When Is A Type Refinement An Inductive Type?

By: Robert Atkey, **Patricia Johann** and Neil Ghani

Abstract

Dependently typed programming languages allow sophisticated properties of data to be expressed within the type system. Of par-ticular use in dependently typed programming are indexed types that refine data by computationally useful information. For example, the 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 often store redundant information about data and their refinements. This paper shows how to generically derive inductive characterisations of refinements of inductive types, and argues that these characterisations can alleviate some of the aforementioned difficulties associated with ad hoc refinements. These 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.

Robert Atkey, **Patricia Johann**, and Neil Ghani. (2011) "When is a Type Refinement an Inductive Type?" *Proceedings, Foundations of Software Science and Computation Structures* (FoSSaCS'11), pp. 72 - 87. Version Of Record Available From www.springer.com

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 about them. One particularly promising approach is to parameterise, or *index*, types by extra information that can be used to express properties of data having those types. For example, most functional languages support a standard list data type parameterised over the type of the data the lists contain, but for some applications it is also crucial to know the length of a list. We may wish, for instance, to ensure that the list argument to the `tail` function has non-zero length — i.e., is non-empty — or that the lengths of the two list arguments to `zip` are the same.

A data type that equips each list with its length can be defined in the dependently typed language Agda [25] by

```
data Vector (B : Set) : Nat -> Set where
  VNil  : Vector B Z
  VCons : (n : Nat) -> B -> Vector B n -> Vector B (S n)
```

This declaration inductively defines a data type `Vector` which, for each choice of element type `B`, is indexed by natural numbers and has two constructors: `VNil`, which constructs a vector of `B`-data of length zero (i.e., `Z`), and `VCons`, which

* This work was funded by EPSRC grant EP/G068917/1.

constructs from an index `n`, an element of `B`, and a vector of `B`-data of length `n`, a new vector of `B`-data of length $\mathtt{n}+1$ (i.e., `S n`). The inductive type `Vector` can be used to define functions on lists which are "length-aware" in a way that functions which process data of standard list types cannot be. For example, length-aware `tail` and `zip` functions can be given via the following types and definitions:

```
tail : (n : Nat) -> Vector B (S n) -> Vector B n
tail (VCons b bs) = bs

zip  : (n : Nat) -> Vector B n -> Vector C n -> Vector (B x C) n
zip  VNil          VNil        = VNil
zip (VCons b bs) (VCons c cs) = VCons (b , c) (zip bs cs)
```

Examples such as those above 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 of the techniques developed for structured programming with and principled reasoning about inductive types — such as those championed in the Algebra of Programming [6] 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 inductive types be systematically augmented with computationally relevant information to give indexed inductive types that store computationally relevant information in their indices? If so, how?*

That is, how can we *refine* a given inductive type to get a new such type, called a *refinement*, that associates with each element of the given type its index?

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

```
data List (B : Set) : Set where
  Nil  : List B
  Cons : B -> List B -> List B
```

and its length function

```
length : List B -> Nat
length Nil        = Z
length (Cons _ l) = S (length l)
```

and construct the following refinement type of indexed lists:

$$\mathtt{IdxList\ B\ n} \cong \{\mathtt{x} : \mathtt{List\ \ B} \mid \mathtt{length\ x} = \mathtt{n}\} \tag{1}$$

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. But the construction suffers from a serious drawback: the resulting refinement — `IdxList` here — need not be inductive, and so is not a solution to the fundamental question posed above.

We propose an alternative construction of refinements that provides a comprehensive answer to the fundamental question raised above in the case when the given refinement function is computed by structural recursion over the data type to be refined. This is often the case in practice. More specifically, we 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. The characterisation of the refinement of μF by α as the inductive type μF^α allows the entire arsenal of structured programming techniques to be brought to bear on them. This construction 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 from the index of the input data structure.

The functor F^α that we construct is intimately connected with the generic structural induction rule for the inductive type μF [15,17]. This is perhaps not surprising: structural induction proves properties of functions defined by structural recursion on elements of inductive types. If the values of those functions are abstracted into the indices of associated indexed inductive types, then the computation of those values need no longer be performed during inductive proofs. In essence, we have shifted work away from computation and onto data. Refinement thus supports reasoning by structural induction "up to" the index of a term.

In this paper, we use the language of category theory to state and develop our results because it allows a high degree of precision and economy. Although we have developed our theory in the abstract setting of fibrations [19], in this paper we specialise to the families fibration over the category of sets to improve accessibility and give useful concrete intuitions. The remainder of this paper is structured as follows. In Section 2 we recall basic categorical preliminaries. In Section 3 we introduce a framework within which refinement may be developed [15,17]. We describe our basic refinement technique in Section 4 and illustrate it with several examples. In Section 5 we show how to refine inductive types which are themselves indexed. 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. Finally, Section 7 discusses applications and future and related work.

2 Inductive Types and *F*-algebras

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

```
data Tree : Set where
```

```
Leaf : Integer -> Tree
Node : (Tree x Tree) -> Tree
```

Inductive types can also be understood in terms of the categorical notion of an F-algebra. If $\mathbb{C}$ is a category and $F : \mathbb{C} \to \mathbb{C}$ is a functor, then an *F-algebra* is a pair $(A, \alpha : FA \to A)$ comprising an object A of $\mathbb{C}$ and a morphism $\alpha : FA \to A$ in $\mathbb{C}$. 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, since the carrier is present in the type of this map.

An *F-algebra homomorphism* from $(\alpha : FA \to A)$ to $(\alpha' : FB \to B)$ is a morphism $f : A \to B$ of $\mathbb{C}$ 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'. The initial F-algebra is unique up to isomorphism, and Lambek's Lemma further ensures that it is itself 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 $\llparenthesis \alpha \rrparenthesis_F : \mu F \to A$ for the unique morphism from $(in_F : F(\mu F) \to \mu F)$ to any F-algebra $(\alpha : FA \to A)$. We write $\llparenthesis - \rrparenthesis$ for $\llparenthesis - \rrparenthesis_F$ when F is clear from context. Of course, not all functors have least fixed points. For instance, the functor $FX = (X \to 2) \to 2$ on Set does not have any fixed point at all.

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, if $(\alpha : \mathbb{Z} + A \times A \to A)$ is an $F_{\texttt{Tree}}$-algebra, then $\llparenthesis \alpha \rrparenthesis : \texttt{Tree} \to A$ is exactly the application of the standard iteration function `fold` for trees to α (actually, to an "unbundling" of α into replacement functions, one for each of $F_{\texttt{Tree}}$'s constructors). More generally, for each functor F, the map $\llparenthesis - \rrparenthesis_F : (FA \to A) \to \mu F \to A$ is the iteration function for μF.

If F is a functor on $\mathbb{C}$, we write Alg_F for the category of all F-algebras and F-algebra homomorphisms between them. Identities and composition in Alg_F are taken directly from $\mathbb{C}$. The existence of initial F-algebras is equivalent to the existence of initial objects in Alg_F. Recall that an *adjunction* between two categories $\mathbb{C}$ and $\mathbb{D}$ consists of a left adjoint functor L and a right adjoint functor R and an isomorphism natural in A and X between the set $\mathbb{C}(LA, X)$ of morphisms in $\mathbb{C}$ from LA to X and the set $\mathbb{D}(A, RX)$ of morphisms in $\mathbb{D}$ 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 write $L \dashv R$.

We will make much use of the following theorem from [17]:

Theorem 1. *If $F : \mathbb{C} \to \mathbb{C}$ and $G : \mathbb{D} \to \mathbb{D}$ are functors, $L \dashv R$, and $FL \cong LG$ is a natural isomorphism, then* $\begin{tikzcd} \mathbb{C} \arrow[r, bend right, "R"'] \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "L"'] \end{tikzcd}$ *lifts to* $\begin{tikzcd} \mathrm{Alg}_F \arrow[r, bend right, "R'"'] \arrow[r, phantom, "\perp"] & \mathrm{Alg}_G \arrow[l, bend right, "L'"'] \end{tikzcd}$ *.*

Theorem 1 will be useful in conjunction with the fact that left adjoints preserve colimits, and thus preserve initial objects. In the setting of the theorem, if G has an initial algebra, then so does F. 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 FL and LG. Then the initial F-algebra is given by applying L' to the initial G-algebra, and so $\mu F = L(\mu G)$.

3 A Framework for Refinement

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. 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, with Pa representing 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 P holds for a. When Pa is inhabited, P is said to hold for a. When Pa is empty, P is said not to hold for a.

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. For each set A, the category $\mathrm{Fam(Set)}_A$ consists of families (A, P) and morphisms $(f, f^\sim)$ between them such that $f = id_A$. We call $\mathrm{Fam(Set)}_A$ the *fibre* of the families fibration over A. A function $f : A \to B$ contravariantly generates a *re-indexing functor* $f^* : \mathrm{Fam(Set)}_B \to \mathrm{Fam(Set)}_A$ which maps (B, Q) to $(A, Q \circ f)$.

3.1 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. This object is called the *truth predicate* for A. The mapping of objects to their truth predicates extends to a functor $K_1 : \mathrm{Set} \to \mathrm{Fam(Set)}$, called the *truth functor*. In addition, for each family (A, P) we can define the *comprehension* of (A, P), denoted $\{(A, P)\}$, to be the set $\{(a, p) \mid a \in A, p \in Pa\}$. The mapping of families to their comprehensions extends to a functor $\{-\} : \mathrm{Fam(Set)} \to \mathrm{Set}$, called the *comprehension functor*, and we end up with the following pleasing collection of adjoint relationships:

$$\begin{tikzcd}[row sep=large]
\mathrm{Fam(Set)} \arrow[d, "U"'] \arrow[d, phantom, shift left=2, "\dashv"] \arrow[d, bend left=70, "\{-\}"] \\
\mathrm{Set} \arrow[u, bend right=30, "K_1\dashv"']
\end{tikzcd} \tag{2}$$

The families fibration U is thus a *comprehension category with unit* [18,19]. 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 $\{(A, P)\}$. In fact, U is *full*, i.e., the functor from Fam(Set) to $\mathrm{Set}^\to$ induced by π is full and faithful.

3.2 Indexed Coproducts and Indexed Products

For each function $f : A \to B$ and family (A, P), we can form the family $(B, \lambda b.\ \Sigma_{a \in A}.\ (b = fa) \times Pa)$, called the *indexed coproduct of* (A, P) *along* f. The mapping of each family to its indexed coproduct along f extends to a functor $\Sigma_f : \mathrm{Fam(Set)}_A \to \mathrm{Fam(Set)}_B$ which is left adjoint to the re-indexing functor f^*. 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-re-indexing* functors. These functors 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-re-indexing commutes with re-indexing [19]. A bifibration which is also a full comprehension category with unit is called a *full cartesian Lawvere category* [18].

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

$$\begin{tikzcd}
\mathrm{Fam(Set)}_B \arrow[r, "f^*" description] & \mathrm{Fam(Set)}_A \arrow[l, bend right=40, "\Sigma_f"'] \arrow[l, bend left=40, "\Pi_f"] \arrow[l, phantom, shift right=3, "\perp"] \arrow[l, phantom, shift left=3, "\perp"]
\end{tikzcd}$$

Like its counterpart for coproducts, the Beck-Chevalley condition for products is often required. However, we do not make use of this condition in this paper.

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

$$\begin{tikzcd}
\{(A, P)\} \arrow[r, "\{\psi\}"] \arrow[d, "\pi_{(A,P)}"'] & \{(B, \Sigma_f(A, P))\} \arrow[d, "\pi_{\Sigma_f(A,P)}"] \\
A \arrow[r, "f"] & B
\end{tikzcd} \tag{3}$$

where ψ is the obvious map of families of sets over f, $\{\psi\}$ is an isomorphism. This definition of strong coproducts naturally generalises the usual one [19], and imposes a condition which is standard in models of type theory.

3.3 Liftings

A *lifting* of a functor $F : \mathrm{Set} \to \mathrm{Set}$ is a functor $\hat{F} : \mathrm{Fam(Set)} \to \mathrm{Fam(Set)}$ such that $FU = U\hat{F}$. A lifting is *truth-preserving* if it satisfies $K_1 F \cong \hat{F} K_1$. Truth-preserving liftings for all polynomial functors — i.e., for all functors built from identity functors, constant functors, coproducts, and products — are given in [17]. Truth-preserving liftings were established for arbitrary functors in [15]. The truth-preserving lifting $\hat{F}$ is defined on objects by

$$\hat{F}(A, P) = (FA, \lambda a.\ \{x : F\{(A, P)\} \mid F\pi_{(A,P)} x = a\}) = \Sigma_{F\pi_{(A,P)}} K_1(F\{(A, P)\}) \tag{4}$$

The final expression is written point-free using the constructions of Sections 3.1 and 3.2. A similar construction is given in a different setting by [22].

Since $\hat{F}$ is an endofunctor on Fam(Set), the category $\mathrm{Alg}_{\hat{F}}$ of $\hat{F}$-algebras exists. The families fibration $U : \mathrm{Fam(Set)} \to \mathrm{Set}$ extends to a fibration $U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$, called the *algebras fibration* induced by U. Moreover, writing K_1^{Alg} and $\{-\}^{\mathsf{Alg}}$ for the truth and comprehension functors, respectively, for U^{Alg}, the adjoint relationships from Diagram 2 all lift to give $U^{\mathsf{Alg}} \dashv K_1^{\mathsf{Alg}} \dashv \{-\}^{\mathsf{Alg}}$. The two adjunctions here follow from Theorem 1 using the fact that $\hat{F}$ is a lifting and therefore preserves truth. That left adjoints preserve initial objects can now be used to establish the following fundamental result from [15,17]:

Theorem 2. *$K_1(\mu F)$ is the carrier $\mu\hat{F}$ of the initial $\hat{F}$-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

$$(A, \lambda a : A.\ \{x : \mu F \mid (\!|\alpha|\!)x = a\}) \tag{5}$$

is inductively characterised as μF^α where $F^\alpha : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_A$ is

$$F^\alpha = \Sigma_\alpha \hat{F} \tag{6}$$

An alternative, set-theoretic presentation of F^α is:

$$F^\alpha(A, P) = (A, \lambda a.\ \{x : F\{(A, P)\} \mid \alpha(F\pi_{(A,P)}x) = a\}) \tag{7}$$

That is, $F^\alpha(A, P)$ is obtained by first building the FA-indexed type $\hat{F}(A, P)$ (cf. Equation 4), and then restricting membership to those elements whose α-values are correctly computed from those of their immediate subterms. The proof consists of the following three theorems, which are, as far as we are aware, new.

Theorem 3. *For each F-algebra $(\alpha : FA \to A)$, $(\mathrm{Alg}_{\hat{F}})_\alpha \cong \mathrm{Alg}_{F^\alpha}$.*

Proof. First note that $(\mathrm{Alg}_{\hat{F}})_\alpha$ is isomorphic to the category $(\hat{F} \downarrow \alpha^*)$ whose objects are morphisms from $\hat{F}(A, P)$ to $\alpha^*(A, P)$ in $\mathrm{Fam}(\mathrm{Set})_{FA}$ and whose morphisms are commuting squares. Then $(\hat{F} \downarrow \alpha^*) \cong \mathrm{Alg}_{F^\alpha}$ because $\Sigma_\alpha \dashv \alpha^*$.

Theorem 3 can be used to prove the following key result:

Theorem 4. *$U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$ is a bifibration.*

Proof. That U^{Alg} is a fibration, indeed a comprehension category with unit, is proved in [17]. Next, let f be an F-algebra morphism from $\alpha : FA \to A$ to $\beta : FB \to B$. We must show that the reindexing functor $f^{*\mathsf{Alg}}$ in U^{Alg} has a left adjoint Σ_f^{Alg}. Such an adjoint can be defined using Theorem 1, Theorem 3, and $F^\beta \Sigma_f \cong \Sigma_f F^\alpha$. By Equation 6, the latter is equivalent to $\Sigma_\beta \hat{F} \Sigma_f \cong \Sigma_f \Sigma_\alpha \hat{F}$. From the definition of $\hat{F}$, we must show that for all (A, P) in $\mathrm{Fam}(\mathrm{Set})_A$,

$$\Sigma_\beta \Sigma_{F\pi_{\Sigma_f(A,P)}} K_1 F\{\Sigma_f(A, P)\} \ \cong\ \Sigma_f \Sigma_\alpha \Sigma_{F\pi_{(A,P)}} K_1 F\{(A, P)\} \tag{8}$$

To see that this is the case, consider the following diagram:

$$\begin{tikzcd}
F\{(A,P)\} \arrow[r, "F\pi_{(A,P)}"] \arrow[d, "F\{\psi\}"'] & FA \arrow[r, "\alpha"] \arrow[d, "Ff"] & A \arrow[d, "f"] \\
F\{\Sigma_f(A,P)\} \arrow[r, "F\pi_{\Sigma_f(A,P)}"'] & FB \arrow[r, "\beta"'] & B
\end{tikzcd}$$

The left-hand square commutes because it is obtained by applying F to the naturality square for π, and the right-hand square commutes because f is an F-algebra morphism. Then $\Sigma_\beta \Sigma_{F\pi_{\Sigma_f(A,P)}} \Sigma_{F\{\psi\}} \cong \Sigma_f \Sigma_\alpha \Sigma_{F\pi_{(A,P)}}$ because opre-indexing preserves composition. Equation 8 now follows by applying both of these functors to $K_1 F\{(A,P)\}$, and then observing that $F\{\psi\}$ is an isomorphism since $\{\psi\}$ is one by assumption (cf. Diagram 3), so that $\Sigma_{F\{\psi\}}$ is a right adjoint (as well as a left adjoint) and thus preserves terminal objects.

We can now give an explicit characterisation for μF^α. We have

Theorem 5. *The functor F^α has an initial algebra with carrier $\Sigma_{(\!|\alpha|\!)} K_1(\mu F)$.*

Proof. The category $\mathrm{Alg}_{\hat{F}}$ has initial object whose carrier is $K_1(\mu F)$ by Theorem 2. Since U^{Alg} is a left adjoint and hence preserves initial objects, Proposition 9.2.2 of [19] ensures that the fibre $(\mathrm{Alg}_{\hat{F}})_\alpha$ — and so, by Theorem 3, that Alg_{F^α} — has an initial object whose carrier is $\Sigma_{(\!|\alpha|\!)} K_1(\mu F)$.

Instantiating Theorem 5 for Fam(Set) gives exactly the inductive characterisation of refinements we set out to find, namely that in Equation 5.

4.1 Some Specific Refinements

The following explicit formulas are used to compute refinements in the examples below. In the expression B^α, B is the constantly B-valued functor.

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

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

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

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

The refinement of μF_{List} by the algebra lengthalg is the least fixed point of

$$F_{List}^{lengthalg}(\mathbb{N}, P) = (\mathbb{N}, \lambda n.(n=0)\ +\ \{n_1 : \mathbb{N}, x_1 : B, x_2 : Pn_1 \mid n = n_1 + 1\})$$

This formulation of $\mu F_{List}^{lengthalg}$ is essentially the declaration `Vector` *from the introduction with the implicit equality constraints in that definition made explicit.*

Example 2 *We can similarly refine μF_{Tree} by the F_{Tree}-algebra*

$$\begin{aligned}
&\mathit{sum} &&: F_{\mathit{Tree}}\mathbb{Z} \to \mathbb{Z}\\
&\mathit{sum}\ (\mathsf{Leaf}\ z) &&= z\\
&\mathit{sum}\ (\mathsf{Node}\ (x_1, x_2)) &&= x_1 + x_2
\end{aligned}$$

which sums the values in a tree. This gives the refinement $\mu F^{\mathit{sum}}_{\mathit{Tree}}$, where

$$F^{\mathit{sum}}_{\mathit{Tree}}(\mathbb{Z}, P) = (\mathbb{Z}, \lambda n.\ \{z : \mathbb{Z} \mid z = n\} + \{n_1, n_2 : \mathbb{Z}, x_1 : Pn_1, x_2 : Pn_2 \mid n = n_1 + n_2\ \})$$

This corresponds to the Agda declaration

```
data IdxTree : Integer -> Set where
  IdxLeaf : (z : Integer) -> IdxTree z
  IdxNode : (l r : Integer) ->
                IdxTree l -> IdxTree r -> IdxTree (l + r)
```

Refinement by the initial algebra $(\mathit{in}_F : F(\mu F) \to \mu F)$ gives a μF-indexed type inductively characterised by $F^{\mathit{in}} = \Sigma_{\mathit{in}}\hat{F}$. Since *in* is an isomorphism, Σ_{in} is as well. Thus $F^{\mathit{in}} \cong \hat{F}$, so that $\mu F^{\mathit{in}} = \mu\hat{F} = K_1(\mu F)$, and each term is its own index. By contrast, refinement by the final algebra $(! : F1 \to 1)$ (which always exists because 1 is the terminal object of Set) gives a 1-indexed type inductively characterised by $F^!$. Since $F^! \cong F$, we have $\mu F^! = \mu F$, and all terms have index 1. Refining by the initial algebra thus has maximal discriminatory power, while refining by the terminal algebra has no discriminatory power.

5 Starting with Already Indexed Types

The development in Section 4 assumes 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) we needed, our results can be extended to *any* fibration having that structure. In particular, we can refine indexed types using a *change-of-base* [19]:

$$\begin{tikzcd}
\mathrm{Fam(Set)}_A \times_{\mathrm{Set}} \mathrm{Fam(Set)} \arrow[r] \arrow[d, "U^A"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathrm{Fam(Set)} \arrow[d, "U"] \\
\mathrm{Fam(Set)}_A \arrow[r, "\{-\}"] & \mathrm{Set}
\end{tikzcd}$$

This diagram, which is a pullback in Cat, the (large) category of categories and functors, generates a new fibration U^A from the functors U and $\{-\}$. The objects of $\mathrm{Fam(Set)}_A \times_{\mathrm{Set}} \mathrm{Fam(Set)}$ are (dependent) pairs $((A, P), (\{(A, P)\}, Y))$ of predicates. Thus, Y is "double indexed" by both $a \in A$ and $x \in Pa$.

The following theorem states that all the structure we require for our constructions is preserved by the change-of-base construction. It also generalises to any full cartesian Lawvere category with strong coproducts, so the change-of-base construction may be iterated.

Theorem 6. *U^A is a full cartesian Lawvere category with strong coproducts.*

Proof. First, U^A is a fibration by construction. The truth functor is defined by $K_1^A(A,P) = ((A,P), K_1\{(A,P)\})$ and the comprehension functor is defined by $\{((A,P),(\{(A,P)\},Y))\}^A = \Sigma_{\pi_{(A,P)}}(\{(A,P)\},Y)$. Coproducts are defined directly using the coproducts of U.

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

$$\begin{aligned} F_{wtexp}(\mathbb{T},P) = (\mathbb{T},\ \lambda t : \mathbb{T}.\ & \{z : \mathbb{Z} \mid t = \mathsf{int}\} + \{b : \mathbb{B} \mid t = \mathsf{bool}\} \\ & + \{t_1,t_2 : \mathbb{T},\ x_1 : Pt_1,\ x_2 : Pt_2 \mid t_1 = t_2 = t = \mathsf{int}\} \\ & + \{t_1,t_2,t_3 : \mathbb{T},\ x_1 : Pt_1,\ x_2 : Pt_2,\ x_3 : Pt_3 \mid \\ & \qquad\qquad t_1 = \mathsf{bool}, t_2 = t_3 = t\}) \end{aligned}$$

For any t*, write* IntConst*,* BoolConst*,* Add*, and* If *for the four injections into* $(snd\,(F_{wtexp}(\mathbb{T},P))\,t$*. Letting* $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ *denoting the set of booleans, and assuming there exist a* $\mathbb{T}$*-indexed family* T *such that* $T\ \mathsf{int} = \mathbb{Z}$ *and* $T\ \mathsf{bool} = \mathbb{B}$*, we have a semantic interpretation of the extended language's types. This can be used to specify a "tagless" interpreter by giving an* F_{wtexp}*-algebra:*

$$\begin{aligned} & eval : \ F_{wtexp}(\mathbb{T},T) \to (\mathbb{T},T) \\ & eval = (id, \lambda x : \mathbb{T}.\ \lambda t : snd\,(F_{wtexp}(\mathbb{T},T))\,x.\ case\ t\ of \\ & \qquad\qquad \mathsf{IntConst}\ z \qquad\qquad\quad \Rightarrow z \\ & \qquad\qquad \mathsf{BoolConst}\ b \qquad\qquad \Rightarrow b \\ & \qquad\qquad \mathsf{Add}\ (\mathsf{int},\mathsf{int},z_1,z_2) \quad \Rightarrow z_1 + z_2 \\ & \qquad\qquad \mathsf{If}\ (\mathsf{bool},t,t,b,x_1,x_2) \Rightarrow if\ b\ then\ x_1\ else\ x_2) \end{aligned}$$

Refining μF_{wtexp} *by eval yields a type indexed by* $\Sigma t : \mathbb{T}.\ Tt$*, i.e., by* $\{(\mathbb{T},T)\}$*. This type associates to every well-typed expression that expression's semantics.*

6 Partial Refinement

In Sections 4 and 5 we assumed that every element of an inductive type has an index that can be assigned to it. 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 examples show, the problem of refining subsets of inductive types is a common and naturally occurring one. Partial refinement is a technique for solving this problem.

The key idea underlying the required generalisation of our theory is to 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 μF_{exp} for the functor $F_{exp}X = \mathbb{Z}+\mathbb{B}+(X\times X)+(X\times X\times X)$. Letting $\mathbb{T} = \{\mathsf{int}, \mathsf{bool}\}$ as in Example 3 and using the obvious convention for naming the injections into $F_{exp}X$, such expressions can be type-checked using the following partial F_{exp}-algebra:*

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

Example 5 *Let $\mathbb{S} = \{\mathsf{R}, \mathsf{B}\}$ be a set of colours. The inductive type of coloured trees is μF_{ctree} for the functor $F_{ctree}X = 1 + \mathbb{S} \times X \times X$. We write Leaf and Br for injections into $F_{ctree}X$. Red-black trees [10] 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_{ctree}-algebra. Its carrier $\mathbb{S} \times \mathbb{N}$ records the colour of the tree 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.

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

The process of (total) refinement described in Section 4 constructs, from a functor F with initial algebra $(in_F : F(\mu F) \to \mu F)$ and an F-algebra $\alpha : FA \to A$, a functor F^α such that μF^α associates to each $x : \mu F$ its index $(\!|\alpha|\!)x$. If we can compute an index for each element of μF from a partial F-algebra, then we can apply the same technique to partially refine μF. The key to doing this is to turn every partial F-algebra into a (total) F-algebra. Let λ be any distributive law for the error monad M over the functor F. Then λ respects the unit and multiplication of M (see [4] for details), and

Lemma 1. *Every partial F-algebra $\kappa : FA \to 1 + A$ generates an F-algebra $\overline{\kappa} : F(1 + A) \to (1 + A)$ defined by $\overline{\kappa} = [\mathsf{fail}, \kappa] \circ \lambda_A$.*

Here, $[\mathsf{fail}, \kappa]$ is the cotuple of the functions fail and κ. 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, as our examples show, 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 $\mu F^{?\kappa}$ is the A-indexed type

$$(A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!) x = \mathsf{ok}\ a\}) = \mathsf{ok}^* \Sigma_{(\!|\overline{\kappa}|\!)} K_1(\mu F) = \mathsf{ok}^* \mu F^{\overline{\kappa}} \tag{9}$$

As we will see in Theorem 7, if

$$F^{?\kappa} = \mathsf{ok}^* \Sigma_{\overline{\kappa}} \hat{F} \tag{10}$$

then $\mu F^{?\kappa} = (A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!) x = \mathsf{ok}\ a\})$. Indeed, since left adjoints preserve initial objects, we can prove $\mu F^{?\kappa} \cong \mathsf{ok}^* \mu F^{\overline{\kappa}}$ by lifting the following adjunction to an adjunction between $\mathrm{Alg}_{F^{?\kappa}}$ and $\mathrm{Alg}_{F^{\overline{\kappa}}}$ via Theorem 1:

$$\begin{tikzcd} \mathrm{Fam}(\mathrm{Set})_A \arrow[r, bend right, "\Pi_{\mathsf{ok}}"'] \arrow[r, phantom, "\perp"] & \mathrm{Fam}(\mathrm{Set})_{1+A} \arrow[l, bend right, "\mathsf{ok}^*"'] \end{tikzcd}$$

To satisfy the precondition of Theorem 1, we prove that $\mathsf{ok}^* F^{\overline{\kappa}} \cong F^{?\kappa} \mathsf{ok}^*$ by first observing that if F preserves pullbacks, then $\hat{F}$ preserves re-indexing, i.e., for every function f, $\hat{F} f^* \cong (Ff)^* \hat{F}$. This is proved by direct calculation. Thus if F preserves pullbacks, and if

$$\mathsf{ok}^* \Sigma_{\overline{\kappa}} \cong \mathsf{ok}^* \Sigma_{\kappa} (F\mathsf{ok})^* \tag{11}$$

then $\mathsf{ok}^* F^{\overline{\kappa}} = \mathsf{ok}^* \Sigma_{\overline{\kappa}} \hat{F} \cong \mathsf{ok}^* \Sigma_{\kappa} (F\mathsf{ok})^* \hat{F} \cong \mathsf{ok}^* \Sigma_{\kappa} \hat{F} \mathsf{ok}^* = F^{?\kappa} \mathsf{ok}^*$. The first equality is by Equation 6, the first isomorphism is by Equation 11, the second isomorphism is by the preceding observation assuming that F preserves pullbacks, and the final equality is by Equation 10. All container functors [1], and hence all polynomial functors, preserve pullbacks. Finally, to verify Equation 11, we require that the distributive law λ for M over F satisfies the following property, which we call *non-introduction of failure*: 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 λ in which the implication holds only from right to left. It ensures that if applying λ does not result in failure, then no failures were present in the data to which it was applied. In an arbitrary category, this property is formulated as requiring the following square (i.e., the unit axiom for λ) to be a pullback:

$$\begin{tikzcd} FA \arrow[r, "F\mathsf{ok}"] \arrow[d, "\mathrm{id}"'] & F(1 + A) \arrow[d, "\lambda_A"] \\ FA \arrow[r, "\mathsf{ok}"] & 1 + FA \end{tikzcd}$$

Every container functor has a distributive law for M satisfying the non-introduction of failure property. We now have

Lemma 2. *If the distributive law λ satisfies non-introduction of failure, then Equation 11 holds.*

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

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

And, putting everything together, we get the correctness of partial refinement:

Theorem 7. *If λ is a distributive law for M over F satisfying the non-introduction of failure property, and if F preserves pullbacks, then $F^{?\kappa}$ has an initial algebra whose carrier is given by Equation 9.*

In fact, Theorem 7 holds in the more general setting of full cartesian Lawvere category whose coproducts satisfy the Beck-Chevalley condition and whose base categories satisfy extensivity [8]. Moreover, Theorem 6 extends to show that these properties are also preserved by the change-of-base construction provided all fibres of the original fibration satisfy extensivity.

7 Conclusions, Applications, Related and Future Work

We have given a clean semantic framework for deriving refinements of inductive types which store computationally relevant information within the indices of refined types. We have also shown how indexed types can be refined further, and how refined types can be derived even when some elements of the original type do not have indices. 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 types into more sophisticated 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: Library implementors need no longer provide a comprehensive collection of data types, but rather methods for defining new data types. Similarly, our results suggest that library implementors need not guess which refinements of data 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` store all index information. For example, a vector of length 3 will store the lengths 3, 2, and 1 of its subvectors. Brady [7] seeks to determine when this information can be generated "on the fly" rather than stored. Our work suggests that the refinement μF^α can be implemented by simply implementing the underlying type μF, since programs requiring indices can reconstruct these as needed. We thus provide a user-controllable tradeoff between space and time efficiency.

Related Work: The work closest to that reported here is McBride's work on ornaments [21]. 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 his notion of an algebraic ornament. However, McBride's ornaments have no counterpart to partial refinement. It will interesting to see to what extent his work can be seen as an implementation of our results.

A line of research allowing the programmer to give refined types to constructors of inductive data types was initiated by Freeman and Pfenning [14] and later developed by Xi [26], Davies [11] and Dunfield [13] for ML-like languages, and by Pfenning [23] and Lovas and Pfenning [20] for LF. This research begins with an existing type system and aims to provide the programmer with a means of expressing richer properties of values that are well-typeable in that type system. It is thus similar to the work reported here, although we instead use a Martin-Löf-style type theory as a programming language and seek techniques for designing invariant-capturing types up-front. Lovas and Pfenning [20] translate LF with refinement into LF with proof irrelevance. Such a translation may extend to richer type theories, thereby allowing our refinement techniques to act as a target for the refinement-type systems presented in the literature, and facilitating precise comparisons of the expressive power of the two approaches.

Refinement types have also been used elsewhere to give more precise types to programs in existing programming languages (but not specifically to inductive types). For example, Denney [12] and Gordon and Fournet [16] use subset types to refine the type systems of ML-like languages. Subset types are also used heavily in the PVS theorem prover [24]. Our results extend the systematic code reuse delivered by generic programming [2,3,5]: in addition to generating new programs we can also generate new types from existing types. This area is being explored in Epigram [9], in which codes for data types can be represented within a predicative intensional system so that programs can generate new data types. It should be possible to implement our refinement process using similar techniques.

Aside from the specific differences between our work and that discussed above, a distinguishing feature of our work 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. It may also lead to new algebraic insights into refinement which complement the logical perspective of previous work.

Finally, we are interested in a number of extensions to the work reported here. Many readers will wonder about the possibility of a more general monadic refinement using, for example, Kleisli categories. We are working on this, but due to space limitations have chosen to concentrate in this paper on partial refinement, which is already sufficient to show that refinement is applicable to sophisticated programming problems. In addition, many more indexed inductive data types exist than can be produced by the refinement process described in this paper. We leave it to future work to discover to what extent this developing world of dependently typed data structures can be organised and characterised by processes like refinement and its extensions.

References

1. M. Abbott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, 2005.
2. T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *Proc., Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 209–257. Springer, 2007.
3. R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors. *Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer, 2007.
4. M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer, 1983.
5. 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.
6. R. S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.
7. E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Proc., TYPES*, volume 3085 of *LNCS*, pages 115–129. Springer, 2004.
8. A. Carboni, S. Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84:145–158, 1993.
9. J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proc., ICFP*, 2010. To appear.
10. T. H. Cormen, C. E. Leiserson, R.L̃. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.
11. R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. Available as Technical Report CMU-CS-05-110.
12. E. Denney. Refinement types for specification. In *Proc., PROCOMET*, pages 148–166. Chapman and Hall, 1998.
13. J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. Available as Technical Report CMU-CS-07-129.
14. T. Freeman and F. Pfenning. Refinement types for ML. In *Proc., Symposium on Language Design and Implementation*, pages 268–277, June 1991.
15. N. Ghani, P. Johann, and C. Fumex. Fibrational induction rules for initial algebras. In *Proc., CSL*, pages 336–350, 2010.
16. A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, Microsoft Research, October 2009.
17. C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145(2):107–152, 1998.
18. B. Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107:169–207, 1993.
19. B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, 1999.
20. W. Lovas and F. Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Logical Methods in Computer Science*, 2010. To appear.
21. C. McBride. Ornamental algebras, algebraic ornaments. Unpublished note, 2010.
22. N. P. Mendler. Predicative type universes and primitive recursion. In *Proc., LICS*, pages 173–184. IEEE Computer Society, 1991.
23. F. Pfenning. Refinement types for logical frameworks. In *Proc., Types for Proofs and Programs*, pages 285–299, 1993.
24. J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
25. The Agda Team, 2010. http://wiki.portal.chalmers.se/agda.
26. H. Xi. Dependently typed data structures. Revision after WAAAPL ’99, 2000.