List}_B}(\mathbb{N},P) & \\ = & (\mathbb{N}, \lambda n.\{x : F_{\mathtt{List}_B}\{(\mathbb{N},P)\} \mid \mathit{lengthalg}(F_{\mathtt{List}_B}\pi_{(A,P)}x) = n\}) & \\ = & (\mathbb{N}, \lambda n.\ \{x : 1 \mid \mathit{lengthalg}(\mathsf{Nil}) = n\} & \\ & \qquad + & \\ & \qquad \{x : B \times \{(\mathbb{N},P)\} \mid \mathit{lengthalg}(\mathsf{Cons}((B \times \pi_{(\mathbb{N},P)})x)) = n\}) & \end{array}$$

The first equality holds by (4.3) and the expansion of this expression in the families fibration. The second is obtained by unfolding the definition of $F_{\mathtt{List}}$ as a coproduct, which allows the refinement to be presented as a coproduct as well. In the first summand of the final expression above, $\mathit{lengthalg}(\mathsf{Nil}) = 0$, so that $\{x : 1 \mid \mathit{lengthalg}(\mathsf{Nil}) = n\}$ reduces to $\{* \mid 0 = n\}$. We can expand the product and comprehension parts of x in the second summand to see that $\{x : B \times \{(\mathbb{N},P)\} \mid \mathit{lengthalg}(\mathsf{Cons}((B \times \pi_{(\mathbb{N},P)})x)) = n\}$ reduces to $\{b : B, n_1 : \mathbb{N}, l : Pn_1 \mid \mathit{lengthalg}(\mathsf{Cons}(b,n_1)) = n\}$. Since $\mathit{lengthalg}(\mathsf{Cons}(b,n_1)) = n_1 + 1$, the whole refinement can therefore be expressed as

$$F^{\mathit{lengthalg}}_{\mathtt{List}_B}(\mathbb{N},P) = (\mathbb{N}, \lambda n.\{* \mid 0 = n\} + \{b : B, n_1 : \mathbb{N}, l : Pn_1 \mid n_1 + 1 = n\})$$

As we shall see in Theorem 4.6 below, the least fixed point $\mu F^{\mathit{lengthalg}}_{\mathtt{List}_B}$ of this functor exists and is $(\mathbb{N}, \lambda n.\ \{x : \mu F_{\mathtt{List}_B} \mid (\!|\mathit{lengthalg}|\!)x = n\})$, exactly as required. Moreover, the expression for $F^{\mathit{lengthalg}}_{\mathtt{List}_B}$ derived here is exactly the same as the definition of the functor $F_{\mathtt{Vector}_B}$ given in Section 2.2 whose least fixed point models the Agda 2 declaration of `Vector B` given in the introduction. The derivation just completed therefore justifies this definition of `Vector B`.

4.1. **Correctness of Refinement.** We now turn our attention to proving the correctness of our refinement construction from (4.2). The proof makes good use of the relationship between the category Fam(Set) and the categories $\mathrm{Fam}(\mathrm{Set})_A$ for various sets A, as well as of the lifting of this relationship to the categories $\mathrm{Alg}_{\hat{F}}$ and Alg_{F^α} of algebras.

We begin with a simple, but key, observation, namely:

Lemma 4.1. *Let (A, P) and (B, Q) be objects in* Fam(Set)*, and let $f : A \to B$ be a function. The set of morphisms h in* Fam(Set) *from (A, P) to (B, Q) such that $Uh = f$ is isomorphic to the set of morphisms in* $\mathrm{Fam}(\mathrm{Set})_A$ *from (A, P) to $f^*(B, Q)$.*

Proof. This follows directly from the definitions. On the one hand, a morphism h in Fam(Set) from (A, P) to (B, Q) such that $Uh = f$ is a pair $(f, h^\sim)$, where $h^\sim : \forall a.Pa \to Q(fa)$. On the other, the definition of the re-indexing functor f^*, i.e. $f^*(B, Q) = (A, Q \circ f)$, entails that a morphism in $\mathrm{Fam}(\mathrm{Set})_A$ from (A, P) to $f^*(B, Q)$ is a pair $(id, h^\sim)$, where $h^\sim : \forall a.Pa \to Q(fa)$. There is clearly an isomorphism between these sets of morphisms. □

To understand the relationship between the category of $\hat{F}$-algebras and the category of F^α-algebras, it is convenient to define category of $\hat{F}$-algebras that are over the F-algebra α with respect to the fibration U^{Alg} defined at the end of Section 3.5.

Definition 4.2. For each F-algebra $\alpha : FA \to A$, the category $(\mathrm{Alg}_{\hat{F}})_\alpha$ of *$\hat{F}$-algebras over α* with respect to U^{Alg} has as objects $\hat{F}$-algebras $k : \hat{F}P \to P$ such that $Uk = \alpha$, and as morphisms $\hat{F}$-algebra morphisms $f : (k_1 : \hat{F}P \to P) \to (k_2 : \hat{F}Q \to Q)$ such that $Uf = id$.

Lemma 4.3. *For each F-algebra $\alpha : FA \to A$, there is an isomorphism of categories $(\mathrm{Alg}_{\hat{F}})_\alpha \cong \mathrm{Alg}_{F^\alpha}$.*

Proof. We demonstrate only the isomorphism on objects here; the isomorphism on morphisms is similar. An object of $(\mathrm{Alg}_{\hat{F}})_\alpha$ is a pair comprising a family (A, P) and a morphism $k : \hat{F}(A, P) \to (A, P)$ in Fam(Set) such that $Uk = \alpha$. By Lemma 4.1, such morphisms k are in one-to-one correspondence with the morphisms $k' : \hat{F}(A, P) \to \alpha^*(A, P)$ in $\mathrm{Fam}(\mathrm{Set})_{FA}$. By the adjunction $\Sigma_\alpha \dashv \alpha^*$, the latter morphisms are in one-to-one correspondence with the morphisms $k'' : \Sigma_\alpha \hat{F}(A, P) \to (A, P)$ in $\mathrm{Fam}(\mathrm{Set})_A$. By the definition of F^α, these morphisms are exactly the F^α-algebras, i.e., the objects of Alg_{F^α}. □

The next lemma shows that the reindexing and op-reindexing functors for $U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$ are inherited from $U : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Set}$. We have:

Lemma 4.4. *For every F-algebra morphism $f : (\alpha : FA \to A) \to (\beta : FB \to B)$, there are functors $f^{*\mathsf{Alg}} : (\mathrm{Alg}_{\hat{F}})_\beta \to (\mathrm{Alg}_{\hat{F}})_\alpha$ and $\Sigma_f^{\mathsf{Alg}} : (\mathrm{Alg}_{\hat{F}})_\alpha \to (\mathrm{Alg}_{\hat{F}})_\beta$ such that $\Sigma_f^{\mathsf{Alg}} \dashv f^{*\mathsf{Alg}}$. Moreover, for any $\hat{F}$-algebra $k : \hat{F}(A, P) \to (A, P)$, the $\hat{F}$-algebra $\Sigma_f^{\mathsf{Alg}}(k : \hat{F}(A, P) \to (A, P))$ has carrier $\Sigma_f(A, P)$, and for any $\hat{F}$-algebra $k' : \hat{F}(B, Q) \to (B, Q)$, the $\hat{F}$-algebra $f^{*\mathsf{Alg}}(k' : \hat{F}(B, Q) \to (B, Q))$ has carrier $f^*(B, Q)$.*

Proof. By Lemma 4.3, we can treat $(\mathrm{Alg}_{\hat{F}})_\alpha$ as if it were Alg_{F^α}, and $(\mathrm{Alg}_{\hat{F}})_\beta$ as if it were Alg_{F^β}. In Section 3, we noted that for any $f : A \to B$, there are functors $f^* : \mathrm{Fam}(\mathrm{Set})_B \to \mathrm{Fam}(\mathrm{Set})_A$ and $\Sigma_f : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_B$ such that $\Sigma_f \dashv f^*$. The lemma statement is now a consequence of Theorem 2.1, provided we can establish the isomorphism $F^\beta \circ \Sigma_f \cong \Sigma_f \circ F^\alpha$. But we can verify the existence of such an isomorphism as follows:

$$
\begin{array}{lll}
 & \Sigma_f \circ F^\alpha & \\
= & \Sigma_f \circ \Sigma_\alpha \circ \hat{F}_A & \text{by the definition of } F^\alpha \\
\cong & \Sigma_\beta \circ \Sigma_{Ff} \circ \hat{F}_A & \text{since } f \text{ is an } F\text{-algebra morphism} \\
\cong & \Sigma_\beta \circ \hat{F}_B \circ \Sigma_f & \text{by Lemma 3.2} \\
= & F^\beta \circ \Sigma_f & \text{by the definition of } F^\beta
\end{array}
$$

This is exactly as required. □

We can now see that Lemma 4.1 generalises from the categories in the families fibration to those in U^{Alg}. This gives:

Lemma 4.5. *Let let $k_1 : \hat{F}(A,P) \to (A,P)$ and $k_2 : \hat{F}(B,Q) \to (B,Q)$ be objects of $(\mathrm{Alg}_{\hat{F}})_\alpha$ and $(\mathrm{Alg}_{\hat{F}})_\beta$, respectively, and let $f : (\alpha : FA \to A) \to (\beta : FB \to B)$ be an F-algebra morphism. The set of morphisms h in $\mathrm{Alg}_{\hat{F}}$ from $k_1 : \hat{F}(A,P) \to (A,P)$ to $k_2 : \hat{F}(B,Q) \to (B,Q)$ such that $U^{\mathsf{Alg}}h = f$ is isomorphic to the set of morphisms in $(\mathrm{Alg}_{\hat{F}})_\alpha$ from $k_1 : \hat{F}(A,P) \to (A,P)$ to $f^{*\mathsf{Alg}}(k_2 : \hat{F}(B,Q) \to (B,Q))$.*

Proof. The proof is tedious but not difficult. The key point entails constructing from each $\hat{F}$-algebra morphism $h : (A,P) \to (B,Q)$ such that $U^{\mathsf{Alg}}h = f$ another $\hat{F}$-algebra morphism $h'' : (A,P) \to f^*(B,Q)$ such that $U^{\mathsf{Alg}}h' = id$. This is made easier by observing that the definition of $f^{*\mathsf{Alg}} : (\mathrm{Alg}_{\hat{F}})_\beta \to (\mathrm{Alg}_{\hat{F}})_\alpha$ obtained by applying Theorem 2.1 in the proof of Lemma 4.4 is equivalent to the functor which on input $k : \hat{F}(B,Q) \to (B,Q)$ returns $\phi \circ (Ff)^* k \circ \hat{F}(f, id)$, where $\phi : (Ff)^*\beta^*(B,Q) \to \alpha^* f^*(B,Q)$ is the isomorphism derived from the fact that f is an F-algebra morphism. □

Putting this all together, we can now give our explicit characterisation of μF^α.

Theorem 4.6. *The functor F^α has an initial algebra with carrier $\Sigma_{(\!|\alpha|\!)}\top(\mu F)$, i.e., with carrier $(A, \lambda a : A.\ \{x : \mu F \mid (\!|\alpha|\!)x = a\})$.*

Proof. By Lemma 4.3, it suffices to show that the category $(\mathrm{Alg}_{\hat{F}})_\alpha$ has an initial object with carrier $\Sigma_{(\!|\alpha|\!)}\top(\mu F)$. We construct an initial object in $(\mathrm{Alg}_{\hat{F}})_\alpha$ from the initial $\hat{F}$-algebra $in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)$ from Theorem 3.3. Since U^{Alg} is a left adjoint, it preserves initial objects, so that $U^{\mathsf{Alg}}(in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F))$ is the initial F-algebra $in_F : F(\mu F) \to \mu F$. We can apply $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)}$ to the initial $\hat{F}$-algebra to get our candidate object $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)}(in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F))$. By Lemma 4.4, this candidate has carrier $\Sigma_{(\!|\alpha|\!)}\top(\mu F)$, as required.

To see that our candidate object is initial in $(\mathrm{Alg}_{\hat{F}})_\alpha$, let $k : \hat{F}(A,P) \to (A,P)$ be any object in $(\mathrm{Alg}_{\hat{F}})_\alpha$. Then

$$
\begin{aligned}
& (\mathrm{Alg}_{\hat{F}})_\alpha(\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)}(in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)), (k : \hat{F}(A,P) \to (A,P))) \\
\cong\ & (\mathrm{Alg}_{\hat{F}})_{in_F}((in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)), (\!|\alpha|\!)^{*\mathsf{Alg}}(k : \hat{F}(A,P) \to (A,P))) \\
& \qquad\qquad \text{by Lemma 4.4} \\
\cong\ & \{h : \mathrm{Alg}_{\hat{F}}((in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)), (k : \hat{F}(A,P) \to (A,P))) \mid U^{\mathsf{Alg}}h = (\!|\alpha|\!)\} \\
& \qquad\qquad \text{by Lemma 4.5}
\end{aligned}
$$

Since $in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)$ is the initial $\hat{F}$-algebra and U^{Alg} takes $(\!|k|\!)$ to $(\!|\alpha|\!)$, the final set in the above sequence has exactly one element. Thus there is exactly one morphism from $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)}(in_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F))$ to $(k : \hat{F}(A,P) \to (A,P))$ in $(\mathrm{Alg}_{\hat{F}})_\alpha$, and so our candidate object is indeed initial in $(\mathrm{Alg}_{\hat{F}})_\alpha$. □

For readers familiar with fibred category theory, we briefly sketch how our definitions and proofs may be generalised. We have been careful to state the definition of F^α in terms of the abstract structure we identified in Section 3. It can therefore be generalised to any full cartesian Lawvere category with very strong coproducts. Lemmas 4.4 and 4.5, as well as Theorem 4.6, can also be generalised. As was shown by Hermida and Jacobs [24], for any lifting $\hat{F}$, the obvious generalisation of the functor $U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$ is a fibration. The

generalisation of Lemma 4.3 is a result about the fibre categories of this fibration, and the generalisation of Lemma 4.4 shows that it is a bifibration (i.e., that the re-indexing functors have left adjoints). The generalisation of Theorem 4.6 then follows from the Proposition 9.2.2 of Jacobs' book [26], which relates initial objects in the total category of a fibration with initial objects in the fibres.

4.2. **More Example Refinements.** The following explicit formulas can be used to compute refinements for polynomial functors with respect to the families fibration:

$$\begin{array}{lcl}
Id^\alpha(A,P) & = & (A, \lambda a.\{x : \{(A,P)\} \mid \alpha(\pi_{(A,P)} x) = a\}) \\
 & = & (A, \lambda a.\{a' : A, p : Pa' \mid \alpha a' = a\}) \\
K_B^\alpha(A,P) & = & (A, \lambda a.\{x : B \mid \alpha x = a\}) \\
(G+H)^\alpha(A,P) & = & (A, \lambda a.\{x : G\ \{(A,P)\} \mid \alpha(\mathsf{inl}(G\pi_{(A,P)} x)) = a\} \\
 & & \qquad + \{x : H\ \{(A,P)\} \mid \alpha(\mathsf{inr}(H\pi_{(A,P)} x)) = a\}) \\
 & = & (A, \lambda a.\ G^{\alpha \circ \mathsf{inl}} Pa + H^{\alpha \circ \mathsf{inr}} Pa) \\
(G \times H)^\alpha(A,P) & = & (A, \lambda a.\ \{\ x_1 : G\ \{(A,P)\},\ x_2 : H\ \{(A,P)\} \mid \\
 & & \qquad \alpha(G\pi_{(A,P)} x_1, H\pi_{(A,P)} x_2) = a\})
\end{array}$$

Refinements of the identity and constant functors are as expected. Refinement splits coproducts of functors into two cases, specialising the refining algebra for each summand. It is not, however, possible to decompose the refinement of a product of functors $G \times H$ into refinements of G and H, not even by algebras other than α. This is because α may need to relate multiple elements to the overall index.

Example 2. We can refine $\mu F_{\mathtt{Tree}}$ by the $F_{\mathtt{Tree}}$-algebra *sumAlg* given by

$$\begin{array}{lcl}
\mathit{sumAlg} & : & F_{\mathtt{Tree}}\mathbb{Z} \to \mathbb{Z} \\
\mathit{sumAlg}\ (\mathsf{Leaf}\ z) & = & z \\
\mathit{sumAlg}\ (\mathsf{Node}\ (l,r)) & = & l + r
\end{array}$$

The fold of *sumAlg* sums the values stored at the leaves of a tree. It yields the refinement $\mu F^{\mathit{sumAlg}}_{\mathtt{Tree}}$ given by

$$F^{\mathit{sumAlg}}_{\mathtt{Tree}}(\mathbb{Z}, P) = (\mathbb{Z}, \lambda n.\ \{z : \mathbb{Z} \mid z = n\} + \{l, r : \mathbb{Z}, x_1 : Pl, x_2 : Pr \mid n = l + r\ \})$$

By Theorem 4.6 and the definition of $\Sigma_{(\!|\mathit{sumAlg}|\!)}$ we have that the refinement $\mu F^{\mathit{sumAlg}}_{\mathtt{Tree}}$ is $\lambda n.\{x : \mu F_{\mathtt{Tree}} \mid (\!|\mathit{sumAlg}|\!)x = n\}$. This refinement indexes the elements of $\mu F_{\mathtt{Tree}}$ by the sums of the values in their leaves. It corresponds to the Agda 2 declaration

```
data SumTree : Integer -> Set where
  SumLeaf : (z : Integer) -> SumTree z
  SumNode : (l r : Integer) -> SumTree l -> SumTree r -> SumTree (l + r)
```

Note that in the second summand of $F^{\mathit{sumAlg}}_{\mathtt{Tree}}$ we have two recursive references to P, each with a separate index, and that these indices are related to the overall index n as in the second case of *sumAlg*. However, the basic refinement process developed in this section cannot be used to require indices of subterms to be related to one another in particular ways. For instance, it cannot enforce the requirement that the two subtrees sum to the same value, or that the tree satisfy some balance property. Indeed, if such restrictions are imposed, then some elements of the underlying data type may fail to be assigned an index. We show how to treat this via partial assignment of indices in Section 6.

4.3. **Limiting cases.** The two limiting cases of refinement are deserving of attention. Refining by the initial F-algebra $in_F : F(\mu F) \to \mu F$ gives a μF-indexed type inductively characterised as the least fixed point of the functor $F^{in_F} = \Sigma_{in_F} \hat{F}$. Since in_F is an isomorphism, Σ_{in_F} is as well. Thus $F^{in_F} \cong \hat{F}$, so that $\mu F^{in_F} = \mu \hat{F} = \top(\mu F)$. Taking, for each $x : \mu F$, the canonical singleton set 1 to be $\{x\}$, we can regard each element of μF is its own index. By contrast, refinement by the final algebra $! : F1 \to 1$ gives a 1-indexed type inductively characterised by $F^!$. Since $F^! \cong F$, the inductive type $\mu F^!$ is actually μF. Since 1 is the canonical singleton set, all elements of μF have exactly the same index. Refining by the initial F-algebra thus has maximal discriminatory power, while refining by the final F-algebra has no discriminatory power whatsoever.

5. Starting with Already Indexed Types

The development in Section 4 assumes that the type being refined is the initial algebra of an endofunctor F on Set. This seems to preclude refining an inductive type that is already indexed. But since we carefully identified the abstract structure of Fam(Set) needed to construct our refinements, our results can be extended to *any* fibration having that structure. We now show that, in particular, we can refine already indexed types.

To this end, let A be a set, and suppose we want to refine an A-indexed type. As we have seen, such types may be interpreted in the category $\mathrm{Fam}(\mathrm{Set})_A$. The carrier of an F-algebra α with respect to which we want to refine an already A-indexed type will thus be an A-indexed set $B : A \to \mathrm{Set}$, and the resulting refinement will be a type of the form $\forall a.Ba \to \mathrm{Set}$, i.e., will be a family of sets that is doubly indexed by both A and B.

Just as the categories of indexed sets comprise the category Fam(Set) in Section 3, the families indexed by A-indexed sets comprise a category $\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$. (Our notation is derived from the pullback construction used to construct this category in the general setting; see below.) Objects of $\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ are pairs (B, P), where $B : A \to \mathrm{Set}$ and $P : \forall a.Ba \to \mathrm{Set}$, and morphisms are pairs $(f, f^\sim) : (B, P) \to (C, Q)$, where $f : \forall a.Ba \to Ca$ and $f^\sim : \forall a, b \in Ba.Pab \to Qa(fab)$. And just as there is a functor $U : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Set}$ defined by $U(A, P) = A$ on objects and $U(f, F^\sim) = f$ on morphisms, there is a functor $U^A : \mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Fam}(\mathrm{Set})_A$ defined by $U^A(B, P) = B$ on objects and $U^A(f, f^\sim) = f$ on morphisms. We may now recreate each of the structures we identified for the families fibration in Section 3 for the new fibration given by U^A. We have:

- *Fibres*: For each object B of $\mathrm{Fam}(\mathrm{Set})_A$, the fibre of $(\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))$ over B is the category $(\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_B$ consisting of objects of $\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ whose first component is B, and morphisms $(f, f^\sim)$, where $f = id$. By abuse of terminology, such morphisms are again said to be *vertical*.
- *Reindexing*: Given a morphism $f : B \to C$ in Fam(Set), we can define the reindexing functor $f^* : (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_C \to (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_B$ by composition, similarly to how reindexing is defined for the families fibration.
- *Truth functor*: For each set A, we can define $\top^A : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ by $\top^A(B) = (B, \lambda a\, b.\, 1)$. As in the families fibration, this mapping of objects to truth predicates extends to a functor, called the *truth functor* for U^A.
- *Comprehension functor*: For each set A, we can define $\{-\}^A : \mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Fam}(\mathrm{Set})_A$ by $\{(B, P)\}^A = \lambda a.\{(b \in Ba, p \in Pab)\}$. As in the families

fibration, this mapping of objects to their comprehensions extends to a functor, called the *comprehension functor* for U^A.

- *Indexed coproducts*: For any morphism $f : B \to C$ in $\mathrm{Fam}(\mathrm{Set})_A$, we can define $\Sigma_f : (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_B \to (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_C$ by
$$\Sigma_f(B, P) = (C,\ \lambda a\, c.\, \Sigma_{b \in Ba}.\, (c = fab) \times Pab).$$
- *Indexed products*: For any morphism $f : B \to C$ in $\mathrm{Fam}(\mathrm{Set})_A$, we can define $\Pi_f : (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_B \to (\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}))_C$ by
$$\Pi_f(B, P) = (C,\ \lambda a\, c.\Pi_{b \in Ba}.\, (c = fab) \to Pab)$$

Given these definitions, we can check by hand that they satisfy the same relationships from Section 3 that their counterparts for the families fibration do. It is therefore possible to re-state each of the definitions and results in Sections 3.5 and 4 for U^A, and, thereby, to derive refinements of already indexed inductive types. The constructions that we carry out in the families fibration in Sections 6 and 7 can similarly be carried out in U^A as well.

For readers familiar with fibred category theory, we now sketch how to generalise the above construction to construct a suitable setting for indexed refinement from any full cartesian Lawvere category with products and very strong coproducts, provided these satisfy the Beck-Chevalley condition for coproducts. For this we can use the *change-of-base* construction for generating new fibrations by pullback [26]. Indeed, if A is an object of $\mathcal{E}$, then the following pullback in Cat, the large category of categories and functors, constructs $\mathcal{E}_A \times_{\mathcal{B}} \mathcal{E}$:

$$\begin{array}{ccc} \mathcal{E}_A \times_{\mathcal{B}} \mathcal{E} & \longrightarrow & \mathcal{E} \\ {\scriptstyle U^A}\downarrow & \lrcorner & \downarrow{\scriptstyle U} \\ \mathcal{E}_A & \xrightarrow{\{-\}} & \mathcal{B} \end{array}$$

Instantiating $\mathcal{E}$ to $\mathrm{Fam}(\mathrm{Set})$ and U to the families fibration constructs $\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ as defined above, up to currying. Moreover, the following theorem shows that all the structure we require for constructing refinements is preserved by the change-of-base construction, and thus ensures that the change-of-base construction can be iterated as often as desired.

Theorem 5.1. *If U is a full cartesian Lawvere category with products and with very strong coproducts satisfying the Beck-Chevalley condition for coproducts, then so is U^A.*

Proof. (Sketch) First, U^A is well-known to be a fibration by its definition via the change-of-base construction [26]. The truth functor for U^A is defined for objects P in $\mathcal{E}_A$ by $\top^A P = (P, \top\{P\})$, and the comprehension functor for U^A is defined by $\{(P, Y)\}^A = \Sigma_{\pi_P} Y$, where $P \in \mathcal{E}_A$ and $Y \in \mathcal{E}_{\{P\}}$. Coproducts are defined directly using the coproducts of U. □

Example 3. To demonstrate the refinement of an inductive type which is already indexed we consider a small expression language of well-typed terms. Let $\mathcal{T} = \{\mathsf{int}, \mathsf{bool}\}$ be the set of possible base types. The language is $\mu F_{\mathtt{wtexp}}$ for the functor $F_{\mathtt{wtexp}} : \mathrm{Fam}(\mathrm{Set})_{\mathcal{T}} \to \mathrm{Fam}(\mathrm{Set})_{\mathcal{T}}$ given by

$$\begin{aligned} F_{\mathtt{wtexp}}(\mathcal{T}, P) \quad = \quad (\mathcal{T}, \lambda t : \mathcal{T}.\ & \{z : \mathbb{Z} \mid t = \mathsf{int}\} \\ & + \{b : \mathbb{B} \mid t = \mathsf{bool}\} \\ & + \{x_1 : Pt,\ x_2 : Pt \mid t = \mathsf{int}\} \\ & + \{x_1 : P\mathsf{bool},\ x_2 : Pt,\ x_3 : Pt\}) \end{aligned}$$

This specification of an inductive type corresponds to the following Agda 2 declaration, where we write `Ty` for the Agda 2 equivalent of the set $\mathcal{T}$:

```
data WTExp : Ty -> Set where
  intConst  : Integer -> WTExp Int
  boolConst : Boolean -> WTExp Bool
  add       : WTExp Int -> WTExp Int -> WTExp Int
  if        : (t : Ty) -> WTExp Bool -> WTExp t -> WTExp t -> WTExp t
```

The type `WTExp` cannot be constructed by the process of refinement presented in Section 4. Indeed, the indices of subexpressions, and not just the overall indexes, are constrained in the types of the `add` and `if` constructors. This accords with the discussion at the end of Section 4.2. Fortunately we can, and will, show in Section 6 how to extend the notion of refinement to the situation where not every element of a data type can be assigned an index.

Meanwhile, in light of Theorem 5.1, we can refine the already indexed type $\mu F_{\texttt{wtexp}}$. For any t, write $\mathsf{IntConst}$, $\mathsf{BoolConst}$, Add, and If for the injections into $(snd\,(F_{\texttt{wtexp}}(\mathcal{T}, P)))\,t$. Let $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ denote the set of booleans, and assume there exists a $\mathcal{T}$-indexed family T such that $T\ \mathsf{int} = \mathbb{Z}$ and $T\ \mathsf{bool} = \mathbb{B}$. Then T gives a semantic interpretation of the types from $\mathcal{T}$ that can be used to define an $F_{\texttt{wtexp}}$-algebra *evalAlg* whose fold specifies a "tagless" interpreter. We have:

$$
\begin{array}{lcl}
\mathit{evalAlg} & : & F_{\texttt{wtexp}}(\mathcal{T}, T) \to (\mathcal{T}, T) \\
\mathit{evalAlg} & = & (\mathit{id}, \lambda x : \mathcal{T}.\ \lambda t : \mathit{snd}\,(F_{\texttt{wtexp}}(\mathcal{T}, T))\,x.\ \mathsf{case}\ t\ \mathsf{of} \\
& & \qquad \begin{array}{lcl}
\mathsf{IntConst}\ z & \Rightarrow & z \\
\mathsf{BoolConst}\ b & \Rightarrow & b \\
\mathsf{Add}\ (z_1, z_2) & \Rightarrow & z_1 + z_2 \\
\mathsf{If}\ (b, x_1, x_2) & \Rightarrow & \mathsf{if}\ b\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)
\end{array}
\end{array}
$$

The function $(\!|\mathit{evalAlg}|\!) : \forall t.\ \mu F_{\texttt{wtexp}} t \to T t$ does indeed give a semantics to each well-typed expression. Refining $\mu F_{\texttt{wtexp}}$ by *evalAlg* yields an object `WTExpSem` of $\mathrm{Fam}(\mathrm{Set})_{\mathcal{T}} \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ over $(\mathcal{T}, T)$, i.e, an object of $\mathrm{Fam}(\mathrm{Set})$ indexed by $\{(\mathcal{T}, T)\}$. This $\{(\mathcal{T}, T)\}$-indexed data type associates to every well-typed expression that expression's semantics. As an Agda 2 declaration, it can be expressed as follows, after applying a few type isomorphisms to make the declaration more idiomatic:

```
data WTExpSem : (t : Ty) -> T t -> Set where
  intConst  : (z : Integer) ->       WTExpSem Int z
  boolConst : (b : Boolean) ->       WTExpSem Bool b
  add       : (z1 z2 : Integer) ->
              WTExpSem Int z1 ->
              WTExpSem Int z2 ->     WTExpSem Int (z1 + z2)
  if        : (b : Boolean) ->
              (t : Ty) ->
              (x1 x2 : T t) ->
              WTExpSem Bool b ->
              WTExpSem t x1 ->
              WTExpSem t x2 ->       WTExpSem t (if b then x1 else x2)
```

Here, we have assumed a standard `if_then_else` notation for eliminating booleans.

6. Partial Refinement

In Sections 4 and 5 we assumed that every element of an inductive type can be assigned an index. Every list has a length, every tree has a number of leaves, every well-typed expression has a semantic meaning, and so on. But how can an inductive type be refined if only *some* data have values by which we want to index? For example, how can the inductive type of well-typed expressions of Example 3 be obtained by refining a data type of untyped expressions by an algebra for type assignment? And how can the inductive type of red-black trees be obtained by refining a data type of coloured trees by an algebra enforcing the well-colouring properties? As these questions suggest, the problem of refining subsets of inductive types is a common and naturally occurring one. Our partial refinement technique, which we now describe, can solve this problem.

6.1. **Partial Algebras.** To generalise our theory to partial refinements we move from algebras to partial algebras. If F is a functor, then a *partial F-algebra* is a pair $(A, \alpha : FA \to (1 + A))$ comprising a carrier A and a structure map $\alpha : FA \to (1 + A)$. We write $\mathsf{ok} : A \to 1 + A$ and $\mathsf{fail} : 1 \to 1 + A$ for the injections into $1 + A$, and often refer to a partial algebra solely by its structure map. The functor $MA = 1 + A$ is (the functor part of) the *error monad*.

Example 4. The inductive type of expressions is $\mu F_{\mathtt{exp}}$ for the functor $F_{\mathtt{exp}}X = \mathbb{Z} + \mathbb{B} + (X \times X) + (X \times X \times X)$. Letting $\mathcal{T} = \{\mathsf{int}, \mathsf{bool}\}$ as in Example 3, and using the obvious convention for naming the injections into $F_{\mathtt{exp}}X$, types can be inferred for expressions using the following partial $F_{\mathtt{exp}}$-algebra:

$$
\begin{array}{lcl}
\mathit{tyInfer} & : & F_{\mathtt{exp}}\mathcal{T} \to 1 + \mathcal{T} \\
\mathit{tyInfer}\ (\mathsf{IntConst}\ z) & = & \mathsf{ok}\ \mathsf{int} \\
\mathit{tyInfer}\ (\mathsf{BoolConst}\ b) & = & \mathsf{ok}\ \mathsf{bool} \\
\mathit{tyInfer}\ (\mathsf{Add}\ (t_1, t_2)) & = & \begin{cases} \mathsf{ok}\ \mathsf{int} & \text{if } t_1 = \mathsf{int} \text{ and } t_2 = \mathsf{int} \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\mathit{tyInfer}\ (\mathsf{If}\ (t_1, t_2, t_3)) & = & \begin{cases} \mathsf{ok}\ t_2 & \text{if } t_1 = \mathsf{bool} \text{ and } t_2 = t_3 \\ \mathsf{fail} & \text{otherwise} \end{cases}
\end{array}
$$

Example 5. Let $\mathbb{C} = \{\mathsf{R}, \mathsf{B}\}$ be a set of colours. The inductive type of coloured trees is $\mu F_{\mathtt{ctree}}$ for the functor $F_{\mathtt{ctree}}X = 1 + \mathbb{C} \times X \times X$. We write Leaf and Br for injections into $F_{\mathtt{ctree}}X$. Red-black trees [13] are coloured trees satisfying the following constraints:

(1) Every leaf is black;
(2) Both children of a red node are black;
(3) For every node, all paths to leaves contain the same number of black nodes.

We can check whether or not a coloured tree is a red-black tree using the following partial $F_{\mathtt{ctree}}$-algebra. Its carrier $\mathbb{C} \times \mathbb{N}$ records the colour of the root in the first component and the number of black nodes to any leaf, assuming this number is the same for every leaf, in the second. We have:

$$
\begin{array}{lcl}
\mathit{checkRB} & : & F_{\mathtt{ctree}}(\mathbb{C} \times \mathbb{N}) \to 1 + (\mathbb{C} \times \mathbb{N}) \\
\mathit{checkRB}\ \mathsf{Leaf} & = & \mathsf{ok}\ (\mathsf{B}, 1) \\
\mathit{checkRB}\ (\mathsf{Br}\ (\mathsf{R}, (s_1, n_1), (s_2, n_2))) & = & \begin{cases} \mathsf{ok}\ (\mathsf{R}, n_1) & \text{if } s_1 = s_2 = \mathsf{B} \text{ and } n_1 = n_2 \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\mathit{checkRB}\ (\mathsf{Br}\ (\mathsf{B}, (s_1, n_1), (s_2, n_2))) & = & \begin{cases} \mathsf{ok}\ (\mathsf{B}, n_1 + 1) & \text{if } n_1 = n_2 \\ \mathsf{fail} & \text{otherwise} \end{cases}
\end{array}
$$

6.2. **Using a Partial Algebra to Select Elements.** We now show how, given a partial algebra, we can use it to select some of the elements of an underlying type and assign them indices. The key to doing this is to turn every partial F-algebra into a (total) F-algebra. Let $\lambda : F \circ M \to M \circ F$ be any distributive law for the error monad M over the functor F. Then λ respects the unit and multiplication of M (see [6] for details). Every partial F-algebra $\kappa : FA \to (1 + A)$ generates an F-algebra $\overline{\kappa} : F(1 + A) \to (1 + A)$ defined by $\overline{\kappa} = [\mathsf{fail}, \kappa] \circ \lambda_A$, where $[\mathsf{fail}, \kappa]$ is the cotuple of the functions fail and κ.

We can use $\overline{\kappa}$ to construct the following global characterisation of the indexed type for which we seek an inductive characterisation:

$$(A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!)x = \mathsf{ok}\ a\})$$

As in (1.1), we can consider this characterisation a specification; it is similar to the specification in Section 4, except that the index generated by the algebra $\overline{\kappa}$ is required to return $\mathsf{ok}\, a$ for some $a \in A$. We can rewrite this specification as follows, using the categorical constructions from Section 3 and Theorem 4.6:

$$(A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!)x = \mathsf{ok}\, a\}) = \mathsf{ok}^* \circ \Sigma_{(\!|\overline{\kappa}|\!)} \top(\mu F) = \mathsf{ok}^* \mu F^{\overline{\kappa}} \tag{6.1}$$

Rewriting the specification in this way links partial refinements with the indexed inductive type generated by the refinement process given in Section 4.

6.3. **Construction and Correctness of Partial Refinement.** Refining μF by the F-algebra $\overline{\kappa}$ using the techniques of Section 4 would result in an inductive type indexed by $1 + A$. But our motivating examples suggest that what we actually want is an A-indexed type that inductively describes only those terms having values of the form $\mathsf{ok}\ a$ for some $a \in A$. Partial refinement constructs, from a functor F with initial algebra $in_F : F(\mu F) \to \mu F$, and a partial F-algebra $\kappa : FA \to 1 + A$, a functor $F^{?\kappa}$ such that $\mu F^{?\kappa} \cong (A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!)x = \mathsf{ok}\ a\}) = \mathsf{ok}^* \mu F^{\overline{\kappa}}$. To this end, we define

$$F^{?\kappa} = \mathsf{ok}^* \circ \Sigma_\kappa \circ \hat{F}_A \tag{6.2}$$

We note that, in the special case of the families fibration, this definition specialises to $F^{?\kappa} = (A, \lambda a.\{x : F\{(A, P)\} \mid \kappa(F\pi_{(A,P)}x) = \mathsf{ok}\ a\})$. Now, since left adjoints preserve initial objects, we can prove $\mu F^{?\kappa} \cong \mathsf{ok}^* \mu F^{\overline{\kappa}}$ by lifting the adjunction on the left below (cf. Section 3.4) to an adjunction between $\mathrm{Alg}_{F^{?\kappa}}$ and $\mathrm{Alg}_{F^{\overline{\kappa}}}$ via Theorem 2.1:

$$\mathrm{Fam(Set)}_A \underset{\Pi_{\mathsf{ok}}}{\overset{\mathsf{ok}^*}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathrm{Fam(Set)}_{1+A} \qquad \Rightarrow \qquad \mathrm{Alg}_{F^{?\kappa}} \underset{\longrightarrow}{\overset{\longleftarrow}{\bot}} \mathrm{Alg}_{F^{\overline{\kappa}}}$$

To satisfy the precondition of Theorem 2.1, we must prove that $F^{?\kappa} \circ \mathsf{ok}^* \cong \mathsf{ok}^* \circ F^{\overline{\kappa}}$. To show this, we reason as follows:

$$\begin{array}{lll} & \mathsf{ok}^* \circ F^{\overline{\kappa}} & \\ = & \mathsf{ok}^* \circ \Sigma_{\overline{\kappa}} \circ \hat{F}_A & \text{by definition of } F^{\overline{\kappa}} \\ \cong & \mathsf{ok}^* \circ \Sigma_\kappa \circ (F\,\mathsf{ok})^* \circ \hat{F}_A & \text{by Lemma 6.1 below} \\ \cong & \mathsf{ok}^* \circ \Sigma_\kappa \circ \hat{F}_{1+A} \circ \mathsf{ok}^* & \text{by Lemma 3.1} \\ = & F^{?\kappa} \circ \mathsf{ok}^* & \text{by definition of } F^{?\kappa} \end{array}$$

In these steps we have made use of two auxiliary results, relying on two assumptions. First, in order to apply Lemma 3.1, we have assumed that F preserves pullbacks. Secondly, we have made use of the vertical natural isomorphism $\mathsf{ok}^* \circ \Sigma_{\overline{\kappa}} \cong \mathsf{ok}^* \circ \Sigma_\kappa \circ (F\mathsf{ok})^*$. We may

deduce the existence of the latter if we assume that the following property, which we call *non-introduction of failure*, is satisfied by the distributive law λ for the error monad M over F: for all $x : F(1 + A)$ and $y : FA$, $\lambda_A\, x = \mathsf{ok}\ y$ if and only if $x = F\,\mathsf{ok}\, y$. This property strengthens the usual unit axiom for distributive laws in which the implication holds only from right to left, and ensures that if applying λ does not result in failure, then no failures were present in the data to which λ was applied. Every container functor has a canonical distributive law for M satisfying the non-introduction of failure property.

Lemma 6.1. *If the distributive law λ satisfies non-introduction of failure, then $\mathsf{ok}^* \circ \Sigma_{\overline{\kappa}} \cong \mathsf{ok}^* \circ \Sigma_\kappa \circ (F\mathsf{ok})^*$.*

Proof. Given $(F(1 + A), P : F(1 + A) \to \mathrm{Set})$, we have

$$\begin{aligned} & (\mathsf{ok}^* \circ \Sigma_{\overline{\kappa}})(F(1 + A), P) \\ = \ & (A, \lambda a : A.\ \{(x_1 : F(1 + A), x_2 : Px_1) \mid [\mathsf{fail}, \kappa](\lambda_A x_1) = \mathsf{ok}\ a\}) \\ \cong \ & (A, \lambda a : A.\ \{x_1 : FA, x_2 : P(F\,\mathsf{ok}\, x_1) \mid \kappa x_1 = \mathsf{ok}\ a\}) \\ \cong \ & (A, \mathsf{ok}^* \circ \Sigma_\kappa \circ (F\,\mathsf{ok})^*(F(1 + A), P)) \end{aligned}$$

Here, we have instantiated the definitions in terms of the constructions from Section 3 for the families fibration. □

Putting everything together, we have shown the correctness of partial refinement:

Theorem 6.2. *If λ is a distributive law for the error monad M over F with the non-introduction of failure property, and if F preserves pullbacks, then $F^{?\kappa}$ has an initial algebra whose carrier is given by any, and hence all, of the expressions in (6.1).*

In fact, Lemma 6.1, and hence Theorem 6.2, holds in the more general setting of a full cartesian Lawvere category with products and very strong coproducts that satisfy the Beck-Chevalley condition for coproducts, provided that the base category satisfies extensivity [10]. In the general setting, the non-introduction of failure property can be formulated as requiring that the following square (which is the unit axiom for the distributive law λ) is a pullback:

$$\begin{array}{ccc} FA & \xrightarrow{F\mathsf{ok}} & F(1 + A) \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle \lambda_A} \\ FA & \xrightarrow{\mathsf{ok}} & 1 + FA \end{array}$$

Moreover, Theorem 5.1 extends to show that extensivity is also preserved by change-of-base provided all of the the fibres of the given full cartesian Lawvere category satisfy extensivity. This ensures that the process of partial refinement can be iterated as often as desired.

7. Refinement by Zygomorphisms and Small Indexed Induction-Recursion

The refinement process of Section 4 allows us to refine an inductive data type by any function definable as a fold. Despite this generality, the restriction to functions defined by folds can be a burden. Consider, for example, the following structurally recursive function on natural numbers that computes factorials:

```
factorial : Nat -> Nat
factorial zero     = succ zero
factorial (succ n) = succ n * factorial n
```

This `factorial` function is not immediately expressible as a fold of an algebra on the natural numbers; indeed, the right-hand side of the second clause uses both the result of a recursive call and the current argument, but a fold cannot use the current argument in computing its result. The style of definition exemplified by `factorial` is known as a *paramorphism* [33]. As we recall in Section 7.1 below, such definitions can be reduced to folds. However, reducing `factorial` to a fold and then refining as in Section 4 yields a (`Nat` × `Nat`)-indexed type, i.e., a doubly indexed type that reveals the auxiliary data used to define `factorial` as a fold. But rather than (`Nat` × `Nat`)-indexed type, what we actually want is an inductive characterisation of the following `Nat`-indexed type:

$$\texttt{FactorialNat n} \cong \{\texttt{x} : \texttt{Nat} \mid \texttt{factorial x} = \texttt{n}\} \tag{7.1}$$

If we try to implement `FactorialNat` inductively in Agda 2, then we get stuck at the point marked by `???` below:

```
data FactorialNat : Nat -> Set where
  fnzero : FactorialNat (succ zero)
  fnsucc : {n : Nat} ->
           (x : FactorialNat n) ->
           FactorialNat (succ ??? * n)
```

We'd like to put `x` in place of `???`, but there is a problem. Indeed, if `x : FactorialNat n`, then in (7.1) we know that `x : Nat`, so we can use the assertion `factorial x = n`. But in the above Agda 2 code we cannot conclude that if `x : FactorialNat n`, then `x : Nat`, and so we cannot use the fact that `factorial x = n`. What is required is a function `forget` of type `n : Nat -> FactorialNat n -> Nat` that converts an element of `FactorialNat n` into its underlying natural number. Unfortunately, we cannot first define the data type `FactorialNat` and then define the function `forget` thereafter. Instead, as becomes evident upon replacing `???` by `forget x` in the definition of `FactorialNat`, we must define both simultaneously.

Fortunately, this can be done using the principle of definition by *indexed induction-recursion* (IIR) due to Dybjer and Setzer [19, 20]. Agda 2 supports indexed induction recursion, and so `FactorialNat` and `forget` can be defined (simultaneously) as follows:

```
mutual
  data FactorialNat : Nat -> Set where
    fnzero : FactorialNat (succ zero)
    fnsucc : {n : Nat} ->
             (x : FactorialNat n) ->
             FactorialNat (succ (forget x) * n)

  forget : {n : Nat} -> FactorialNat n -> Nat
  forget fnzero     = zero
  forget (fnsucc x) = succ (forget x)
```

As we have already noted, it is possible to make sense of functions such as `factorial` in terms of initial F-algebras by using the existing notion of a *paramorphism* and its generalisation, a *zygomorphism*, but this gives incorrectly indexed types. Instead, making use of a presentation of inductive-recursive definitions as initial algebras (Section 7.2), we show in Section 7.3 that the definition of `FactorialNat` can be generalised to an inductive-recursive type satisfying the analogue of (7.1) for all zygomorphisms (rather than just `factorial`) and all initial algebras of functors (rather than just `Nat`).

7.1. **Zygomorphisms and Paramorphisms.** *Zygomorphisms* were introduced by Malcolm [31], and have as a special case the concept of a *paramorphism* [33]. Given a morphism $\gamma : F(D \times A) \to A$ and an F-algebra $\delta : FD \to D$ we define the F-algebra $\overline{\gamma, \delta} : F(D \times A) \to D \times A$ by $\langle \delta \circ F\pi_1, \gamma \rangle$. The zygomorphism h associated with $\gamma, \overline{\delta}$ is defined to be $\pi_2 \circ (\!|\overline{\gamma, \delta}|\!) : \mu F \to A$. It is the unique morphism satisfying the equation $h \circ in_F = \gamma \circ F\langle (\!|\delta|\!), h \rangle$. Paramorphisms are a special case of zygomorphisms for which δ is the initial F-algebra $in_F : F(\mu F) \to \mu F$.

The `factorial` function above can be represented as a paramorphism (and hence as a zygomorphism). Recalling that the carrier of the initial algebra for the functor $F_{\mathtt{Nat}} X = 1 + X$ is $\mathbb{N}$, we can define

$$\begin{array}{lcl} \mathit{fact} & : & F_{\mathtt{Nat}}(\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \\ \mathit{fact}\ \mathsf{zero} & = & 1 \\ \mathit{fact}\ (\mathsf{succ}\ (n, x)) & = & (n + 1) * x \end{array} \tag{7.2}$$

Here, we have used zero and succ as suggestive names for the two injections into $1 + X$. Taking γ to be *fact*, the induced paramorphism from $\mathbb{N}$ to $\mathbb{N}$ is exactly the factorial function.

7.2. **Initial Algebra Semantics of Indexed Small Induction-Recursion.** Indexed induction-recursion allows us to define a family of types $X : A \to \mathrm{Set}$ simultaneously with a recursive function $f : \forall a.\ Xa \to Da$, for some A-indexed collection of potentially large types Da. We are interested in the case when D does not depend on A, so that Da is D, and D is small, i.e., D is a set. In this situation, the semantics of IIR definitions can be given as initial algebras of functors over slice categories. We recall the definition of slice categories on Set. Given a set D, the *slice category* Set/D on Set has as objects pairs $(Z : \mathrm{Set}, f : Z \to D)$. A morphism from (Z, f) to (Y, g) in Set/D is a function from $h : Z \to Y$ such that $f = g \circ h$. We write f for (Z, f) when Z can be inferred from context.

Noting that $\forall a.Xa \to D$ is isomorphic to $(\Sigma a.Xa) \to D$ and that $\Sigma a.Xa = \{(A, X)\}$, this leads us to consider the category $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ each of whose objects is an A-indexed set X together with a function from $\{(A, X)\}$ to D. A morphism in this category from (X, f) to (X', g) is a function $\phi : \forall a.Xa \to X'a$ such that $\forall a : A.\, p : Xa.\, f(a, p) = g(a, \phi a p)$. In fact, this category is the following pullback:

$$\begin{array}{ccc} \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D & \longrightarrow & \mathrm{Set}/D \\ \downarrow & & \downarrow{\scriptstyle \pi_1} \\ \mathrm{Set}^A & \xrightarrow{\{-\}} & \mathrm{Set} \end{array}$$

The pair (`FactorialNat`, `forget`) can be interpreted as the carrier of the initial algebra of the following functor on $\mathrm{Set}^{\mathbb{N}} \times_{\mathrm{Set}} \mathrm{Set}/\mathbb{N}$:

$$\begin{array}{l} F_{\mathtt{FactorialNat}}(X : \mathrm{Set}^{\mathbb{N}}, f : \{(\mathbb{N}, X)\} \to \mathbb{N}) = \\ \quad (\lambda n.\ \{* \mid n = 1\} + \{(n_1 : \mathbb{N}, x : Xn_1) \mid n = (n_1 + 1) * f(n_1, x)\}, \\ \quad\ \lambda(n, x).\ \mathsf{case}\ x\ \mathsf{of} \\ \qquad\qquad \mathsf{inl}\ * \Rightarrow 0 \\ \qquad\qquad \mathsf{inr}\ (n_1, x) \Rightarrow f(n_1, x) + 1) \end{array} \tag{7.3}$$

The first component of $F_{\mathtt{FactorialNat}}(X, f)$ defines the constructors of `FactorialNat` in a manner similar to that described in Section 2.2. Note that this first component depends on

both X and f, which is characteristic of inductive-recursive, as well as of indexed inductive-recursive, definitions. The second component of $F_{\texttt{FactorialNat}}(X, f)$ extends the function f to the new cases given in the first component of $F_{\texttt{FactorialNat}}(X, f)$.

To develop refinement by zygomorphisms, we use a similar methodology to that in Section 6. We first use the refinement process of Section 4 to generate a functor on $\mathrm{Set}^{D\times A}$ which has an initial algebra, and then apply Theorem 2.1 with the adjoint equivalence in the next theorem to produce the initial algebra for the functor on $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ that we define in (7.6) below.

Theorem 7.1. *There is an adjoint equivalence* $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D \simeq \mathrm{Set}^{D\times A}$ *which is witnessed by the following pair of functors:*

$$\begin{array}{lcl} \Psi & : & \mathrm{Set}^{D\times A} \to \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D \\ \Psi(X) & = & (\,\lambda a.\,\{(d,x) \mid d : D, x : X(d,a)\},\ \lambda(a,(d,x)).d) \end{array}$$

$$\begin{array}{lcl} \Phi & : & \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D \to \mathrm{Set}^{D\times A} \\ \Phi(X,f) & = & \lambda(d,a).\,\{x : Xa \mid f(a,x) = d\} \end{array}$$

Proof. This is a simple consequence of the fact that, for any set X, $\mathrm{Set}^X \simeq \mathrm{Set}/X$. □

In light of the equivalence demonstrated in Theorem 7.1, we could use $\mathrm{Set}^{D\times A}$, rather than $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$, as the appropriate category for refinement by zygomorphisms. Our reasons for choosing the latter are twofold. First, as we noted in the introduction to this section, we want an A-indexed type rather than a $(D \times A)$-indexed type. Secondly, we want to define a function from that A-indexed type into D itself, rather than into a D-indexed type.

7.3. **Refinement by Zygomorphisms.** We now show how to refine an inductive type by a zygomorphism to obtain an indexed inductive-recursive definition. Generalising the example of `FactorialNat` above, we want to construct from an F-algebra $\delta : FD \to D$ and a morphism $\gamma : F(D \times A) \to A$ an inductive-recursive characterisation of the following A-indexed set and accompanying D-valued function:

$$(\lambda a.\,\{(d : D, x : \mu F) \mid (\!|\overline{\gamma,\delta}|\!)x = (d,a)\},\ \lambda(a,(d,x)).\ d) : \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D \tag{7.4}$$

Note that although the fold $(\!|\overline{\gamma,\delta}|\!)$ applied to x produces a pair (d,a), the first component of the pair in (7.4) is an A-indexed set, rather than an $(A \times D)$-indexed set. We can now see that the object of $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ in (7.4) is isomorphic to

$$(\lambda a.\,\{x : \mu F \mid \pi_2((\!|\overline{\gamma,\delta}|\!)x) = a\}, \lambda(a,x).(\!|\delta|\!)x) \tag{7.5}$$

The first component of (7.5), and hence the first component of (7.4), is the refinement of μF by the zygomorphism $\pi_2 \circ (\!|\overline{\gamma,\delta}|\!)$, and is thus is the A-indexed set we want to characterise inductively. To do this, we characterise (7.4) inductively. More specifically, we prove in Theorem 7.2 below that the least fixed point of the following functor on $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ gives an inductive-recursive characterisation of (7.4):

$$F^{\gamma,\delta}(X,f) = \begin{array}{l} (\lambda a.\ \{x : F\{(D \times A, \Phi(X,f))\} \mid \gamma(F\pi_{(D\times A,\Phi(X,f))}x) = a\}, \\ \quad \lambda(a,x).\ \delta(F\pi_1(F\pi_{(D\times A,\Phi(X,f))}x))) \end{array} \tag{7.6}$$

This definition makes use of the functor $\Phi : \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}_{/D} \to \mathrm{Set}^{A\times D}$ defined in Theorem 7.1. The first component of $F^{\gamma,\delta}(X,f)$ uses Φ to bundle up X and f into a $(D \times A)$-indexed

set, and then applies $\Sigma_\gamma \circ \hat{F}$ as in the basic refinement construction in Section 4. The second component of $F^{\gamma,\delta}(X, f)$ extracts the underlying FD component of x and then applies δ.

Example 6. We instantiate the characterisation of $F^{\gamma,\delta}$ in (7.6) for the `factorial` function from the introduction to this section. That is, we consider the functor $F_{\mathtt{Nat}}X = 1 + X$, the F-algebra $in_{F_{\mathtt{Nat}}} : F_{\mathtt{Nat}}\mathbb{N} \to \mathbb{N}$, and the morphism $fact : F_{\mathtt{Nat}}(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ defined in (7.2). Instantiating (7.6) gives

$$\begin{array}{ll} & F_{\mathtt{Nat}}^{fact,in_{F_{\mathtt{Nat}}}}(X, f) \\ = & (\lambda n.\{x : F_{\mathtt{Nat}}\{(D \times A, \Phi(X, f)\}) \mid fact(F_{\mathtt{Nat}}\pi_{(D\times A,\Phi(X,f))}x) = n\}, \\ & \quad \lambda(n, x).in_{F_{\mathtt{Nat}}}(F_{\mathtt{Nat}}\pi_1(F_{\mathtt{Nat}}\pi_{(D\times A,\Phi(X,f))}x))) \\ = & (\lambda n.\{x : 1 + \{(D \times A, \Phi(X, f))\} \mid fact((1 + \pi_{(D\times A,\Phi(X,f))})x) = n\}, \\ & \quad \lambda(n, x).in_{F_{\mathtt{Nat}}}((1 + \pi_1)((1 + \pi_{(D\times A,\Phi(X,f))})x))) \end{array}$$

We can rewrite the first component of $F_{\mathtt{Nat}}^{fact,in_{F_{\mathtt{Nat}}}}(X, f)$ to the following $\mathbb{N}$-indexed set depending on X and f:

$$\lambda n.\{* \mid fact(\mathsf{zero}) = n\} + \{(d, n_1), x : Xn_1 \mid f(n_1, x) = d, fact(\mathsf{succ}(d, n_1)) = n\}.$$

The d component in the second summand above is constrained to be $f(n_1, x)$, so we can first remove all references to d and then rewrite according to the definition of $fact$ to obtain

$$\lambda n.\{* \mid 1 = n\} + \{n_1, x : Xn_1 \mid (f(n_1, x) + 1) * n_1 = n\}$$

Using this rewriting of the first component of the instantiation, we can rewrite the second component of $F_{\mathtt{Nat}}^{fact,in_{F_{\mathtt{Nat}}}}(X, f)$ to use pattern matching and normal arithmetic notation to get

$$\lambda(n, x).\,\mathsf{case}\ x\ \mathsf{of} \begin{cases} \mathsf{zero} & \Rightarrow \quad 0 \\ \mathsf{succ}(n_1, x) & \Rightarrow \quad f(n_1, x) + 1 \end{cases}$$

We have thus derived the definition of $F_{\mathtt{FactorialNat}}$ from (7.3) solely by way of a mechanical process, using the components of the paramorphism that computes factorials. Moreover, by Theorem 7.2 below, we know that this functor has an initial algebra, and that this initial algebra represents the refinement of the natural numbers by the zygomorphism defining the function `factorial`.

As described above, the correctness of refinement by a zygomorphism is a consequence of Theorem 2.1 and the adjoint equivalence from Theorem 7.1. Indeed, we have:

Theorem 7.2. *The functor $F^{\gamma,\delta} : \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D \to \mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ defined in (7.6) has an initial algebra whose carrier is given in (7.4).*

Proof. Observe that the object of $\mathrm{Set}^A \times_{\mathrm{Set}} \mathrm{Set}/D$ in (7.4) is isomorphic to the result of applying the functor Ψ defined in Theorem 7.1 to the result of refining μF by the algebra $(\overline{\gamma, \delta}) : F(D \times A) \to D \times A$. Indeed,

$$\begin{array}{ll} & \Psi(\mu F^{\overline{\gamma,\delta}}) \\ \cong & \Psi(\lambda(d, a).\,\{x : \mu F \mid (\!|\overline{\gamma, \delta}|\!)x = (d, a)\}) \\ = & (\lambda a.\,\{(d : D, x : \mu F) \mid (\!|\overline{\gamma, \delta}|\!)x = (d, a)\},\ \lambda(a, (d, x)).d) \end{array}$$

The isomorphism in the first step above is by the refinement process from Section 4, and the equality in the second is by definition of Ψ. Now, to apply Theorem 2.1 we must show

that $F^{\gamma,\delta} \circ \Psi \cong \Psi \circ F^{\overline{\gamma,\delta}}$. So suppose X is in $\mathrm{Set}^{D\times A}$. Then

$$\begin{array}{ll} & F^{\gamma,\delta}(\Psi(X)) \\ = & (\lambda a.\, \{x : F\{(D \times A, \Phi(\Psi(X)))\} \mid \gamma(F\pi x) = a\},\, \lambda(a,x).\, \delta(F\pi_1(F\pi x))\}) \\ \cong & (\lambda a.\, \{x : F\{(D \times A, X)\} \mid \gamma(F\pi x) = a\},\, \lambda(a,x).\, \delta(F\pi_1(F\pi x))\}) \end{array}$$

Here, we have used the fact that the functors Φ and Ψ form an adjoint equivalence by Theorem 7.1. On the other hand,

$$\begin{array}{ll} & \Psi(F^{\overline{\gamma,\delta}}X) \\ = & \Psi(\lambda(d,a).\, \{x : F\{(D \times A, X)\} \mid \overline{(\gamma,\delta)}(F\pi x) = (d,a)\}) \\ = & \Psi(\lambda(d,a).\, \{x : F\{(D \times A, X)\} \mid \gamma(F\pi x) = a,\, \delta(F\pi_1(F\pi x)) = d\}) \\ \cong & (\lambda a.\, \{(d : D, x : F\{(D \times A, X)\}) \mid \gamma(F\pi x) = a,\, \delta(F\pi_1(F\pi x)) = d\},\, \lambda(a,(d,x)).\, d) \\ \cong & (\lambda a.\, \{x : F\{(D \times A, X)\} \mid \gamma(F\pi x) = a\},\, \lambda(a,x).\, \delta(F\pi_1(F\pi x))) \end{array}$$

by the definition of $\overline{\gamma,\delta}$. So, by the comment after Theorem 2.1, $\Psi(\mu F^{\overline{\gamma,\delta}}) \cong \mu F^{\gamma,\delta}$. But since $\Psi(\mu F^{\overline{\gamma,\delta}})$ is the same as (7.4), we have that (7.4) can indeed be inductively characterised as $\mu F^{\gamma,\delta}$. □

It is also possible to state and prove a generalisation of Theorem 7.2 in the general setting of a full cartesian Lawvere category with very strong coproducts, as defined in Section 3. In this case, we make use of the category $\mathcal{E}_A \times_{\mathcal{B}} \mathcal{B}/D$, which is defined by a pullback construction similar to that in Section 7.2. The use of very strong coproducts is essential to proving the generalised analogue of the adjoint equivalence in Theorem 7.1. In the general fibrational setting, we have the following definition of $F^{\gamma,\delta}$:

$$F^{\gamma,\delta}(X,f) = (\Sigma_f(\hat{F}_{D\times A}(\Phi(X,f))), \delta \circ F\pi_1 \circ F\pi_{\Phi(X,f)})$$

The formulation of zygomorphic refinement in the general setting of a full cartesian Lawvere category with very strong coproducts means that we can use the process described in Section 5 to derive a fibration in which to perform zygomorphic refinement on indexed inductive types.

Example 7. Example 6 illustrates refinement by a paramorphism, but does not use the full generality of refinement by a zygomorphism. We now demonstrate the power of refinement by a zygomorphism to mechanically derive an inductive characterisation of the data type of lists of rational numbers indexed by their average.

We specialise the functor $F_{\mathtt{List}_B}$ from Example 1 to get the functor representing the type of lists of rational numbers: $F_{\mathtt{List}_{\mathbb{Q}}}X = 1 + \mathbb{Q} \times X$. We reuse the $F_{\mathtt{List}_B}$-algebra *lengthalg* : $F_{\mathtt{List}_B}\mathbb{N} \to \mathbb{N}$, also from Example 1, whose fold computes the length of a list. We also consider the following $F_{\mathtt{List}_{\mathbb{Q}}}$-algebra *sumalg*, which is used to compute the sum of the elements of a list:

$$\begin{array}{lcl} \mathit{sumalg} & : & F_{\mathtt{List}_{\mathbb{Q}}}\mathbb{Q} \to \mathbb{Q} \\ \mathit{sumalg}\ \mathsf{Nil} & = & 0 \\ \mathit{sumalg}\ (\mathsf{Cons}(q,s)) & = & q + s \end{array}$$

By the standard construction of the product of two F-algebras, we combine *lengthalg* and *sumalg* to produce the following single $F_{\mathtt{List}_{\mathbb{Q}}}$-algebra whose fold will simultaneously compute the sum and length of a list of rational numbers:

$$\mathit{sumlengthalg} : F_{\mathtt{List}_{\mathbb{Q}}}(\mathbb{Q} \times \mathbb{N}) \to \mathbb{Q} \times \mathbb{N}$$

This algebra will form the F-algebra component of the zygomorphism by which we will refine $\mu F_{\mathtt{List}_{\mathbb{Q}}}$.

The morphism component of the zygomorphism by which we will refine $\mu F_{\mathtt{List}_{\mathbb{Q}}}$ has carrier $1 + \mathbb{Q}$. Here, the non-$\mathbb{Q}$ case caters for empty lists, for which the average is not defined. We use empty and avg as mnemonics for the left and right injections into $1 + \mathbb{Q}$. The morphism *avg* is defined by

$$\begin{array}{lcl} \mathit{avg} & : & F_{\mathtt{List}_{\mathbb{Q}}}((\mathbb{Q} \times \mathbb{N}) \times (1 + \mathbb{Q})) \to 1 + \mathbb{Q} \\ \mathit{avg}\ \mathsf{Nil} & = & \mathsf{empty} \\ \mathit{avg}\ (\mathsf{Cons}(q, ((s, l), \_))) & = & \mathsf{avg}(\frac{q+s}{l+1}) \end{array}$$

Following a similar process to that in Example 6, we can now compute the refinement of $\mu F_{\mathtt{List}_{\mathbb{Q}}}$ by *sumlengthalg* and *avg*:

$$\begin{array}{l} F_{\mathtt{List}_{\mathbb{Q}}}^{\mathit{avg},\mathit{sumlengthalg}}(X, f) = \\ \quad (\lambda a.\ \{* \mid a = \mathsf{empty}\} + \{(q, a', x : Xa') \mid a = \mathsf{avg}(\frac{q+\pi_1(f(a',x))}{\pi_2(f(a',x))+1})\}, \\ \quad \lambda(a, x).\ \mathsf{case}\ x\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{Nil} & \Rightarrow & (0, 0) \\ \mathsf{Cons}(q, a', x) & \Rightarrow & (q + \pi_1(f(a', x)), \pi_2(f(a', x)) + 1)) \end{array} \right. \end{array}$$

In this definition, we have used $\pi_1(f(a', x))$ to obtain the sum of the list underlying x, and have likewise used $\pi_2(f(a', x))$ to obtain its length. Expressing this refinement in Agda 2 gives the following definition:

```
mutual
  data AvgList : 1 + Rational -> Set where
    nil  : AvgList empty
    cons : (q : Rational) ->
           {a : 1 + Rational} ->
           (x : AvgList a) ->
           AvgList (avg ((q + sum x) / (length x + 1)))

  sum : {a : 1 + Rational} -> AvgList a -> Rational
  sum nil        = 0
  sum (cons q x) = q + sum x

  length : {a : 1 + Rational} -> AvgList a -> Nat
  length nil        = 0
  length (cons q x) = length x + 1
```

The fact we have generated small indexed inductive-recursive types by a process of refinement by a zygomorphism leads to the interesting question of whether it is possible to further refine small indexed inductive-recursive types by any sort of refinement process. A thorough investigation of such processes should also involve large induction-recursion (recall that a large inductive-recursive type entails the definition of a Set-valued recursive function simultaneously with the inductive type). The setting of large inductive-recursive types is much more complicated than small (indexed) inductive-recursive types, and so we leave investigation of the refinement of general inductive-recursive types to future work. Recent work by Malatesta, Altenkirch, Ghani, Hancock and McBride [30] has shown that a large universe of small inductive-recursive types described by codes is equivalent to the universe of indexed containers [3]. This work may point to a way to formulate the development of

this section in terms of codes for functors describing types rather than directly in terms of the functors themselves.

Another interesting avenue for future work is to determine whether the partial refinement process of Section 6 can be combined with the zygomorphic refinement process presented in this section.

8. Conclusions, Applications, Related and Future Work

We have given a clean semantic framework for deriving refinements of inductive types that store computationally relevant information within the indices of the resulting refined types. We have also shown how already indexed types can be refined further, how refined types can be derived even when some elements of the original type do not have indices, and how refinement by zygomorphisms entails the use of small indexed induction-recursion for information hiding. In addition to its theoretical clarity, the theory of refinement we have developed has potential applications in the following areas:

Dependently Typed Programming: Often a user is faced with a choice between building properties of elements of data types into more sophisticated data types, or stating these properties externally as, say, pre- and post-conditions. While the former is clearly preferable because properties can then be statically type-checked, it also incurs an overhead which can deter its adoption. Supplying the programmer with infrastructure to produce refined types as needed can reduce this overhead.

Libraries: With the implementation of refinement, library implementers will no longer need to provide comprehensive collections of data types, but instead only methods for defining new data types. Our results also ensure that library implementers will not need to guess which refinement types will prove useful to programmers, and can instead focus on providing useful abstractions for creating more sophisticated data types from simpler ones.

Implementation: Current implementations of types such as `Vector` types store all index information. For example, a vector of length 3 will store the lengths 3, 2, and 1 of its subvectors. Since this can be very space-consuming, Brady *et al.* [9] have sought to determine when this information need not be stored in memory. Our work suggests that a refinement μF^{α} can be implemented by simply implementing the underlying type μF, since programs requiring indices can reconstruct these as needed. It could therefore provide a user-controllable tradeoff between space and time efficiency.

8.1. Related Work.

The work closest to that reported here is McBride's work on ornaments [32]. McBride defines a type of descriptions of inductive data types, along with a notion of one description "ornamenting" another. Despite the differences between our fibrational approach and his type-theoretic approach, the notion of refinement presented in Sections 4 and 5 is very similar to McBride's notion of an algebraic ornament. Ornamentation further allows for additional arbitrary data to be attached to constructors, something that is not possible with any of the refinement processes that we have discussed in this paper. On the other hand, ornamentation is restricted to inductive types and so does not allow for the generation of indexed inductive-recursive types that we presented in Section 7. The theory of ornamentation has been developed by Ko and Gibbons [28], who examine the relationship between the ornamental versions of the "local" and "global" refinement that we discussed in Section 1.2. More recently, Dagand and McBride [14] have described an

extension of McBride's original definition of ornamentation which allows for the removal of constructors. In our setting, the removal of constructors is possible with the use of partial refinement (Section 6).

An interesting question for future work is to determine the relationship between functions defined on data types and functions defined on refined versions of data types. This question has been addressed in the setting of McBride's work on ornaments by Ko and Gibbons [28] and also by Dagand and McBride [14]. We have not considered the question of refinement of functions in this paper, and we leave it as future work to determine whether or not the fibrational approach taken here can provide any insight.

Chuang and Lin [12] present a way to derive new indexed inductive types from existing inductive types and algebras that is very similar to our basic refinement process in Section 4. Chuang and Lin work in the setting of the codomain fibration, which makes some calculations easier, but extensions to partial and zygomorphic refinement more difficult.

A line of research allowing the programmer to give refined types to constructors of inductive data types was initiated by Freeman and Pfenning [21]. Freeman and Pfenning defined a variant of ML that allowed programmers to define refinements of inductive types by altering the types of constructors, or by disallowing the use of certain constructors. Refinement of this sort did not require dependent types. This work was later developed by Xi [37], Davies [15] and Dunfield [17] for extensions of ML-like languages with dependent types, and by Pfenning [35] and Lovas and Pfenning [29] for LF. The work of Kawaguchi *et al.* [27] is also similar. This research begins with an existing type system and provides a mechanism for expressing richer properties of values that are well-typeable in that type system. It is thus similar to the work reported here, although a major focus of the work of Freeman and Pfenning and its descendants is on the decidability of type checking and inference of refined types, which we have not considered in this paper. On the other hand, we formally prove that each refinement is isomorphic to the richer, property-expressing data type it is intended to capture, rather than leaving this to the programmer to justify on a refinement-by-refinement basis.

Refinement types have been used in other settings to give more precise types to programs in existing programming languages (but not specifically to inductive types). For example, Denney [16] and Gordon and Fournet [23] use subset types to refine the type systems of ML-like languages. Subset types are also used heavily in the PVS theorem prover [36].

Our results extend the systematic code reuse delivered by generic programming [2, 5, 7]: in addition to generating new programs we can also generate new types from existing types. This area is being explored in Epigram [11], with codes for data types being represented within a predicative intensional system. This enables programs to generate new data types. It should be possible to implement our refinement process using similar techniques.

In addition to the specific differences between our work and that discussed above, a distinguishing feature of ours is the semantic methodology we use to develop refinement. We believe that this methodology is new. We also believe that a semantic approach is important: it can serve as a principled foundation for refinement, as well as provide a framework in which to compare different implementations. Moreover, it may lead to new algebraic insights into refinement that complement the logical perspective of previous work.

Acknowledgements: We thank Conor McBride, Frank Pfenning, and Pierre-Evariste Dagand for helpful comments on this work. The anonymous FoSSaCS and LMCS reviewers also provided useful feedback. This work was funded by EPSRC grant EP/G068917/1.

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, C. McBride, and P. Morris. Generic programming with dependent types. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 209–257. Springer, 2007.

[3] T. Altenkirch and P. Morris. Indexed Containers. In A. Pitts, editor, *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science*, LICS 2009, pages 277–285. IEEE Computer Society, 2009.

[4] R. Atkey, P. Johann, and N. Ghani. When Is a Type Refinement an Inductive Type? In M. Hofmann, editor, *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 72–87. Springer, 2011.

[5] R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors. *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer, 1983.

[7] M. Benke, P. Dybjer, and P. Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

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

[9] E. Brady, C. McBride, and J. McKinna. Inductive Families Need Not Store Their Indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.

[10] A. Carboni, S. Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84(2):145–158, 1993.

[11] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In S. Weirich, editor, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14. ACM, 2010.

[12] T.-R. Chuang and J.-L. Lin. An algebra of dependent data types. Technical Report TR-IIS-06-012, Institute of Information Science, Academia Sinica, Taiwan, 2006.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[14] P.-É. Dagand and C. McBride. Transporting functions across ornaments. *The Computing Research Repository (CoRR)*, abs/1201.4801, 2012.

[15] R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005.

[16] E. Denney. Refinement types for specification. In D. Gries and W. P. de Roever, editors, *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, PROCOMET '98, pages 148–166. Chapman & Hall, Ltd., 1998.

[17] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007.

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

[19] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1-3):1–47, 2003.

[20] P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[21] T. Freeman and F. Pfenning. Refinement types for ML. In D. Wise, editor, *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277. ACM, 1991.

[22] N. Ghani, P. Johann, and C. Fumex. Fibrational induction rules for initial algebras. In A. Dawar and H. Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2010.

[23] A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, Microsoft Research, 2009.

[24] C. Hermida and B. Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation*, 145(2):107–152, 1998.

[25] B. Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169–207, 1993.

[26] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1999.

[27] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In A. Diwan, editor, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 304–315. ACM, 2009.

[28] H.-S. Ko and J. Gibbons. Modularising inductive families. In J. Järvi and S.-C. Mu, editors, *ACM SIGPLAN Workshop on Generic Programming (WGP)*, pages 13–24. ACM, 2011.

[29] W. Lovas and F. Pfenning. Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance. *Logical Methods in Computer Science*, 6(4), 2010.

[30] L. Malatesta, T. Altenkirch, N. Ghani, P. Hancock, and C. McBride. Small Induction Recursion, Indexed Containers and Dependent Polynomials are equivalent. Submitted for Publication, 2012.

[31] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen, 1990.

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

[33] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

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

[35] F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, 1993.

[36] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[37] H. Xi. Dependently typed data structures. Unpublished Note, Revision after WAAAPL '99, 2000.