Fibrational Induction Rules For Initial Algebras

By: Neil Ghani, **Patricia Johann**, and Clement Fumex

Abstract

This paper provides an induction rule that can be used to prove properties of data structures whose types are inductive, i.e., are carriers of initial algebras of functors. Our results are semantic in nature and are inspired by Hermida and Jacobs' elegant algebraic formulation of induction for polynomial data types. Our contribution is to derive, under slightly different assumptions, an induction rule that is generic over all inductive types, polynomial or not. Our induction rule is generic over the kinds of properties to be proved as well: like Hermida and Jacobs, we work in a general fibrational setting and so can accommodate very general notions of properties on inductive types rather than just those of particular syntactic forms. We establish the correctness of our generic induction rule by reducing induction to iteration. We show how our rule can be instantiated to give induction rules for the data types of rose trees, finite hereditary sets, and hyperfunctions. The former lies outside the scope of Hermida and Jacobs' work because it is not polynomial; as far as we are aware, no induction rules have been known to exist for the latter two in a general fibrational framework. Our instantiation for hyperfunctions underscores the value of working in the general fibrational setting since this data type cannot be interpreted as a set.

Neil Ghani, **Patricia Johann**, and Clement Fumex (2010) "Fibrational Induction Rules for Initial Algebras" *Proceedings, Computer Science Logic* (CSL'10), pp. 336 - 350.
Version Of Record Available At www.springer.com

1 Introduction

Iteration operators provide a uniform way to express common and naturally occurring patterns of recursion over inductive data types. Expressing recursion via iteration operators makes code easier to read, write, and understand; facilitates code reuse; guarantees properties of programs such as totality and termination; and supports optimising program transformations such as `fold` fusion and short cut fusion. Categorically, iteration operators arise from initial algebra semantics of data types, in which each data type is regarded as the carrier of the initial algebra of a functor F. Lambek's Lemma ensures that this carrier is the least fixed point μF of F, and initiality ensures that, given any F-algebra $h : FA \to A$, there is a unique F-algebra homomorphism, denoted $fold\, h$, from the initial algebra $in : F(\mu F) \to \mu F$ to that algebra. For each functor F, the map $fold : (FA \to A) \to \mu F \to A$ is the iteration operator for the data type μF. Initial algebra semantics thus provides a well-developed theory of iteration which is i) *principled*, and so helps ensure that programs have rigorous mathematical foundations that can be used to ascertain their meaning and correctness; ii) *expressive*, and so is applicable to all inductive types, rather than just syntactically defined classes of data types such as polynomial data types; and iii) *correct*, and so is valid in any model — set-theoretic, domain-theoretic, realisability, etc. — in which data types are interpreted as carriers of initial algebras.

Because induction and iteration are closely linked we may reasonably expect that initial algebra semantics can be used to derive a principled, expressive, and correct theory of induction for data types we well. In most treatments of induction, given a functor F together with a property P to be proved about data of type μF, the premises of the induction rule for μF constitute an F-algebra with carrier $\Sigma x : \mu F.\, Px$. The conclusion of the rule is obtained by supplying such an F-algebra as input to the *fold* for μF. This yields a function from μF to $\Sigma x : \mu F.\, Px$ from which function of type $\forall x : \mu F.\, Px$ can be obtained. It has not, however, been possible to characterise F-algebras with carrier $\Sigma x : \mu F.\, Px$ without additional assumptions on F. Induction rules are thus typically derived under the assumption that the functors involved have a certain structure, e.g., that they are polynomial. Moreover, taking the carriers of the algebras to be Σ-types assumes that properties are represented as type-valued functions. So while induction rules derived as described are both principled and correct, their expressiveness is limited along two dimensions: with respect to the data types for which they can be derived and the nature of the properties they can verify.

One principled and correct approach to induction is given by Hermida and Jacobs [6]. They lift each functor F on a base category of types to a functor $\hat{F}$ on a category of properties over those types, and take the premises of the induction rule for the type μF to be an $\hat{F}$-algebra. Hermida and Jacobs work in a fibrational setting and the notion of property they consider is, accordingly, very general. Indeed, they accommodate any notion of property that can be fibred over the category of types, and so overcome one of the two limitations mentioned above. On the other hand, their approach is only applicable to polynomial data types, so the limitation on the class of data types treated remains in their work.

This paper shows how to remove the restriction on the class of data types treated. *Our main result is a derivation of a generic induction rule that can be instantiated to* every *inductive type* — i.e., to *every* type which is the carrier of the initial algebra of a functor — regardless of whether it is polynomial. We take Hermida and Jacobs' approach as our point of departure and show that, under slightly different assumptions on the fibration involved, we can lift *any* functor on its base category that has an initial algebra to a functor on its properties. This is clearly an important theoretical result, but it also has practical consequences:

- We show in Example 2 how our generic induction rule can be instantiated to the codomain fibration to derive the rule for rose trees that one would intuitively expect. The data type of rose trees lies outside the scope of Hermida and Jacobs' results because it is not polynomial. On the other hand, an induction rule for rose trees is available in the proof assistant Coq, although it is neither the one we intuitively expect nor expressive enough to prove properties that ought to be amenable to inductive proof. The rule we derive for rose trees is indeed the expected one, which suggests that our derivation may enable automatic generation of more useful induction rules in Coq, rather than requiring the user to hand code them as is currently necessary.
- We further show in Example 3 how our generic induction rule can be instantiated, again to the codomain fibration, to derive a rule for the data type of

finite hereditary sets. This data type is defined in terms of quotients and so lies outside current theories of induction.
- Finally, we show in Example 4 how our generic induction rule can be instantiated to the subobject fibration over ωcpo to derive a rule for the data type of hyperfunctions. Because this data type cannot be interpreted as a set, a fibration other than the codomain fibration over Set is required; in this case, use of the subobject fibration allows us to derive an induction rule for admissible subsets of hyperfunctions. Moreover, the functor underlying the data type of hyperfunctions is not strictly positive, and this fact again underscores the advantage of being able to handle a very general class of functors. As far as we know, induction rules for finite hereditary sets and hyperfunctions have not previously existed in the general fibrational framework.

Although our theory of induction is applicable to all functors having initial algebras, including higher-order ones, our examples show that working in the general fibrational setting is beneficial even if attention is restricted to first-order functors. Note also that our induction rules coincide with those of Hermida and Jacobs when specialised to polynomial functors in the codomain fibration. But the structure we require of fibrations generally is slightly different from that required by Hermida and Jacobs, so while our theory is in essence a generalisation of theirs, the two are, strictly speaking, incomparable. The structure we require is, however, still minimal and certainly present in all standard fibrational models of type theory (see Section 4). Like Hermida and Jacobs, we prove our generic induction rule correct by reducing induction to iteration.

We take a purely categorical approach to induction in this paper, and derive our generic induction rule from only the initial algebra semantics of data types. As a result, our work is inherently extensional. While translating our constructions into intensional settings may therefore require additional effort, we expect the guidance offered by the categorical viewpoint to support the derivation of induction rules for functors that are not treatable at present. Since we do not use any form of impredicativity in our constructions, and instead use only the weaker assumption that initial algebras exist, this guidance will be widely applicable.

The remainder of this paper is structured as follows. To make our results as accessible as possible, we illustrate them in Section 2 with a categorical derivation of the familiar induction rule for the natural numbers. In Section 3 we derive an induction rule for the special case of the codomain fibration over Set, i.e., for functors on the category of sets and properties representable as set-valued predicates. We also show how this rule can be instantiated to derive the one from Section 2, and the ones for rose trees and finite hereditary sets mentioned above. Then, in Section 4 we sketch the general fibrational form of our derivation (space constraints prevent a full treatment) and illustrate the resulting generic induction rule with the aforementioned application to hyperfunctions. Section 5 concludes and offers some additional directions for future research.

When convenient, we identify isomorphic objects of a category and write $=$ rather than $\simeq$. We write 1 for the canonical singleton set and denote its single element by $\cdot$. In Sections 2 and 3 we assume that types are interpreted as objects in the category Set of sets, and so 1 also denotes the unit type in those sections.

2 A Familiar Induction Rule

Consider the inductive data type *Nat*, which defines the natural numbers and can be specified in a programming language with Haskell-like syntax by

$$data\ Nat\ =\ Zero\ \mid\ Succ\ Nat$$

The observation that *Nat* is the least fixed point of the functor N on Set defined by $NX\ =\ 1\ +\ X$ can be used to define the following iteration operator for it:

$$\begin{array}{lcl} foldNat & = & X \to (X \to X) \to Nat \to X \\ foldNat\ z\ s\ Zero & = & z \\ foldNat\ z\ s\ (Succ\ n) & = & s\ (foldNat\ z\ s\ n) \end{array}$$

Categorically, iteration operators such as *foldNat* arise from the initial algebra semantics of data types. If $\mathcal{B}$ is a category and F is a functor on $\mathcal{B}$, then an *F-algebra* is a morphism $h : FX \to X$ for some object X of $\mathcal{B}$. We call X the *carrier* of h. For any functor F, the collection of F-algebras itself forms a category Alg_F which we call the *category of F-algebras*. In Alg_F, an *F-algebra morphism* between F-algebras $h : FX \to X$ and $g : FY \to Y$ is a map $f : X \to Y$ such that $f \circ h = g \circ Ff$. When it exists, the initial F-algebra $in : F(\mu F) \to \mu F$ is unique up to isomorphism and has the least fixed point μF of F as its carrier. Initiality ensures that there is a unique F-algebra morphism $fold\ h : \mu F \to X$ from in to any other F-algebra $h : FX \to X$. This gives rise to the following iteration operator *fold* for F or, equivalently, for the inductive type μF:

$$\begin{array}{lcl} fold & : & (FX \to X) \to \mu F \to X \\ fold\ h\ (in\ t) & = & h\ (F\ (fold\ h)\ t) \end{array}$$

Since *fold* is derived from initial algebra semantics it is principled and correct. It is also expressive, since it can be defined for *every* inductive type. In fact, *fold* is a single iteration operator parameterised over inductive types rather than a family of iteration operators, one for each such type, and the iteration operator *foldNat* above is the instantiation to *Nat* of the generic iteration operator *fold*.

The iteration operator *foldNat* can be used to derive the standard induction rule for *Nat*. This rule says that if a property P holds for 0, and if P holds for $n{+}1$ whenever it holds for a natural number n, then P holds for all natural numbers. Representing each property of natural numbers as a predicate $P\ :\ Nat \to \mathsf{Set}$ mapping each $n : Nat$ to the set of proofs that P holds for n, we wish to represent this rule at the object level as a function *indNat* with type

$$\forall (P : Nat \to \mathsf{Set}).\ P\ Zero \to (\forall n : Nat.\ P\ n \to P\ (Succ\ n)) \to (\forall n : Nat.\ P\ n)$$

Code fragments such as the above, which involve quantification over sets, properties, or functors, are to be treated as "categorically inspired" within this paper. This is because quantification over such higher-kinded objects cannot be interpreted in Set. In order to give a formal interpretation to code fragments like the one above, we would need to work in a category such as that of modest sets. The ability to work with functors over categories other than Set is one of the

motivations for working in the general fibrational setting of Section 4. Of course, the use of category theory to suggest computational constructions has long been accepted within the functional programming community (see, e.g., [1, 2, 13]).

A function *indNat* with the above type takes as input the property P to be proved, a proof ϕ that P holds for *Zero*, and a function ψ mapping each $n : Nat$ and each proof that P holds for n to a proof that P holds for $Succ\, n$, and returns a function mapping each $n : Nat$ to a proof that P holds for n. We can write *indNat* in terms of *foldNat* — and thus reduce induction for *Nat* to iteration for *Nat* — as follows. First note that *indNat* cannot be obtained by instantiating the type X in the type of *foldNat* to a type of the form Pn for a specific n because *indNat* returns elements of the types $P\, n$ for different values n and these types are, in general, distinct from one another. We therefore need a type containing all of the elements of $P\, n$ for every n. Such a type can be formally given by the dependent type $\Sigma n : Nat.\, P\, n$ comprising pairs (n, p) where $n : Nat$ and $p : P\, n$.

The standard approach to defining *indNat* is thus to apply *foldNat* to an N-algebra with carrier $\Sigma n : Nat.\, P\, n$. Such an algebra has components $\alpha : \Sigma n : Nat.\, P\, n$ and $\beta : \Sigma n : Nat.\, P\, n \rightarrow \Sigma n : Nat.\, P\, n$. Given $\phi : P\, Zero$ and $\psi : \forall n.\, P\, n \rightarrow P\, (Succ\, n)$, we choose $\alpha = (Zero, \phi)$ and $\beta\, (n, p) = (Succ\, n, \psi\, n\, p)$ and note that $foldNat\, \alpha\, \beta : Nat \rightarrow \Sigma n : Nat.\, P\, n$. We tentatively take $indNat\ P\ \phi\ \psi\ n$ to be p, where $foldNat\, \alpha\, \beta\, n = (m, p)$. But in order to know that p actually gives a proof for n itself, we must show that $m = n$. Fortunately, this follows easily from the uniqueness of $foldNat\, \alpha\, \beta$. Letting π'_P be the second projection on dependent pairs, the induction rule for *Nat* is

$$\begin{aligned} indNat\ &:\ \forall (P : Nat \rightarrow \mathsf{Set}).\ P\ Zero \rightarrow (\forall n : Nat.\, P\, n \rightarrow P\, (Succ\, n)) \\ &\qquad \rightarrow (\forall n : Nat.\, P\, n) \\ indNat\ P\ \phi\ \psi\ &=\ \pi'_P\ \circ (foldNat\ \ (Zero, \phi)\ \ (\lambda (n, p).\, Succ\, n, \psi\, n\, p)) \end{aligned}$$

The use of dependent types is fundamental to this formalization of the induction rule for *Nat*, but this is only possible because properties are taken to be set-valued functions. The remainder of this paper uses fibrations to generalise the above treatment of induction to arbitrary functors which have initial algebras and arbitrary properties which are fibred over the category whose objects interpret types. In the general fibrational setting, properties are given axiomatically via the fibrational structure rather than assumed to be (set-valued) functions.

3 Induction rules over Set

The main result of this paper is the derivation of an induction rule that is generic over inductive types and can be used to verify any notion of property that is fibred over the category whose objects interpret types. In the remainder of the paper we restrict attention to functors that have initial algebras. In this section we further assume that types are modelled by sets, so the functors we consider are on Set and the properties we consider are functions mapping data to sets of proofs that these properties hold for them. We make these assumptions to present our derivation in the simplest setting possible. But they are not always valid, so we derive a more general induction rule which relaxes them in Section 4. It should be more easily digestible once the derivation in this section is understood.

The derivation for *Nat* in Section 2 suggests that an induction rule for an inductive data type μF should, in general, look something like this:

$$ind \;:\; \forall P : \mu\, F \to \mathsf{Set}.\;\; ???\;\; \to\;\; \forall\, x : \mu F.\, P\, x$$

But what should the premises — denoted ??? here — of the generic induction rule *ind* be? Since we want to construct, for any term $x : \mu F$, a proof term of type $P\,x$ from proof terms for x's substructures, and since the functionality of the *fold* operator for μF is precisely to compute a value for $x : \mu F$ from the values for x's substructures, it is natural to try to equip P with an F-algebra structure that can be input to *fold* to yield a mapping of each $x : \mu F$ to an element of $P\,x$. But this approach quickly hits a snag. Since the codomain of every predicate $P : \mu F \to \mathsf{Set}$ is Set itself, rather than an object of Set, F cannot be applied to P as is needed to equip P with an F-algebra structure. Moreover, an induction rule for μF cannot be obtained by applying *fold* to an F-algebra with carrier $P\,x$ for any specific x.

Such considerations led Hermida and Jacobs [6] to define a category of predicates $\mathcal{P}$ and a lifting $\hat{F}$ for every polynomial functor F on Set to a functor $\hat{F}$ on $\mathcal{P}$ that respects the structure of F. They then constructed $\hat{F}$-algebras with carrier P to serve as the premises of their induction rules. Their construction is very general: they consider functors on bicartesian categories rather than just on Set, and represent properties by bicartesian fibrations over such categories instead of using the specific notion of predicate from Definition 2 below. On the other hand, they define liftings for polynomial functors only. In this section we focus exclusively on functors on Set and a particular category of predicates, and show how to define a lifting for all functors on Set, including non-polynomial ones. In this setting our results properly extend those of Hermida and Jacobs, thus catering for a variety of data types that they cannot treat.

Definition 1 *Let X be a set. A* predicate on X *is a pair (X, P) where $P : X \to \mathsf{Set}$ maps each $x \in X$ to a set $P\,x$. We call X the* domain *of the predicate (X, P).*

Definition 2 *The* category of predicates $\mathcal{P}$ *has predicates as its objects. A morphism from a predicate (X, P) to a predicate (X', P') is a pair $(f, f^{\sim}) : (X, P) \to (X', P')$ of functions, where $f : X \to X'$ and $f^{\sim} : \forall x : X.\, P\,x \to P'(f\,x)$.*

The notion of a morphism from (X, P) to (X', P') does not require the sets of proofs $P\,x$ and $P'\,(f\,x)$, for any $x \in X$, to be *equal*. Instead, it requires only the existence of a function $f^{\sim}$ which maps, for each x, each proof in $P\,x$ to a proof in $P'\,(f\,x)$. We denote by $U : \mathcal{P} \to \mathsf{Set}$ the *forgetful functor* mapping each predicate (X, P) to its domain X and each predicate morphism $(f, f^{\sim})$ to f.

An alternative to Definition 2 would take the category of predicates to be the arrow category over Set, but the natural lifting in this setting does not generalise to arbitrary fibrations. Indeed, if properties are modelled as functions, then every functor can be applied to a property, and hence every functor can be its own lifting. In the general fibrational setting, properties are not necessarily modelled by functions, so a functor cannot, in general, be its own lifting. The decision not

to use arrow categories to model properties is thus dictated by our desire to lift functors in such a way that it can be extended to the general fibrational setting.

Definition 3 *Let F be a functor on* Set. *A* lifting *of F from* Set *to $\mathcal{P}$ is a functor $\hat{F}$ on $\mathcal{P}$ such that $FU = U\hat{F}$.*

Note that if P is a predicate on X, then $\hat{F}P$ is a predicate on FX. We can now derive the standard induction rule from Section 2 for *Nat* as follows.

Example 1 *The data type of natural numbers is μN where N is the functor on* Set *defined by $N\,X = 1+X$. If P is a predicate on X, then a lifting $\hat{N}P : 1+X \to$* Set *of N from* Set *to $\mathcal{P}$ is given by $\hat{N}P\,(inl\,\cdot) = 1$ and $\hat{N}P\,(inr\,n) = P\,n$. An $\hat{N}$-algebra with carrier $P : Nat \to$* Set *can be given by $in : 1 + Nat \to Nat$ and $in^{\sim} : \forall t : 1{+}Nat.\ \hat{N}P\,t \to P(in\,t)$. Since $in\,(inl\,\cdot) = 0$ and $in\,(inr\,n) = n{+}1$, we see that $in^{\sim}$ is an element $h_1 : P\,0$ and a function $h_2 : \forall n : Nat.\ P\,n \to P\,(n{+}1)$. Thus, the second component $in^{\sim}$ of the $\hat{N}$-algebra with carrier $P : Nat \to$* Set *and first component in gives the premises of the familiar induction rule in Example 1.*

The notion of predicate comprehension is a key ingredient of our lifting.

Definition 4 *Let P be a predicate on X. The* comprehension *of P, denoted $\{P\}$, is the type $\Sigma x : X.\,P\,x$ comprising pairs (x, p) where $x : X$ and $p : Px$. The map taking each predicate P to $\{P\}$, and taking each predicate morphism $(f, f^{\sim}) : P \to P'$ to $\{(f, f^{\sim})\} : \{P\} \to \{P'\}$ defined by $\{(f, f^{\sim})\}(x, p) = (fx, f^{\sim}x\,p)$, defines the* comprehension functor *$\{-\}$ from $\mathcal{P}$ to* Set.

Definition 5 *If F is a functor on* Set, *then the lifting $\hat{F}$ is the functor on $\mathcal{P}$ given as follows. For every predicate P on X, $\hat{F}\,P : F\,X \to$* Set *is defined by $\hat{F}\,P = (F\,\pi_P)^{-1}$, where the natural transformation $\pi : \{-\} \to U$ is given by $\pi_P\,(x, p) = x$. For every predicate morphism $f : P \to P'$, $\hat{F}f = (k, k^{\sim})$ where $k = F(Uf)$, and $k^{\sim} : \forall y : FX.\,\hat{F}P\,y \to \hat{F}P'\,(k\,y)$ is given by $k^{\sim}\,y\,z = F\{f\}z$.*

The inverse image f^{-1} of $f : X \to Y$ is a predicate $P : Y \to$ Set. Thus if P is a predicate on X, then $\hat{F}P$ is a predicate on FX, so $\hat{F}$ is a lifting of F from Set to $\mathcal{P}$. The lifting $\hat{F}$ captures an "all" modality generalising Haskell's `all` function on lists to arbitrary data types. A similar modality is given in [12] for indexed containers.

The lifting in Example 1 is the instantiation of the construction in Definition 5 to $NX = 1+X$ on Set. Indeed, if P is any predicate, then $\hat{N}\,P = (N\,\pi_P)^{-1}$, i.e., $\hat{N}\,P = (id + \pi_P)^{-1}$, by Definition 5. Since the inverse image of the coproduct of functions is the coproduct of their inverse images, since $id^{-1}\,1 = 1$, and since $\pi_P^{-1}n = \{(n, p) \mid p : Pn\}$ for all n, we have $\hat{N}\,P\,(inl\,\cdot) = 1$ and $\hat{N}\,P\,(inr\,n) = P\,n$.

The rest of this section shows that F-algebras with carrier $\{P\}$ are interderivable with $\hat{F}$-algebras with carrier P, and then uses this result to derive our induction rule.

Definition 6 *The functor K_1 : Set $\to \mathcal{P}$ maps each set X to the predicate $K_1X = \lambda x : X.\,1$ and each $f : X \to Y$ to the predicate morphism $(f, \lambda x : X.\,id)$.*

The predicate K_1X is called the *truth predicate on* X. For every $x : X$, the set K_1Xx of proofs that K_1X holds for x is a singleton, and thus is non-empty. For any functor F, the lifting $\hat{F}$ maps the truth predicate on a set X to that on FX.

Lemma 1 *For any functor* F *on* Set *and any* $X : \mathsf{Set}$, $\hat{F}(K_1X) = K_1(FX)$.

Proof: By Definition 5, $\hat{F}(K_1X) = (F\pi_{K_1X})^{-1}$. We have that π_{K_1X} is a isomorphism since there is only one proof of K_1X for each $x : X$, and thus that $F\,\pi_{K_1X}$ is an isomorphism as well. As a result, $(F\,\pi_{K_1X})^{-1}$ maps every $y : FX$ to a singleton set, and therefore $\hat{F}(K_1X) = (F\pi_{K_1X})^{-1} = \lambda y : FX.\, 1 = K_1(FX)$.

The fact that $K_1 \dashv \{-\}$ is critical to the constructions below. This is proved in [6]; we include its proof here for completeness and to establish notation. The description of comprehension as a right adjoint can be traced back to Lawvere [10].

Lemma 2 K_1 *is left adjoint to* $\{-\}$.

Proof: We must show that, for any predicate P and any set Y, the set $\mathcal{P}(K_1Y, P)$ of morphisms from K_1Y to P in $\mathcal{P}$ is in bijective correspondence with the set $\mathsf{Set}(Y, \{P\})$ of morphisms from Y to $\{P\}$. Define maps $(-)^\dagger : \mathsf{Set}(Y, \{P\}) \to \mathcal{P}(K_1Y, P)$ and $(-)^\# : \mathcal{P}(K_1Y, P) \to \mathsf{Set}(Y, \{P\})$ by $h^\dagger = (h_1, h_2)$ where $hy = (v, p)$, $h_1y = v$ and $h_2y = p$, and $(k, k^\sim)^\# = \lambda(y : Y).\,(ky, k^\sim y)$. These give a natural isomorphism between $\mathsf{Set}(Y, \{P\})$ and $\mathcal{P}(K_1Y, P)$.

Naturality of $(-)^\dagger$ ensures that $(g \circ f)^\dagger = g^\dagger \circ K_1 f$ for all $f : Y' \to Y$ and $g : Y \to \{P\}$. Similarly for $(-)^\#$. Moreover, $id^\dagger$ is the counit of the adjunction between K_1 and $\{-\}$. These observations are used in the proof of Lemma 4.

Lemma 3 *There is a functor* $\Phi : Alg_F \to Alg_{\hat{F}}$ *such that if* $k : FX \to X$, *then* $\Phi k : \hat{F}(K_1X) \to K_1X$.

Proof: For an F-algebra $k : FX \to X$ define $\Phi k = K_1 k$, and for two F-algebras $k : FX \to X$ and $k' : FX' \to X'$ and an F-algebra morphism $h : X \to X'$ between them define the $\hat{F}$-algebra morphism $\Phi h : \Phi k \to \Phi k'$ by $\Phi h = K_1 h$. Then $K_1(FX) = \hat{F}(K_1X)$ by Lemma 1, so that Φk is an $\hat{F}$-algebra and $K_1 h$ is an $\hat{F}$-algebra morphism. It is easy to see that Φ preserves identities and composition.

Lemma 4 *The functor* Φ *has a right adjoint* Ψ *such that if* $j : \hat{F}P \to P$, *then* $\Psi j : F\{P\} \to \{P\}$.

Proof: We construct $\Psi : Alg_{\hat{F}} \to Alg_F$ as follows. Given an $\hat{F}$-algebra $j : \hat{F}P \to P$, we use the fact that $\hat{F}(K_1\{P\}) = K_1(F\{P\})$ by Lemma 1 to define $\Psi j : F\{P\} \to \{P\}$ by $\Psi j = (j \circ \hat{F}\, id^\dagger)^\dagger$. To specify the action of Ψ on an $\hat{F}$-algebra morphism h, define $\Psi h = \{h\}$. Clearly Ψ preserves identity and composition.

Next we show $\Phi \dashv \Psi$, i.e., for every F-algebra $k : FX \to X$ and $\hat{F}$-algebra $j : \hat{F}P \to P$ with P a predicate on X, there is a natural isomorphism between F-algebra morphisms from k to Ψj and $\hat{F}$-algebra morphisms from Φk to j. First

observe that an F-algebra morphism from k to Ψj is a map from X to $\{P\}$, and an $\hat{F}$-algebra morphism from Φk to j is a map from $K_1 X$ to P. An isomorphism between such maps is given by the adjunction $K_1 \dashv \{-\}$ from Lemma 2, and so is natural. We must check that $f : X \to \{P\}$ is an F-algebra morphism from k to Ψj iff $f^\dagger : K_1 X \to P$ is an $\hat{F}$-algebra morphism from Φk to j.

So assume $f : X \to \{P\}$ is an F-algebra morphism from k to Ψj, i.e., $f \circ k = \Psi j \circ F f$. We must prove that $f^\dagger \circ \Phi k = j \circ \hat{F} f^\dagger$. By the definition of Φ in Lemma 3, thus amounts to showing $f^\dagger \circ K_1 k = j \circ \hat{F} f^\dagger$. Since $(-)^\dagger$ is an isomorphism, f is an F-algebra morphism iff $(f \circ k)^\dagger = (\Psi j \circ F f)^\dagger$. Naturality of $(-)^\dagger$ ensures that $(f \circ k)^\dagger = f^\dagger \circ K_1 k$ and that $(\Psi j \circ F f)^\dagger = (\Psi j)^\dagger \circ K_1(F f)$, so the previous equality holds iff $f^\dagger \circ K_1 k = (\Psi j)^\dagger \circ K_1(F f)$. But

$$
\begin{array}{ll}
& j \circ \hat{F} f^\dagger \\
= & j \circ \hat{F}(id^\dagger \circ K_1 f) \qquad \text{by naturality of } (-)^\dagger \text{ and } f = id \circ f \\
= & (j \circ \hat{F}\, id^\dagger) \circ \hat{F}(K_1 f) \qquad \text{by the functoriality of } \hat{F} \\
= & (\Psi j)^\dagger \circ K_1(F f) \qquad \text{by the definition of } \Psi, \text{ the fact that } (-)^\dagger \text{ and } (-)^\# \\
& \qquad\qquad \text{are inverses, and Lemma 1} \\
= & f^* \circ K_1 k \qquad \text{by the observation immediately preceding this proof} \\
= & f^\dagger \circ \Phi k \qquad \text{by the definition of } \Phi
\end{array}
$$

So $f^\dagger \circ K_1 k = (\Psi j)^\dagger \circ K_1(F f)$, and $f^\dagger$ is an $\hat{F}$-algebra morphism from Φk to j.

Lemma 4 ensures that F-algebras with carrier $\{P\}$ are interderivable with $\hat{F}$-algebras with carrier P. For example, the N-algebra $[\alpha, \beta]$ with carrier $\{P\}$ from Section 2 can be derived from the $\hat{N}$-algebra with carrier P given in Example 1. Since we define a lifting $\hat{F}$ for any functor F, Lemma 4 thus shows how to construct F-algebras with carrier $\Sigma x : \mu F.\, P x$ for any F.

We can now derive our generic induction rule. For every predicate P on X and every $\hat{F}$-algebra $(k, k^\sim) : \hat{F} P \to P$, Lemma 4 ensures that Ψ constructs from $(k, k^\sim)$ an F-algebra with carrier $\{P\}$. Thus, $fold\ (\Psi\,(k, k^\sim)) : \mu F \to \{P\}$ and this map decomposes into two parts: $\phi = \pi_P \circ fold\ (\Psi\,(k, k^\sim)) : \mu F \to X$ and $\psi : \forall (t : \mu F).\, P\,(\phi\, t)$. Initiality of in ensures $\phi = fold\, k$. This gives the following generic induction rule for the type X, which reduces induction to iteration:

$$
\begin{array}{lcl}
genind & : & \forall\, (F : \mathsf{Set} \to \mathsf{Set})\ (P : X \to \mathsf{Set})\ ((k, k^\sim) : (\hat{F} P \to P))\ (x : \mu F). \\
& & \qquad P\,(fold\, k\, x) \\
genind\, F\, P & = & \pi'_P \circ fold \circ \Psi
\end{array}
$$

When $X = \mu F$ and $k = in$, initiality of in further ensures that $\phi = fold\, in = id$, and thus that *genind* specialises to the expected induction rule for an inductive data type μF:

$$
\begin{array}{lcl}
ind & : & \forall\, (F : \mathsf{Set} \to \mathsf{Set})\ (P : \mu F \to \mathsf{Set})\ ((k, k^\sim) : (\hat{F} P \to P)). \\
& & \qquad (k = in) \to \forall (x : \mu F).\, P\, x \\
ind\, F\, P & = & \pi'_P \circ fold \circ \Psi
\end{array}
$$

This rule can be instantiated to familiar rules for polynomial data types, as well as to ones we would expect for data types such as rose trees and finite hereditary sets, both of which lie outside the scope of standard methods.

Example 2 *The data type of rose trees is given in Haskell-like syntax by*

$$data\ Rose = Node(Int, List\ Rose)$$

The functor underlying Rose is $FX = Int \times List\, X$ *and its induction rule is*

$$\begin{array}{lcl} indRose & : & \forall\,(P : Rose \to \mathsf{Set})\ ((k, k^{\sim}) : (\hat{F}P \to P)).\\ & & \quad (k = in) \to \forall(x : Rose).\, P\, x \\ indRose\, F\, P & = & \pi'_P \circ fold \circ \Psi \end{array}$$

Calculating $\hat{F}P = (F\pi_P)^{-1} : F\, Rose \to \mathsf{Set}$*, and writing* $xs\,!!\,k$ *for the* k^{th} *component of a list* xs*, we have that*

$$\begin{array}{l} \quad \hat{F}\, P\, (i, rs) \\ = \{z : F\{P\} \mid F\pi_p z = (i, rs)\} \\ = \{(j, cps) : Int \times List\, \{P\} \mid F\pi_P(j, cps) = (i, rs)\} \\ = \{(j, cps) : Int \times List\, \{P\} \mid (id \times List\, \pi_P)(j, cps) = (i, rs)\} \\ = \{(j, cps) : Int \times List\, \{P\} \mid j = i\ and\ List\ \pi_P\ cps = rs\} \\ = \{(j, cps) : Int \times List\, \{P\} \mid j = i\ and\ \forall k < length\, cps.\, \pi_P\, (cps\,!!\,k) = rs\,!!\,k\} \end{array}$$

An $\hat{F}$*-algebra whose underlying* F*-algebra is* $in : F\, Rose \to Rose$ *is thus a pair of functions* $(in, k^{\sim})$*, where* $k^{\sim}$ *has type*

$$\begin{array}{l} \quad \forall i : Int.\ \forall rs : List\ Rose. \\ \qquad \{(j, cps) : Int \times List\, \{P\} \mid j = i\ and\ \forall k < length\, cps.\, \pi_P\, (cps\,!!\,k) = rs\,!!\,k\} \\ \qquad\quad \to\ P(Node\, (i, rs)) \\ = \forall i : Int.\ \forall rs : List\ Rose. \\ \qquad \{cps : List\, \{P\} \mid \forall k < length\, cps.\, \pi_P\, (cps\,!!\,k) = rs\,!!\,k\}\ \to\ P\, (Node\, (i, rs)) \\ = \forall i : Int.\ \forall rs : List\ Rose.\ (\forall k < length\ rs.\, P\, (rs\,!!\,k)) \to P(Node\, (i, rs)) \end{array}$$

The last equality is due to surjective pairing for dependent products and the fact that $length\, cps = length\, rs$*. The type of* $k^{\sim}$ *gives the hypotheses of the induction rule for rose trees.*

Example 3 *Hereditary sets are sets whose elements are themselves sets, and so are the core data structures within set theory. The data type* HS *of finite hereditary sets is* μP_f *for the finite powerset functor* P_f*. If* $P : X \to \mathsf{Set}$*, then* $P_f\pi_P : P_f(\Sigma x : X.Px) \to P_f X$ *maps each set* $\{(x_1, p_1), \ldots, (x_n, p_n)\}$ *to the set* $\{x_1, \ldots, x_n\}$*, so that* $(P_f\pi_P)^{-1}$ *maps a set* $\{x_1, \ldots, x_n\}$ *to the set* $Px_1 \times \ldots \times Px_n$*. A* $\hat{P}_f$*-algebra with carrier* $P : HS \to \mathsf{Set}$ *and first component in therefore has as its second component a function of type*

$$\forall(\{s_1, \ldots, s_n\} : P_f(HS)).\, Ps_1 \times \ldots \times Ps_n \to P(in\{s_1, \ldots, s_n\})$$

The induction rule for finite hereditary sets is thus

$$\begin{array}{l} indHS :: (\forall(\{s_1, \ldots, s_n\} : P_f(HS)).\, Ps_1 \times \ldots \times Ps_n \to P(in\{s_1, \ldots, s_n\})) \\ \qquad\qquad \to \forall(s : HS).P\, s \end{array}$$

4 Induction rules in the fibrational setting

We can treat a more general notion of predicate using fibrations. We motivate the move from the codomain fibration to arbitrary fibrations by observing that i) the semantics of data types in languages involving recursion and other effects usually involves categories other than Set; ii) in such circumstances, the notion of a predicate can no longer be taken as a function with codomain Set; iii) even when working in Set there are reasonable notions of "predicate" other than that in Section 3 (for example, a predicate on a set X could be a subobject of X); and iv) when, in future work, we come to consider induction rules for data types such as nested types, GADTs, indexed containers, and dependent types (see Section 5), we will want to appropriately instantiate a general theory of induction rather than having to invent a new one. Thus, although we could develop an *ad hoc* theory for each choice of category, functor, and predicate, it is far preferable to develop a uniform, axiomatic approach that is widely applicable.

Fibrations support precisely this kind of axiomatic approach, so this section generalises the constructions of the previous one to the general fibrational setting. The standard model of type theory based on locally cartesian closed categories does arise as a specific fibration — namely, the codomain fibration — but the general fibrational setting is far more flexible. In locally cartesian closed models of type theory, predicates and types coexist in the same category, so a functor can be taken to be its own lifting. In the general setting, predicates are not simply functions or morphisms, properties and types do not coexist in the same category, and a functor cannot be taken to be its own lifting. There is no choice but to construct a lifting. Details about fibrations can be found in standard references such as [8, 14].

Working in the general fibrational setting also facilitates a direct comparison of our work with that of Hermida and Jacobs [6]. The main difference is that they use fibred products and coproducts in defining their liftings, whereas we use left adjoints to reindexing functors instead. The codomain fibration over Set has both, so their derivation gives exactly the same induction rule as ours in the setting of Section 3.

Let $U : \mathcal{E} \to \mathcal{B}$ be a fibration. Objects of the total category $\mathcal{E}$ can be thought of as properties, objects of the base category $\mathcal{B}$ can be thought of as types, and U can be thought of as mapping each property E in $\mathcal{E}$ to the type UE of which E is a property. One fibration U can capture many different properties of the same type, so U is not injective on objects. For any object B of $\mathcal{B}$, we write $\mathcal{E}_B$ for the *fibre above* B, i.e., for the subcategory of $\mathcal{E}$ consisting of objects E such that $UE = B$ and morphisms k between objects of $\mathcal{E}_B$ such that $Uk = id_B$. Let $f^{\S}_E$ be the cartesian morphism determined by f and E. Then $f^{\S}_E$ is unique up to isomorphism for every choice of object E and morphism f with codomain UE. If $f : B \to B'$ is a morphism in $\mathcal{B}$, then the *reindexing functor induced by* f is the functor $f^* : \mathcal{E}_{B'} \to \mathcal{E}_B$ defined on objects by $f^*E = dom(f^{\S}_E)$ and, for a morphism $k : E \to E'$ in $\mathcal{E}_{B'}$, f^*k is the morphism such that $k \circ f^{\S}_E = f^{\S}_{E'} \circ f^*k$. The universal property of $f^{\S}_{E'}$ ensures the existence and uniqueness of f^*k.

Proceeding by analogy with the situation for Set-based predicates — where Set plays the role of $\mathcal{B}$ and $\mathcal{P}$ plays the role of $\mathcal{E}$ — we now define, for every functor F on $\mathcal{B}$, a lifting $\hat{F}$ of F to $\mathcal{E}$ such that $U\hat{F} = FU$. We construct $\hat{F}$ by generalising each aspect of the construction of Section 3 to the general setting.

• *The Truth Functor* To construct the truth functor in the general setting, we assume $\mathcal{B}$ and $\mathcal{E}$ have terminal objects $1_\mathcal{B}$ and $1_\mathcal{E}$, respectively, such that $U(1_\mathcal{E}) = 1_\mathcal{B}$. Writing $!_B$ for the unique map from an object B of $\mathcal{B}$ to $1_\mathcal{B}$, we define $K_1 : \mathcal{B} \to \mathcal{E}$ by setting $K_1 B = (!_B)^* 1_\mathcal{E}$ and, for a morphism $k : B \to B'$, taking $K_1 k$ to be the unique morphism guaranteed to exist by the universal property of $k^\S$. Then, for every B in $\mathcal{B}$, $K_1 B$ is the terminal object of $\mathcal{E}_B$, so $U(K_1 B) = B$. In fact, $U \dashv K_1$, and the unit of this adjunction is id, so $UK_1 = id$.

• *Comprehension* Recall from Lemma 2 that the comprehension functor of Section 3 is right adjoint to the truth functor. Since right adjoints are defined up to isomorphism, in the general fibrational setting we can define the comprehension functor $\{-\}$ to be the right adjoint to the truth functor K_1. A fibration $U : \mathcal{E} \to \mathcal{B}$ which has a right adjoint K_1 which itself has a right adjoint $\{-\}$ is called a *comprehension category* [8]. We henceforth restrict attention to comprehension categories. We write ϵ for the counit of the adjunction $\{-\} \vdash K_1$ and so, for any object E of $\mathcal{E}$, we have that $\epsilon_E : K_1\{E\} \to E$.

• *Projection* Recall from Section 3 that the first step of the construction of our lifting is to define the projection π_P mapping the comprehension of a predicate P to P's domain UP. As in Section 3, we also want comprehension to be a natural transformation, so we actually seek to construct $\pi : \{-\} \to U$ for an arbitrary comprehension category. Since $\epsilon_E : K_1\{E\} \to E$ for every object E of $\mathcal{E}$, we have that $U\epsilon_E : UK_1\{E\} \to UE$. Because $UK_1 = id$, defining $U\epsilon$ by $(U\epsilon)_E = U\epsilon_E$ gives a natural transformation from $\{-\}$ to U. We may therefore define the projection in an arbitrary comprehension category by $\pi = U\epsilon$.

• *Inverses* The final step in defining $\hat{F}$ is to turn each component $F\pi_E$ of the natural transformation $F\pi : F\{-\} \to FU$ defined by $(F\pi)_E = F\pi_E$ into a predicate over FUE. In Section 3, this was done via an inverse image construction. To generalise it, first note that we can construct a predicate $invf$ in $\mathcal{E}_{B'}$ for any map $f : B \to B'$ in $\mathcal{B}$ if we assume a small amount of additional standard fibrational structure, namely that for each such f the functor $f^* : \mathcal{E}_{B'} \to \mathcal{E}_B$ has a left adjoint. As in [6], no Beck-Chevalley condition is required on this adjoint, which we denote $\Sigma_f : \mathcal{E}_B \to \mathcal{E}_{B'}$. We define $invf$ to be $\Sigma_f(K_1 B)$.

• *The Lifting* Putting this all together, we now define the lifting $\hat{F} : \mathcal{E} \to \mathcal{E}$ by $\hat{F}E = \Sigma_{F\pi_E}(K_1(F\{E\}))$ for every object E of $\mathcal{E}$. For completeness we also give the action of $\hat{F}$ on morphisms. For each $k : E \to E'$, define $\hat{F}k = (FUk)^\S \, \alpha_{K_1 F\{E'\}} \, \Sigma_{F\pi_E}\gamma(K_1 F\{E\})$. Here, i) $\alpha_{K_1 F\{E'\}} : \Sigma_{F\pi E}(F\{k\})^* K_1 F\{E'\} \to (FU\{k\})^* \Sigma_{F\pi_{E'}} K_1 F\{E'\}$ is the component for $K_1 F\{E'\}$ of the natural transformation from $\Sigma_{F\pi E}(F\{k\})^*$ to $(FU\{k\})^* \Sigma_{F\pi_{E'}}$ arising from the facts that $\Sigma_{F\pi_E}$ is the left adjoint of $(F\pi_E)^*$ and that $F\pi$ is a natural transformation, and ii) $\gamma : \Sigma_{F\pi_E} K_1 F\{E\} \to \Sigma_{F\pi E}(F\{k\})^* K_1 F\{E'\}$ is the isomorphism arising from the fact that $(F\{k\})^*$ is a right adjoint by the existence of $\Sigma_{F\{k\}}$ and hence preserves terminal objects. It is trivial to check that $\hat{F}$ is indeed a lifting.

• *Generalising Lemma 1* As in Section 3, we ultimately want to show that F-algebras with carrier $\{P\}$ are interderivable with $\hat{F}$-algebras with carrier P. We first show that, as in Lemma 1, $\hat{F}(K_1 B) = K_1(FB)$ for any functor F on $\mathcal{B}$ and B in $\mathcal{B}$. Recall that $UK_1 = id$ and define $\pi K_1 : \{-\}K_1 \to Id$ to be the natural transformation with components $(\pi K_1)_B = \pi_{K_1 B}$. Note that $((\pi K_1)_B)^{-1} = UK_1\eta_B$, where $\eta : Id \to \{-\}K_1$ is the unit of the adjunction $\{-\} \vdash K_1$, so that πK_1 is a natural isomorphism. If we further define $F\pi K_1 : F\{-\}K_1 \to FUK_1$ to be the natural transformation with components $(F\pi K_1)_B = F((\pi K_1)_B)$, then $F\pi K_1$ is also a natural isomorphism. We will use this observation below to show that, for every object B of $\mathcal{B}$, $\Sigma_{(F\pi K_1)_B}$ is not only left adjoint by definition, but also right adjoint, to $((F\pi K_1)_B)^*$. Then, observing that right adjoints preserve terminal objects and that $K_1(FB)$ is the terminal object of $\mathcal{E}_{FB}$ (since $K_1 B$ is the terminal object of $\mathcal{E}_B$ for any B), we will have shown that $\hat{F}(K_1 B)$ — i.e., $\Sigma_{(F\pi K_1)_B}(K_1(F\{K_1 B\}))$ — must be the terminal object of $\mathcal{E}_{FU(K_1 B)}$, i.e., of $\mathcal{E}_{FB}$. In other words, we will have shown that $\hat{F}(K_1 B) = K_1(FB)$.

So, fix an object B of $\mathcal{B}$. To see that $\Sigma_{(F\pi K_1)_B} \vdash ((F\pi K_1)_B)^*$, first note that, for any isomorphism $f : B \to B'$ in $\mathcal{B}$, f^* and $(f^{-1})^*$ both exist and both $f^* \vdash (f^{-1})^*$ and $(f^{-1})^* \vdash f^*$ hold. Then, since $f^* \vdash \Sigma_f$ by definition, we have Σ_f is $(f^{-1})^*$, and thus that $\Sigma_f \vdash f^*$. Instantiating f to $(F\pi K_1)_B$ and recalling that $(F\pi K_1)_B$ is an isomorphism, we have that $\Sigma_{(F\pi K_1)_B} \vdash ((F\pi K_1)_B)^*$.

• *A Generic Fibrational Induction Rule* Analogues of Lemma 3 and Lemma 4 hold in the general fibrational setting provided all occurrences of Set are replaced by $\mathcal{B}$ and all occurrences of $\mathcal{P}$ are replaced by $\mathcal{E}$ and, in the analogue of Lemma 2, $(-)^\dagger : \mathcal{B}(B, \{E\}) \to \mathcal{E}(K_1 B, E)$ and $(-)^\# : \mathcal{E}(K_1 B, E) \to \mathcal{B}(B, \{E\})$ are defined by the adjunction $\{-\} \vdash K_1$.

The above construction thus yields the following generalised induction rule:

$$\begin{array}{lcl} \mathit{genfibind} & : & \forall\, (F : \mathcal{B} \to \mathcal{B})\ (E : \mathcal{E}_X)\ (k : \hat{F}E \to E).\, \mu F \to \{E\} \\ \mathit{genfibind}\, F\, E & = & \mathit{fold} \circ \Psi \end{array}$$

This induction rule looks slightly different from the one for set-valued predicates. In Section 3, we were able to use the specific structure of comprehensions for set-valued predicates to extract proofs for particular data elements from them. But in the general fibrational setting, predicates, and hence comprehensions, are left abstract, so we take the return type of the general induction scheme *genfibind* to be a comprehension. We expect that, when *genfibind* is instantiated to a fibration of interest, we should be able to use knowledge about that fibration to extract from the comprehension it constructs further proof information relevant to the application at hand. This expectation is justified, as in [6], by $\{-\} \vdash K_1$.

We now give an induction rule for a data type and properties that cannot be modelled in Set.

Example 4 *The fixed point $Hyp = \mu F$ of the functor $FX = (X \to Int) \to Int$ is the data type of hyperfunctions. Since F has no fixed point in* Set*, we interpret it in the category $\omega CPO_\perp$ of ω-cpos with $\perp$ and strict continuous monotone functions. In this setting, a property of an object X of $\omega CPO_\perp$ is an admissible*

sub-$\omega CPO_\perp$ A of X. Admissibility means that the bottom element of X is in A and A is closed under the least upper bounds of X. This structure forms a Lawvere category [7, 8]. The truth functor maps X to X, and comprehension maps a sub-$\omega CPO_\perp$ A of X to A. The lifting $\hat{F}$ maps a sub-$\omega CPO_\perp$ A of X to the sub-$\omega CPO_\perp$ FA of FX. Finally, the derived induction rule states that if A is an admissible sub-$\omega CPO_\perp$ of Hyp, and if $\hat{F}(A) \subseteq A$, then $A = Hyp$.

5 Conclusion and future work

We give an induction rule that can be used to prove properties of data structures of inductive types. Like Hermida and Jacobs, we give a fibrational account of induction, but we derive, under slightly different assumptions on fibrations, a generic induction rule that can be instantiated to *any* inductive type rather than just to polynomial ones. This rule is based on initial algebra semantics of data types, and is parameterised over both the data types and the properties involved. It is also principled, expressive, and correct. Our derivation yields the same induction rules as Hermida and Jacobs' when specialised to polynomial functors in the codomain fibration, but it also gives induction rules for non-polynomial data types such as rose trees, and for data types such as finite hereditary sets and hyperfunctions, for which no induction rules have previously been known.

There are several directions for future work. The most immediate is to instantiate our theory to give induction rules for nested types. These are exemplified by the data type of perfect trees given in Haskell-like syntax as follows:

$$\begin{array}{l} \mathit{data}\ \mathit{PTree}\ a : \mathsf{Set}\ \mathit{where} \\ \quad \mathit{PLeaf} : a \to \mathit{PTree}\ a \\ \quad \mathit{PNode} : \mathit{PTree}\ (a, a) \to \mathit{PTree}\ a \end{array}$$

Nested types arise as least fixed points of rank-2 functors; for example, the type of perfect trees is μH for the functor H given by $HF = \lambda X.\ X + F(X \times X)$. An appropriate fibration for induction rules for nested types thus takes $\mathcal{B}$ to be the category of functors on Set, $\mathcal{E}$ to be the category of functors from Set to $\mathcal{P}$, and U to be postcomposition with the forgetful functor from Section 3. A lifting $\hat{H}$ of H is given by $\hat{H}\ P\ X\ (\mathit{inl}\ a) = 1$ and $\hat{H}\ P\ X\ (\mathit{inr}\ n) = P\ (X \times X)\ n$. Taking the premise to be an $\hat{H}$-algebra gives the following induction rule for perfect trees:

$$\begin{array}{l} \mathit{indPTree} : \forall\ (P : \mathsf{Set} \to \mathcal{P}). \\ \qquad (UP = \mathit{PTree}) \to (\forall (X : \mathsf{Set})(x : X).\ P\ (\mathit{PLeaf}\ x)) \to \\ \qquad (\forall (X : \mathsf{Set})(t : \mathit{PTree}\ (X \times X).\ P\ (X \times X)\ t \to P\ (\mathit{PNode}\ t))) \to \\ \qquad\quad \forall (X : \mathsf{Set})(t : \mathit{PTree}\ X).\ P\ X\ t \end{array}$$

This rule can be used to show, for example, that PTree is a functor.

Extending the above instantiation for the codomain fibration to "truly nested types" and fibrations is current work. We expect to be able to instantiate our theory for truly nested types, GADTs, indexed containers, and dependent types, but initial investigations show care is needed. We must ascertain which fibrations can model predicates on such types, since the codomain fibration may not

give useful induction rules, as well as how to translate the rules to which these fibrations give rise to an intensional setting.

Matthes [11] gives induction rules for nested types (including truly nested ones) in an intensional type theory. He handles only rank-2 functors that underlie nested types (while we handle any functor of any rank with an initial algebra), but his insights may help guide choices of fibrations for truly nested types. These may in turn inform choices for GADTs, indexed containers, and dependent types.

Induction rules can automatically be generated in many type theories. Within the Calculus of Constructions [3] an induction rule for a data type can be generated solely from the inductive structure of that type. Such generation is also a key idea in the Coq proof assistant [4]. But as far as we know, generation can currently be done only for syntactic classes of functors rather than for all functors with initial algebras. In some type theories induction schemes are added as axioms rather than generated. For example, attempts to generate induction schemes based on Church encodings in the Calculus of Constructions proved unsuccessful and so initiality was added to the system, thus giving the Calculus of Inductive Constructions. Whereas Matthes' work is based on concepts such as impredicativity and induction recursion rather than initial algebras, ours reduces induction to initiality, and may therefore help lay the groundwork for extending implementations of induction to more sophisticated data types.

References

1. R. S. Bird and O. De Moor. Algebra of Programming. International Series in Computing Science, volume 100. Prentice Hall, 1997.
2. R. Bird and L. Meertens. Nested Datatypes. Proceedings, Mathematics of Program Construction, pp. 52–67, 1998.
3. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation* 76 (2-3), pp. 95–120, 1988.
4. The Coq Proof Assistant. Available at `coq.inria.fr`
5. N. Ghani and P. Johann. Foundations for Structured Programming with GADTs. Proceedings, Principles of Programming Languages, pp. 297–308, 2008.
6. C. Hermida and B. Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation* 145 (2), pp. 107–152, 1998.
7. B. Jacobs. Comprehension Categories and the Semantics of Type Dependency. *Theoretical Computer Science* 107, pp. 169–207, 1993.
8. B. Jacobs. *Categorical Logic and Type Theory.* North Holland, 1999.
9. P. Johann and N. Ghani. Initial Algebra Semantics is Enough! Proceedings, Typed Lambda Calculus and Applications, pp. 207–222, 2007.
10. F. W. Lawvere. Equality in Hyperdoctrines and Comprehension Scheme as an Adjoint Functor. *Applications of Categorical Algebra*, pp. 1–14, 1970.
11. R. Matthes. An Induction Principle for Nested Datatypes in Intensional Type Theory. *Journal of Functional Programming* 19 (3&4), pp. 439–468, 2009.
12. P. Morris. *Constructing Universes for Generic Programming.* Dissertation, University of Nottingham, 2007.
13. E. Moggi. Notations of Computation and Monads. *Information and Computation* 93 (1), pp. 55–92, 1991.
14. D. Pavlovic. *Predicates and Fibrations.* Dissertation, University of Utrecht, 1990.