Under consideration for publication in Math. Struct. in Comp. Science

An Algebraic Foundation and Implementation of Induction Recursion and Indexed Induction Recursion

Neil Ghani, Peter Hancock

Received October 2001

Induction recursion offers the possibility of a clean, simple and yet powerful meta-language for the type system of a dependently typed programming language. At its crux, induction recursion allows us to defining a universe, that is a set U of *codes* and a decoding function $T : U \to D$ which assigns to every code $u : U$, a value Tu of some type D. The key feature of induction recursion is that the codes in U are built up inductively at the same time as the recursive definition of their decoding function T.

Despite its potential, induction recursion has not become as widely understood, or used, as it should be. We believe this is in part because: i) there is still scope for analysing the theoretical foundations of induction recursion; and ii) a presentation of induction recursion for the wider functional programming community still needs to be developed. The aim of this paper is to tackle exactly these two issues. That is, we aim to i) develop an algebraic foundation for induction recursion to complement the original type-theoretic one; and ii) use this foundation to construct a clean implementation of induction recursion which thereby broadens its accessibility to functional programmers. Theory and practice, hand in hand, as it should be!

1. Introduction

Recursion is one of the most fundamental concepts in computation. Its importance lies in the ability it gives us to define computational agents in terms of themselves - these could be recursive programs, recursive data types, recursive algorithms or any of a myriad of other structures. The first formal analysis of recursion go back a century or more, to the birth of the theory of general recursive functions, fixed points, and induction. It is virtually impossible to overestimate how recursion has contributed to our ability to compute, and to understand the process of computation.

Is it possible that there is anything fundamental left to say about recursion? We believe there is and so, in this article, we want to focus on just one strand, namely *induction-recursion.* When defining a function $f : A \to B$ recursively, A is usually fixed in advance. But what if it is not? What if, as we build up the function f recursively, we also build up "just-in-time" the type A inductively? Induction recursion concerns itself with the study of functions defined in this way. This name is due to Peter Dybjer, who together with Anton Setzer wrote a number of papers developing the subject. The idea was first introduced formally by Dybjer (Dybjer(2000)) and then Dybjer and Setzer proposed a closed form for induction recursion in (Dybjer and Setzer(2003)). However, the origin of induction recursion can be traced back further to the notion of universe introduced by Martin-Löf in the early 70's (Martin-Löf(1973)).

It is our opinion that dependently typed programmers have much to gain from an understanding of induction recursion because induction recursion offers the possibility of a clean, simple and yet powerful meta-language for the type system of a dependently typed programming language. To see this consider the evolution of the theory of data types within programming languages.

- **Inductive types:** At the simplest level, *inductive types* arise as the least fixed points of operators $F : \mathsf{Set} \to \mathsf{Set}$ [†]. Natural numbers, lists storing data of a given type, binary trees *etc.* fit into this framework. For example, a data type $Tree$ of binary trees (storing no data at the leaves) is the least type satisfying

$$Tree = 1 + Tree \times Tree$$

and hence arises as the least fixed point of the operator $F : \mathsf{Set} \to \mathsf{Set}$ defined by

$$FX = 1 + X \times X$$

Of course, not all operators $F : \mathsf{Set} \to \mathsf{Set}$ have least fixed points which behave well and so grammars for generating operators with well behaved least fixed points have been developed. Amongst these grammars are the polynomial operators, strictly positive operators and containers (Abott et al.(2005)Abott, Altenkirch, and Ghani).

- **Inductive Families:** At the next level of sophistication, one considers *inductive families* which do not just define *one* type inductively, but rather a U-indexed family $T : U \to \mathsf{Set}$ of types simultaneously. Inductive families arise as least fixed points of operators $F : (U \to \mathsf{Set}) \to U \to \mathsf{Set}$. Crucially, U here is a fixed set defined independently and in advance of the inductive family. For example, the terms of the untyped λ-calculus upto α-equivalence form a $\mathbb{N}$-indexed set $\mathsf{Lam} : \mathbb{N} \to \mathsf{Set}$ which is the least $\mathbb{N}$-indexed set satisfying the equation

$$\mathsf{Lam}\ n = \mathsf{Fin}\ n + (\mathsf{Lam}\ n) \times (\mathsf{Lam}\ n) + \mathsf{Lam}\ (n+1)$$

where $\mathsf{Fin}\ n$ is a type with n-elements. Thus Lam arises as as the least fixed point of the operator $F : (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$ defined by

$$F\ X\ n = \mathsf{Fin}\ n\ +\ (X\ n) \times (X\ n)\ +\ X\ (n+1)$$

Grammars for defining operators which give rise to well behaved inductive families

[†] In this paper we are attempting to address dependently typed programmers and hence have stayed away from the categorical lexicon which some dependently typed programmers may be unfamiliar with. However, for those who it may be helpful, we will reinterpret our constructions in the categorical lexicon at the end of this paper

include indexed containers (Morris and Altenkirch(2009)) and dependent polynomials (Gambino and Hyland(2004)).

• **Induction Recursion:** We believe that the next step in this hierarchy of definitional formats is induction recursion which generalises inductive families by allowing a set U to be defined simultaneously with a function $T : U \to D$ where D is usually a large type such as Set. This means that inductive recursive types arise as least fixed points of operators $F : \mathsf{Fam}(D) \to \mathsf{Fam}(D)$ where the elements of $\mathsf{Fam}(D)$ are families (U, T) where U is a set (called the indexing set of the family) an $T : U \to D$ is a function (called the decoding function of the family).

To give an example of an inductive recursive definition, first recall that a universe is a pair (U, T) where we think of the indexing set U as consisting of a the names or *codes* for types and the decoding function $T : U \to \mathsf{Set}$ assigning to each code $u : U$ the type Tu of elements of the type named by the code u. An example of an inductive recursive definition is that of a universe containing a code for the natural numbers and closed under Σ-types. Such a universe is the smallest family of sets (U, T) satisfying the equations

$$\begin{array}{lcl} U & = & 1 + \Sigma u{:}U.\ Tu \to U \\ T(\mathsf{inl}\ *) & = & \mathbb{N} \\ T(\mathsf{inr}\ (u, f)) & = & \Sigma x{:}Tu.\ T(fx) \end{array}$$

To understand the above definition, note that a Σ-type ΣAB consists of a type A and a function B mapping each element of A to a type. Thus the name of a Σ-type in a universe (U, T) will consist of a name in U for the type A, *i.e.* an element of $u : U$, and a function assigning to every element of the type denoted by u, *i.e.* every element of Tu, the name of a type, *i.e.* an element of U. An element of the type denoted by a name (u, f) consists of an element of the type denoted by u, *i.e.* an element of Tu, and an element of the type named by fu, *i.e.* an element of $T(fu)$).

The crucial point about the above example is that U depends upon T and so U cannot be defined in advance of T. Thus, this universe is inductive recursive but not an inductive family.

As with inductive types and inductive families, induction recursion contains: i) a represention data types as least fixed points of operators between families; and ii) a grammar for defining such operators. Elements of the grammar are called IR-codes, while the function assigning to each IR-code the operator it represents is called the decoding function. Induction recursion covers all of the data types mentioned above but comes into its own when we wish to defined universes closed under dependently typed operations (as in the example above). This is because dependently typed operations use elements of types within types and so universes closed under such operations must have their codes U (*i.e.* the names of types in the universe) and their decoding functions T (*i.e.* the elements of a type within the universe) defined simultaneously. Such examples cannot be easily represented within any of the theories of data types mentioned above. Further examples of inductive recursive definitions will be given in the body of the paper. Indexed induction recursion (Dybjer and Setzer(2006)) goes beyond induction recursion in that one can define a set of codes and a decoding function with the possibility that different codes can decode to elements of different types. Thus, within indexed induction recursion, we can have some codes which decode to, say, types and some codes which decode to, say, operators on types.

Despite the fundamental conceptual insight of Dybjer and Setzers type-theoretic foundation, induction recursion has yet to become as widely used as its potential suggests it should. We believe this is because both the theoretic foundations and pragmatic aspects of induction recursion need further development. This paper proposes to address both these issues by i) developing an algebraic foundation for induction recursion to complement the type-theoretical foundation of Dybjer and Setzer; and ii) following in the footsteps of the algebra of programming school and reflecting this algebraic structure into an implementation. This methodology has proved to be very fruitful in the past probably because both the algebra of programming and functional programming are based upon understanding computational phenomena using high levels of abstraction, and then reflected this understanding into structured programming idioms. More concretely: i) clean foundations lead to clean code which is therefore easier to both understand and experiment with; and ii) implementing these foundations also guarantees their partial correctness via type checking. We may summarise this methodology by paraphrasing a famous quote "Theorists have so far only interpreted code in various ways – the point, however, is to structure it!".

In more detail, after recapping Dybjer and Setzer's theory of induction recursion in section 2, our contributions are as follows:

- In section 3 we introduce a number of algebraic ideas which we then use to structure our implementation of induction recursion. In more detail, i) we investigate families and the algebraic structure they support; ii) separate positive and negative uses of families within an IR-code; iii) introduce large families as the essential structure from which IR-codes are derived; and iv) introduce *localisation* as the key structure from which the decoding function for IR-codes are derived.

- In section 4 we show how induction recursion can be reformulated using containers. That is, we replace the notion of large families and localisation from section 3 with a different notion of large family based upon containers from which the codes of induction recursion can be derived and a new notion of localisation from which the associated decoding function can be derived. This allows us to compactify the definition of IR-codes so as to expose the connection between contianers and induction recursion in preparation for our treatment of indexed induction recursion.

- In section 5 we show that the relationship between containers and induction recursion developed in section 4 can be replicated in the indexed world. That is, we use indexed containers to explain, structure and implement indexed induction recursion.

Concretely, this means the definition of a new notion of large family based upon indexed containers from which the codes of indexed induction recursion can be derived and a new notion of localisation from which the associated decoding function can be derived.

• Despite the temptation to do otherwise, we have taken a resolutely non-categorical approach in this paper so as to ensure our work is as accessible as possible. However, this does loose something, and so in section 6 we recast some our ideas in a more categorical way. However, none of the rest of the paper depends upon this section and the reader can safely skip it should they not have the prerequisites.

• We conclude in section 7 and offer directions for future research.

In terms of programming, we use the dependently typed programming language Agda to present our implementation of induction recursion and indexed induction recursion. Those not familiar with this dependently typed programming language can either find more details at (Bove et al.(2009)Bove, Dybjer, and Norell) or can infer the meaning of the Agda code from general functional programming knowledge. The only Agda2 specific notion is that of implicit parameters to functions which are written "$\{x : A\} \rightarrow$ ..." and which can be thought of as the declaration of an input which can be inferred from its context when used and hence need not be given. Our code can be found at `http://www.cis.strath.ac.uk/~ng`. Its a testament to the progress in dependently typed programming that we now have a language which is abstract enough to closely mirror the conceptual foundations of induction recursion with relatively minor overhead.

2. Induction Recursion in A Nutshell

In this section, we set the scene for our work by giving the essence of Dybjer and Setzer's system of codes for induction recursion. The codes themselves have a very elegant and compact definition as follows:

Definition 1 (Dybjer and Setzer's IR-Codes). Let D be a type. The large type $\mathsf{IR}\ D$ of IR-codes has the following constructors

$$\frac{d : D}{\iota\, d : \mathsf{IR}\ D}$$

$$\frac{A : \mathsf{Set} \quad f : A \rightarrow \mathsf{IR}\ D}{\sigma_A f : \mathsf{IR}\ D}$$

$$\frac{A : \mathsf{Set} \quad F : (A \rightarrow D) \rightarrow \mathsf{IR}\ D}{\delta_A F : \mathsf{IR}\ D}$$

Understanding IR codes, and thus each of the IR-constructors, is best done by understanding the semantics of each IR-code. Dybjer and Setzer gave this semantics in the form of a decoding function which associates to each IR code a mapping of families to families. We first define families before defining Dyber and Setzer's decoding function.

Definition 2 (Families). Let D be a type. Then $\mathsf{Fam}(D)$ is the large type whose elements are pairs (U, T) where U is a set and $T : U \to D$. We call U the index of the family and T the decoding function.

A degenerate case is worth pointing out as we shall use it later

Example 3 ($\mathsf{Fam\ One}$)**.** A degenerate case occurs when $D = \mathsf{One}$, the unit type. In this case, $\mathsf{Fam\ One}$ simply consists of sets, *i.e.* $\mathsf{Fam\ One} = \mathsf{Set}$. This special case will allow us to study operators $F : \mathsf{Set} \to \mathsf{Set}$ within induction recursion. More precisely, it will allow us to see that IROne codes represent exactly those operators on sets which are containers.

The next part of Dybjer and Setzer's work is to define a decoding function which assigns to every type D and IR code $c \in \mathsf{IR}\ D$ a mapping $[\![c]\!] : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$ of families to families. To do this, Dybjer and Setzer define the mapping $[\![c]\!]$ in two parts: i) a function $[\![c]\!]_0 : \mathsf{Fam}\ D \to \mathsf{Set}$ with the intent that $[\![c]\!]_0(U, T)$ is the index set of the family $[\![c]\!](U, T)$; and ii) a function $[\![c]\!]_1$ which assigns to each family (U, T) a function $[\![c]\!]_0(U, T) \to D$ with the intent that $[\![c]\!]_1(U, T)$ is the decoding function of $[\![c]\!](U, T)$. These functions are defined as follows:

Definition 4 (Dybjer and Setzer's Decoding Function). Let D be a type and $c \in \mathsf{IR}\ D$. Define the mapping $[\![c]\!] : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$ as follows:

- When $c = \iota d$, the functor $[\![c]\!]$ denotes the constant functor which returns a family with one index which decodes to the element d. Hence $[\![\iota d]\!]$ is defined by

$$\begin{array}{rcl} [\![\iota\ d]\!]_0\ (U, T) & = & 1 \\ [\![\iota\ d]\!]_1\ (U, T)\ \_ & = & d \end{array}$$

- When $c = \sigma_A f$, for each element a of the set A, we have a code fa and hence an operator $[\![fa]\!] : \mathsf{Fam}(D) \to \mathsf{Fam}(D)$. The intended meaning of the code $\sigma_A f$ is simply the pointwise sum of all of the operators $[\![fa]\!]$ for $a \in A$. This intended meaning is formalised by defining

$$\begin{array}{rcl} [\![\sigma_A f]\!]_0\ (U, T) & = & \Sigma a : A. [\![fa]\!]_0\ (U, T) \\ [\![\sigma_A f]\!]_1\ (U, T)\ (a, i) & = & [\![fa]\!]_1\ (U, T)\ i \end{array}$$

- When $c = \delta_A F$, for each $X : A \to D$, we have a code FX and hence an operator $[\![FX]\!] : \mathsf{Fam}D \to \mathsf{Fam}D$. If we consider a function $X : A \to D$ as an A-tuple of elements of D, then we can consider $[\![FX]\!]$ as a code parameterised by such a collection. The intended meaning of $\delta_A F$ is similar to the pointwise sum of all the operators $[\![FX]\!]$. However, rather than summing over the large collection of all functions $A \to D$, the action of $[\![\delta_A F]\!]$ on a family (U, T) restricts to only those elements of D which are in the image of T, *i.e.* only those elements of D which are definable within the family (U, T). This intended meaning is formalised by defining

$$\begin{array}{rcl} [\![\delta_A F]\!]_0\ (U, T) & = & \Sigma g : A \to U.\ \ [\![F(T \circ g)]\!]_0\ (U, T) \\ [\![\delta_A F]\!]_1\ (U, T)\ (g, i) & = & [\![F(T \circ g)]\!]_1\ (U, T)\ i \end{array}$$

We note that in the definition of decoding notice that various Σ-types exist but they are all indexed by sets and hence produce sets. For example, the definition of $[\![\sigma_A f]\!]$ uses a Σ-type over elements of the set A, while the definition of $[\![\delta_A f]\!](U,T)$ uses a Σ-type indexed by elements of $A \to U$. Since A and U are sets, such elements also form a set.

In fact, Dybjer and Setzer proved that $\mathsf{Fam}(D)$ is a category and that if c is an IR code, then $[\![c]\!]$ not only maps families to families, but also morphisms of $\mathsf{Fam}D$ to morphisms of $\mathsf{Fam}D$. That is, they showed that $[\![c]\!] : \mathsf{Fam}D \to \mathsf{Fam}D$ is a functor. However, the focus of this paper is not the categorical foundations of IR and hence we do not pursue such issues until section 6. Rather, we content ourselves with continuing with the goal of producing a clean implementation of induction recursion.

The final part of the work of Dybjer and Setzer is to state the principle of definition by induction recursion. This principle is the assertion that for every code c, the mapping $[\![c]\!]$ has a least fixed point. As a result, induction recursion is a principle asserting the existence of various set-indexed families.

Definition 5 (IR Datatypes). A family $T : U \to D$ is inductive recursive iff there is an IR-code c such that (U,T) is the least fixed point of $[\![c]\!]$.

So the main question is ... what does the above mean? It looks like fairly technical type theory and many researchers have tried, and found it hard, to understand IR-codes and their semantics in the form of their decoding function. This is clearly a problem since induction recursion has enough potential that it deserves study from a variety of researchers with varying perspectives and backgrounds. We hope our algebraic perspective on induction recursion, together with its implementation in Agda, will help to rectify this situation from both the theoretical and practical perspectives. But before we delve into our own results, we present some examples to show both the power, and the inscrutability, of induction recursion.

2.1. *Examples of Induction Recursion:*

If $D = \mathsf{One}$, then $\mathsf{Fam}\ D$ is essentially the category Set of sets. Hence functors on Set can be discussed within the framework of induction recursion.

Example 6 (Natural numbers). The functor mapping X to $X+1$ is the decoding of the IR-code

$$\iota * + \delta_1(\lambda x.\, \iota *) : \mathsf{IR\ One}$$

We can embellish or transform this code to one that can be used not only to define the set of natural numbers, but can also be used to define the set-valued function $\mathsf{Fin} : \mathbb{N} \to \mathsf{Set}$ that assigns to each natural number the enumerated type $\mathsf{Fin}(n) = \{0, \ldots, n-1\}$. The code for this is:

$$\iota\, 0 + \delta_1\ (\lambda X.\, \iota\, (X+1)) : \mathsf{IR\ Set}$$

In the last example, we coded-up the inductively defined datatype $\mathbb{N}$ with an IR-code, and showed how to embellish it to define directly by recursion a useful set-valued function. This is known as *large elimination*, or *large-valued recursion*, and was discussed in (Smith(1989)). One of the main reasons (stated in (Martin-Löf(1973))) for the invention of universe types in dependent type theory was to make it possible to define families of datatypes defined by structural recursion on their indices, so obtaining the effect of large-valued recursion.

A further use of induction recursion is to define closure operations over families of sets. The following example shows how this can be done in a variety of different ways.

Example 7 (Closure under Σ). Suppose $(S, E) : \mathsf{Fam\,Set}$. We wish to define a universe extending this given family of sets, that contains the singleton set One, and is closed under the quantifier Σ. This universe is the initial algebra for the following endofunctor on $\mathsf{Fam\,Set}$.

$$(U, T) \mapsto \ (S, E) \ + \ (\mathsf{One}, \lambda - . \,\mathsf{One}) \ + \ ([\![U, T]\!]U \ , \ \lambda(u, f).\, \Sigma\,(T\,u)(T \cdot f))$$

The code for this functor is:

$$\sigma\iota\,(S, E) \ + \ \iota\,\mathsf{One} \ + \ \delta_{\mathsf{One}}\,(\lambda U.\,\delta_U\,(\lambda T.\,\iota(\Sigma\,U\,T)))$$

The following example shows how induction recursion can be used to used to define functions whose values are themselves codes for inductive recursive functors, and so shows how induction recursion can be used for its own metaprogramming.

Example 8 (Church numerals). Consider the following operators

$$\begin{array}{lcl}
(U, T) & \mapsto & (\mathsf{One}, \_ \mapsto \mathsf{One}) \\
(U, T) & \mapsto & (U, T) \\
(U, T) & \mapsto & (U, T) * (U, T) \\
(U, T) & \mapsto & (U, T) * (U, T) * (U, T) \\
\ldots & &
\end{array}$$

where $*$ is a binary operation on families sending (U, T) and (U', T') to the family with index set $\Sigma u\!:\!U.\,T u \to U'$ and with decoding function sending (u, f) to $\Sigma x\!:\!T u.\, T(f x)$. These operators have the following codes

$$\begin{array}{l}
\iota\ \mathsf{One} \\
\delta_{\ \mathsf{One}}\ (\lambda X.\, \iota\ (\Sigma_{\ \mathsf{One}}\, X)) \\
\delta_{\ \mathsf{One}}\ (\lambda X.\, \delta_{\ (\Sigma_{\ \mathsf{One}}\, X)}\ (\lambda Y.\, \iota\ (\Sigma_{\ (\Sigma_{\ \mathsf{One}}\, X)}\, Y))) \\
\delta_{\ \mathsf{One}}\ (\lambda X.\, \delta_{\ \Sigma_{\ \mathsf{One}}\, X}\ (\lambda Y.\, \delta_{\ (\Sigma_{\ \Sigma_{\ \mathsf{One}}\, X}\, Y)}\ (\lambda Z.\, \iota\ (\Sigma_{\ \Sigma_{\ (\Sigma_{\ \mathsf{One}}\, X)}\, Y}\, Z)))) \\
\ldots
\end{array}$$

The n^{th} term sequence can be obtained as $n|_{\mathsf{One}}$ where $n|_{\_} : \mathsf{Set} \to \mathsf{IR(Set)}$ is the n^{th} term of the following sequence of polymorphic codes.

$$\begin{array}{rcl}
0|_A & = & \iota\ A \\
(n+1)|_A & = & \delta_A\,(\lambda X : A \to \mathsf{Set}.\, n|_{(\Sigma\,A\,X)})
\end{array}$$

We may then, as in the first example above, embellish the code : $\mathsf{IR}(\mathsf{One})$ that defines the datatype $\mathbb{N}$ so as to obtain

$$\begin{array}{ll} & \iota(\lambda A.\, \iota\, A) \\ + & \delta_{\mathsf{One}}\, (\lambda F : \mathsf{Set} \to \mathsf{IR}(\mathsf{Set}). \\ & \iota(\lambda A : \mathsf{Set}.\, \delta_A\, (\lambda X.\, F\, (\Sigma\, A\, X)))) \\ : & \mathsf{IR}\, (\mathsf{Set} \to \mathsf{IR}\, \mathsf{Set}) \end{array}$$

by means of which the entire sequence of polymorphic codes can be defined in one go. Note in this example one does not take D to be the usual choices of One or Set.

3. Families, Large Families, Mixed Variance and Localisation

Families and their Structure: The presentation of the decoding function given above takes as input a code c and a family (U, T) and first computes the index set of the output family and then computes the decoding function of the output family. This is essentially a two stage process which treats a family as an index set together with a decoding function. There is an alternative - namely to work directly at the level of families by treating families as atomic entities. Doing this will raise the level of abstraction at which we understand induction recursion and, as found within the algebra of programming school, creates greater conceptual simplicity and thereby increases the tractability of the theory.

In this section we develop the properties of families we need and then use them to give an algebraic foundation and implementation of of induction recursion based upon this structure. As we shall do throughout the rest of the paper, we present all of our constructions in Agda so dependently typed programmers can experiment with them. Families were defined in Definition 2 and can be implemented in Agda by using the Agda type Set to represent sets and the Agda type Set_1 to represent types. This may be done as follows:

$$\begin{array}{l} \mathsf{Fam} : \mathsf{Set}_1 \to \mathsf{Set}_1 \\ \mathsf{Fam} D = \Sigma\ \ \mathsf{Set}\ \ (\lambda A \to (A \to D)) \\[4pt] \mathsf{Fam}_1 : \{D\ \ D' : \mathsf{Set}_1\} \to (D \to D') \to \mathsf{Fam} D \to \mathsf{Fam} D' \\ \mathsf{Fam}_1\ \ g\ \ (A\ \ ,\ \ \ f) = (A\ \ ,\ \ g \circ f) \\[4pt] \pi_0 : \{D : \mathsf{Set}_1\} \to \mathsf{Fam} D \to \mathsf{Set} \\ \pi_0\ \ (U, T) = U \\[4pt] \pi_1 : \{D : \mathsf{Set}_1\} \to (P : \mathsf{Fam} D) \to \pi_0 P \to D \\ \pi_1\ \ (U\ \ ,\ \ T) = T \end{array}$$

The above implementation of families also defines the projections π_0 and π_1 which take a family as input and extract the index set and the decoding function respectively. In addition to implementing the large type $\mathsf{Fam} D$, we have also implemented a function Fam_1 which lifts functions between types to functions between families of types. Most of the operators on types we use in this paper posses such liftings of functions and so this pattern will be repeated on several occasions.

One other key concept we need is that of a monad which, thanks to the work of first Eugenio Moggi and then Phillip Wadler, has become a standard technqiue used by functional programmers to model effectful computation. In keeping with the implementation focussed nature of this paper, we won't delve into the categorical roots of monads - an interested reader can consult (Mac Lane(1998)). Instead, we simply implement the the monadic structure of Fam that we shall need. To understand this structure, notice that a family of Ds is like a subset of D or, more accurately, a multiset of Ds. Now, just as subsets (and multisets) support a singleton operation and a union operation so do families. The first operation is called the *unit* of Fam while the second operation is called the *multiplication* of Fam and they can be implemented as follows:

$$\begin{aligned}
&\eta : \{D : \mathsf{Set}_1\} \to D \to \mathsf{Fam} D\\
&\eta \;\; d = (\top \;\;,\;\; \mathsf{const}\; d)
\end{aligned}$$

$$\begin{aligned}
&\mu : \{D : \mathsf{Set}_1\} \to \mathsf{Fam}(\mathsf{Fam} D) \to \mathsf{Fam} D\\
&\mu \;\; \{D\} \;\; (A \;\;,\;\; B) = (\Sigma \;\; A \;\; (\pi_0 \circ B) \;\;,\;\; g)\\
&\qquad\qquad\qquad\qquad \mathsf{where} \;\; g : \Sigma \quad A \quad (\pi_0 \circ B) \to D\\
&\qquad\qquad\qquad\qquad\qquad\quad g \;\; (a \;\;,\;\; k) = \pi_1 \;\; (Ba) \;\; k
\end{aligned}$$

The above code uses several predefined Agda primitives: $\top$ is the Agda unit type and const is the Agda function that takes a value as input and returns the constant function with that value.

Large Families: We have seen that families are important in induction recursion as IR-codes produce functors mapping families to families. Further, the input to the σ-constructor is in fact a family of IR-codes. That is σ actually has type Fam IR $\to$ IR. But what about the δ-constructor? If we think about σ-constructor as taking as input a code parameterised over *elements of a set* A, then (when $A = 1$ and $D =$ Set) the δ-constructor takes as input a code parameterised over *an arbitrary set.* At this point the reader should think about the difference between a function consisting of a value parameterised by elements of a *specific* set, and a function consisting of a value parameterised by *sets themselves.* Similarly, one can see σ as building codes from A-indexed codes, for a specific set A, while δ-builds codes from Set-indexed codes (again when $D =$ Set). To model this alternative form of parameterisation, we introduce the notion of a large family, so called because such a large family existentially quantifies over Set unlike the usual notion of family which existentially quantifies over a specific set. In Agda, one may define large families as follows:

$$\begin{aligned}
&\mathsf{LFam} : \mathsf{Set}_1 \to \mathsf{Set}_1 \to \mathsf{Set}_1\\
&\mathsf{LFam} \;\; I \;\; O \;\; = \;\; \Sigma \;\; \mathsf{Set} \;\; (\lambda A \to (A \to I) \to O)
\end{aligned}$$

Notice the similarity with families we have alluded to. While families are collections indexed by elements of a type indexed by a set, the above definition of the large family LFam I O is a collection of Os indexed by (when $A = 1$) elements of I where I is not of type Set, but rather of type Set_1. Of course there are other signifcant differences between

LFam and Fam and we turn to those differences now.

Mixed Variant IR: After this discussion of families, we turn our attention to variance. The reader may have wondered why Fam is a function of one input while LFam is a function of two inputs. That is, why did we not define

$$\begin{aligned}&\mathsf{BadLFam} : \mathsf{Set}_1 \to \mathsf{Set}_1\\ &\mathsf{BadLFam}\ \ D = \ \ \Sigma\ \ \mathsf{Set}\ \ (\lambda A \to (A \to D) \to D)\end{aligned}$$

The reason is simple — while the definition of Fam supports a lifting of functions to families as given by the function Fam_1, the definition of BadLFam does not. The problem is that in BadLFam, the type D occurs in both a negative position, or *contravariantly* and in a positive position, or *covariantly.* The defintion of LFam separates these contravariant and covariant positions of D into two separate arguments. As a result, functions between types can be lifted to functions between large families over those types in the covariant argument. This leads to the following definition of LFam_1

$$\begin{aligned}&\mathsf{LFam}_1 : \{I\ \ D\ \ D' : \mathsf{Set}_1\} \to (D \to D') \to \mathsf{LFam}\ \ I\ \ D \to \mathsf{LFam}\ \ I\ \ D'\\ &\mathsf{LFam}_1\ \ g\ \ (A\ \ ,\ \ \ F) = (A\ \ ,\ \ g \circ F)\end{aligned}$$

The natural next question is then to wonder about the variance within IR-codes themselves. For example, the reader may have asked whether there is a function that lifts a function $D \to D'$ to a function $\mathsf{IR}\ D \to \mathsf{IR}\ D'$. The answer is clearly no - since the δ-constructor has covariant and contrvariant parts, clearly $\mathsf{IR}\ D$ will be both covariant and contravariant in D. However, just as with large families, we can tease out the covariance and contravariance if we notice that all the covariance comes from the ι constructor, while all the contravariance comes from the δ-constructor. The constructor σ is variance neutral since I and O don't play a direct role in the constructor σ.

Putting together the above ideas concerning famlilies, large families and variance, we arrive at our first reformulation of IR codes as the following Agda definition

$$\begin{aligned}&\mathsf{data}\ \ \mathsf{IR}(I\ \ O : \mathsf{Set}_1) : \mathsf{Set}_1\ \ \mathsf{where}\\ &\quad \iota : O \to \mathsf{IR}\ \ I\ \ O\\ &\quad \sigma : \mathsf{Fam}(\mathsf{IR}\ \ I\ \ O) \to \mathsf{IR}\ \ I\ \ O\\ &\quad \delta : \mathsf{LFam}\ \ I\ \ (\mathsf{IR}\ \ I\ \ O) \to \mathsf{IR}\ \ I\ \ O\end{aligned}$$

The above presentation of IR codes makes it clear that the type $\mathsf{IR}\ \ I\ \ O$ can be thought of as a simple data type, namely the least fixed point of the operator which maps a family X to the family $O + \mathsf{Fam}\ \ X + \mathsf{LFam}\ \ I\ \ X$. Here, the action of $+$ maps two families to the family i) whose index set is the disjoint union of the index sets of the input families; and ii) whose decoding function is given by cotupling. This is the most fundamental pay off for mixed variant IR - by separating out the variance in $\mathsf{IR}\ \ I\ \ O$'s constructors, we get a presentation of IR-codes in terms of the standard theory of data types which opens the way to using the standard techniques from the algebra of programming to structure and reason about implementations of induction recursion. We can put this observation to use immediately by using it to give a structured implementation of the decoding for IR-codes.

Decoding and Localisation: What is left to do is give our implementation of the decoding function for our presentation of IR-codes using Fam and LFam given above. As our mixed variant codes IR I O are parameterised by a negatively occurring I and a positively occurring O, the reader will not be surprised to discover that each mixed variant IR-code decodes to a mapping sending families in Fam I to families in Fam O. That is, for our mixed variant IR-codes, we shall define the decoding function to have type

$$[\![-]\!] : \{I \;\; O : \mathsf{Set}_1\} \rightarrow \mathsf{IR}\;\; I \;\; O \rightarrow \mathsf{Fam}\;\; I \rightarrow \mathsf{Fam}\;\; O$$

Further, because we can now see IR I O through the prism of the algebra of programming, we can structure the decoding function for IR-codes by using the associated *fold* operator for IR I O. This operator is defined as follows:

$\mathsf{fold} : \{I \;\; O \;\; X : \mathsf{Set}_1\} \rightarrow$
$\qquad\qquad (O \rightarrow X) \rightarrow (\mathsf{Fam}\, X \rightarrow X) \rightarrow (\mathsf{LFam}\;\; I \;\; X \rightarrow X) \rightarrow \mathsf{IR}\;\; I \;\; O \rightarrow X$
$\mathsf{fold}\;\; i \;\; s \;\; d \;\; (\iota \;\; o) = i \;\; o$
$\mathsf{fold}\;\; i \;\; s \;\; d \;\; (\sigma \;\; f) = s \;\; (\mathsf{Fam}_1 \;\; (fold \;\; i \;\; s \;\; d) \;\; f)$
$\mathsf{fold}\;\; i \;\; s \;\; d \;\; (\delta \;\; F) = d \;\; (\mathsf{LFam}_1 \;\; (fold \;\; i \;\; s \;\; d) \;\; F)$

Note how the definition of fold requires both Fam and $LFam$ to lift functions between types to functions between the associated families. A naive way to use fold would be to define $[\![c]\!]$ as a fold by taking the type X in the definition of fold given above to be Fam $I \rightarrow$ Fam O. Indeed, that is what we did first! However, it is easier to fix a family m : Fam I and then define $[\![-]\!]m$ as a fold as, in this case, X can be taken to be the simpler type Fam O. To do this, we will have to supply the three parameters for fold, the so-called replacement functions for the constructors ι, σ and δ. We do so as follows:

- The first parameter of the fold turns out simply to be the unit η of the families monad. This is because $[\![\iota\; -]\!]m$ takes a $d{:}D$ as input and returns the family ηd, *i.e.* the family with one index which decodes to d.

- To understand the second parameter of the fold function which will compute $[\![c]\!]m$, note that if c is of the form $\sigma_A f$, then the fold will act in a structurally recursive fashion. That is, for each $a : A$, it will compute $[\![fa]\!]m$: Fam O. Thus, overall we have an A-indexed family of elements of Fam O, ie an element of Fam(Fam O). At this point, the penny drops and we notice that hidden inside Dybjer and Setzer's definition of the decoding function for codes of the form $\sigma_A f$ is the application the multiplication μ of the families monad to obtain the desired result of type Fam O. That is, the replacement function for σ is nothing other than μ!

- To understand the third parameter, notice that both $[\![\sigma_A \;\; f]\!](U,T)$ and $[\![\delta_A \;\; F]\!](U,T)$ both are sums, the first indexed by A and the second indexed by $A \rightarrow U$. Indeed, calcuations show that

$$[\![\delta_A\ F]\!](U,T) = [\![\sigma_{A\to U}(\lambda g : A \to U.\ \ F(T\circ g))]\!](U,T)$$

To formalise this observation, lets define the following Agda function which takes a family $T : U \to I$ and a large family $F : (A \to I) \to O$ as input and restricts the large family so that it can only range over elements of type I which have codes in (U, T).

$$\begin{array}{l} \rho : \{I\ \ O : \mathsf{Set}_1\} \to \mathsf{Fam}\ \ I \to \mathsf{LFam}\ \ I\ \ O \to \mathsf{Fam}\ \ O \\ \rho\ \ (U\ \ ,\ \ T)\ \ (A\ \ ,\ \ F) = (A \to U\ \ ,\ \ \lambda g \to F(T\circ g)) \end{array}$$

We call such a function a *localisation* for LFam since, given a fixed family (U, T), the function $\rho(U, T)$ turns a large family into a family by localising the large family to consider only those elements of I which have codes in U. Note how this formalises the intuition we gave for Dyber and Setzer's decoding function for codes of the form $\delta_A F$. Using ρ, we have

$$[\![\delta_A\ F]\!](U,T) = [\![\sigma(\rho\ \ (U,T)\ \ (A,F))]\!](U,T)$$

and so our third replacement function is simply $\mu\circ\rho m$.

Thus, our implementation of the decoding function for our mixed variant IR-codes is

$$[\![c]\!]\ \ m = \mathsf{fold}\ \ \eta\ \ \mu\ \ (\mu\circ\rho m)\ \ c$$

Notice several things: as promised, working directly with the families monad has both allowed us to define the decoding function of an IR-code uniformly over families rather than by giving first the index set and then the decoding function of the output. Further, we see the monadic structure of the families monad as playing a crucial role in the semantics of each of the three IR-constructors. Thirdly, it was only because we separated out the variance and defined IR I O rather than IR D, that we could implement the decoding function via a fold.

Given an IR-code, we have seen how to generate a mapping between families. When $I = O$, the least fixed point of this mapping is called an inductive recursive type. In Agda, we can implement the family (U, T) arising from an IR-code c as follows:

```
mutual
    data $U$ $\{D : \mathsf{Set}_1\}$ $(c : \mathsf{IR}\ D\ D)$  :  Set where
      $C : (\pi_0 \circ [\![c]\!])\ (F\ \ c) \to U\ \ c$

    $T : \{D : \mathsf{Set}_1\} \to (c : \mathsf{IR}\ D\ D) \to U\ c \to D$
    $T\ \ c\ \ (C\ a) = (\pi_1 \circ [\![c]\!])\ \ (F\ c)\ \ a$

    $F : \{D : \mathsf{Set}_1\} \to (c : \mathsf{IR}\ D\ D) \to \mathsf{Fam} D$
    $F\ \ c = (U\ c\ ,\ T\ c)$
```

Of course, using mixed variant IR-codes has the advantage that we could define a more general fixed point operator IR $(I + O)$ $O \to$ Fam $I \to$ Fam O but this is standard and so we don't do it here. Our final construction is to note that the construction of IR codes is itself mixed variant, that is IR is contravariant in its first argument and covariant in its second argument. This is implemented in Agda as follows:

$$\begin{aligned}
&\_\circ_l\_ : \{I\ I'\ O : \mathsf{Set}_1\} \to \mathsf{IR}\ I\ O \to (I' \to I) \to \mathsf{IR}\ I'\ O\\
&\iota\ d\ \circ_l\ f = \iota\ d\\
&\sigma(A\ ,\ g)\ \circ_l\ f = \sigma(A\ ,\ \lambda a \to ga\ \circ_l\ f)\\
&\delta(A\ ,\ F)\ \circ_l\ f = \delta(A\ ,\ \lambda k \to F(f \circ k)\ \circ_l\ f)
\end{aligned}$$

$$\begin{aligned}
&\_\circ_r\_ : \{I\ O\ O' : \mathsf{Set}_1\} \to (O \to O') \to \mathsf{IR}\ I\ O \to \mathsf{IR}\ I\ O'\\
&f\ \circ_r\ \iota\ d = \iota\ (fd)\\
&f\ \circ_r\ \sigma(A\ ,\ g) = \sigma(A\ ,\ \lambda a \to f\ \circ_r\ (g\ a))\\
&f\ \circ_r\ \delta(A\ ,\ F) = \delta(A\ ,\ \lambda k \to f\ \circ_r\ Fk)
\end{aligned}$$

4. Container Based IR

Having three constructors is worrying. It may not seem so at first glance, but it wrankles as there seems to be something non-canonical in the presentation of IR we have developed. Further, when the semantics of σ and δ are presented using a localisation for LFam, these two constructors seem similar. Such potential unity between σ and δ is crying out to be formalised and this leads us to wonder if there is a common pattern underlying these constructors ... and then Thorsten Altenkirch pointed out what that common pattern was. In more detail, Thorsten pointed out we don't need to have three constructors as the constructors σ and δ can naturally be compressed into one constructor which leads to a containerification of induction recursion. In this section we develop container based IR because it both i) further simplifies our understanding of the essence of induction recursion by highlighting even further the fundamental role played by localisation; and ii) provides an essential stepping stone on the road to indexed induction recursion.

Definition 9 (Container). A container (Abott et al.(2005)Abott, Altenkirch, and Ghani) (S, P) is a pair where S is a set P is a function $P : S \to \mathsf{Set}$.

In Agda we may define

$$\begin{aligned}
&\mathsf{Cont} : \mathsf{Set}_1\\
&\mathsf{Cont} = \mathsf{Fam}\ \mathsf{Set}
\end{aligned}$$

In the ordinary theory of data types, one may think of S as containing operator symbols and P assigns to each such operator, a set of positions where data is stored. This *shapes-and-positions* metaphor is formalised by showing how containers define operations on sets as the following Agda definition shows

$$\begin{aligned}
&[-]_0 : \forall\{a\} \to \mathsf{Cont} \to \mathsf{Set}\ a \to \mathsf{Set}\ a\\
&[(S, P)]_0\ \ X = \Sigma\ \ S\ \ (\lambda s \to Ps \to X)
\end{aligned}$$

$$\begin{aligned}
&[-]_1 : \forall\{a\ b\} \to \forall\{X : \mathsf{Set}\ a\} \to \forall\{Y : \mathsf{Set}\ b\}\\
&\qquad\qquad \to (H : \mathsf{Cont}) \to (X \to Y) \to [H]_0 X \to [H]_0 Y\\
&[(S, P)]_1\ \ f\ \ (s\ ,\ g) = (s\ ,\ f \circ g)
\end{aligned}$$

Notice that an element of $[(S, P)]_0\ X$ consists of a choice of a shape, *i.e.* an element of $s : S$, and a function $f : Ps \to X$ which assigns to every position of the operator s, an element of the type X to be stored there. We call $[(S,P)]_0$ the *extension* of the container. Just like with Fam and LFam, not only do containers operate on sets, but they also lift functions between sets to functions between the resulting sets as shown by the definition of the function $[-]_1$ above. Here are two examples of containers and their extensions:

Example 10. Fix a set A. The operator sending a set X to A is the extension of the container KA defined in Agda as follows:

$$\begin{array}{l} K : \mathsf{Set} \to \mathsf{Cont} \\ K\ S = (S\ ,\ \mathsf{const}\ \bot) \end{array}$$

Example 11. Fix a set A. The operator sending a set X to $A \to X$ is the extension of the container RA defined in Agda as follows:

$$\begin{array}{l} R : \mathsf{Set} \to \mathsf{Cont} \\ R\ A = (1\ ,\ \mathsf{const}\ A) \end{array}$$

Infact all contianer functors are examples of IR-codes when $D = \mathsf{One}$. This is demonstrated by the following example

Example 12 (Containers and IR-codes). If (S,P) is any container, then $[S,P]_0 : \mathsf{Set} \to \mathsf{Set}$ is the decoding of the IR-code

$$\sigma_S(\lambda s.\, \delta_{Ps}(\lambda\_.\, \iota *)) : \mathsf{IR\ One}$$

Indeed, it is an easy inductive proof to see that the class of IR-functors when $D = \mathsf{One}$ is exactly the class of containers. A simple embellishment of this code assigns, to each element of the initial algebra of a container, the set of paths through it from the root to some subtree. The code here is

$$\sigma_S(\lambda s.\, \delta_{Ps}(\lambda X : Ps \to \mathsf{Set}.\, \iota\,(1 + \Sigma\,(Ps)\,X))) : \mathsf{IR}(\mathsf{Set})$$

We are almost ready to define the codes and decoding functions for container based IR. But to do this we need first to understand what happens in container based IR to the σ and δ constructors. As mentioned earlier, they are amalgamated into one single type constructor generalising both Fam and LFam from the previous section. This gives us the operator on types LCFam which in Agda may be defined by

$$\begin{array}{l} \mathsf{LCFam} : \mathsf{Set}_1 \to \mathsf{Set}_1 \to \mathsf{Set}_1 \\ \mathsf{LCFam}\ I\ O = \Sigma\ \mathsf{Cont}\ (\lambda H \to [H]_0\ I \to O) \end{array}$$

Just like LFam, LCFam is contravariant in its first argument and covariant in its second argument. We will need the covariant action here and so give its definition in Agda now

$$\begin{array}{l} \mathsf{LCFam}_1 : \{I\ \ D\ \ D' : \mathsf{Set}_1\} \to (D \to D') \to \mathsf{LCFam}\ \ I\ \ D \to \mathsf{LCFam}\ \ I\ \ D' \\ \mathsf{LCFam}_1\ \ g\ \ (A\ \ ,\ \ \ F) = (A\ \ ,\ \ g \circ F) \end{array}$$

Note that Fam and LFam are special cases where the chosen container is taken to be firstly the container $K\ A$ which represents the constantly A-valued opertator, and secondly the container $R\ A$ which represents the operator $A \to -$. Thus container based IR has the same expressive power as Dybjer and Setzer's induction recursion. Container based IR simply presents induction recursion differently and, crucially, this presentational difference will be fundamental to a smooth generalisation from induction recursion to indexed induction recursion possible. With these definitions in place, container based IR can be defined to have the following codes:

$$\begin{array}{l} \mathsf{data}\ \ \mathsf{IR}(I\ \ O : \mathsf{Set}_1) : \mathsf{Set}_1\ \ \mathsf{where} \\ \quad \iota : O \rightarrow \mathsf{IR}\ \ I\ \ O \\ \quad \sigma\delta : \mathsf{LCFam}\ \ I\ \ (\mathsf{IR}\ \ I\ \ O) \rightarrow \mathsf{IR}\ \ I\ \ O \end{array}$$

As with the presentation of induction recursion in the previous section, the above data type shows that $\mathsf{IR}\ \ I\ \ O$ is the least fixed point of an operator on types. This time the operator sends a type X to the type $O + \mathsf{LCFam}\ I\ X$. For those with some categorical background, it is worth remarking that when put in this form, $\mathsf{IR}\ \ I\ \ O$ is also the free monad over $\mathsf{LCFam}\ I$ at O. This seems a relatively clean description which shows that the essence of IR-codes are captured by the operator LCFam. Returning to the characterisation of $\mathsf{IR}\ \ I\ \ O$ as a least fixed point, this gives us the following fold-operator

$$\begin{array}{l} \mathsf{fold} : \{I\ \ O\ \ X : \mathsf{Set}_1\} \rightarrow (O \rightarrow X) \rightarrow (\mathsf{LCFam}\ \ I\ \ X \rightarrow X) \rightarrow \mathsf{IR}\ \ I\ \ O \rightarrow X \\ \mathsf{fold}\ \ i\ \ k\ \ (\iota\ \ o) = i\ \ o \\ \mathsf{fold}\ \ i\ \ k\ \ (\sigma\delta\ \ F) = k\ \ (\mathsf{LCFam}_1\ \ (\mathsf{fold}\ \ i\ \ k)\ \ F) \end{array}$$

Learning from the presentation of the previous section, we can use this fold operator to write a decoding function for each code if we can define a localisation operator for LCFam. Indeed, this can be done as the following Agda code shows:

$$\begin{array}{l} \rho : \{I\ O : \mathsf{Set}_1\} \rightarrow \mathsf{LCFam}\ I\ O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O \\ \rho\ (H\ ,\ F)\ (B\ ,\ f) = ([H]_0 B\ ,\ F \circ [H]_1 f) \end{array}$$

Notice how the localisation operator for LCFam, is similar to that for LFam. Indeed, when H is the contianer RA, the localisation function we have just defined for LCFam turns out to be exactly that for LFam. Given this localisaton operator, we can define the decoding function for container IR codes as follows:

$$\begin{array}{l} [\![-]\!] : \{I\ O : \mathsf{Set}_1\} \rightarrow \mathsf{IR}\ I\ O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O \\ [\![c]\!]\ \ m = \mathsf{fold}\ \ \eta\ \ (\mu \circ \rho m)\ \ c \end{array}$$

Of course, the reader may want to know concretely what this decoding formula computes and so we answer that question now:

Lemma 13. Let $\sigma\delta(H\ ,\ F) \in \mathsf{IR}\ I\ O$ and $(U,T) \in \mathsf{Fam}\ I$. Then

$$[\![c]\!](U\ ,\ T) = \Sigma(s,f){:}[H]_0 U\ .\ [\![F(s, T \circ f)]\!](U\ ,\ T)$$

Proof. Direct calculation □

The fact that the decoding function remains very similar to that presented in the previous section is very pleasing! Our understanding of IR is beginning to become sufficient that we can make changes to the theory of induction recursion locally, eg change the number of constructors, without having to redevelop the whole of the theory of induction recursion for our new theory. In essence we are beginning to see a modular understanding of IR develop where we can change parts of the theory without affecting other parts. And, the key features of induction recursion seems to be i) a large constructor like $\mathsf{Fam}, \mathsf{LFam}$ or LCFam from which codes for IR can be derived; ii) an operation which lifts functions between types to functions between the large families defined over those types from which an associated fold-operator can be derived; and iii) a localisation operator for the said notion of large family from which a decoding function for IR-codes can be derived. As we shall see, this axiomatic framework will stand us in good stead when we come to tackle indexed induction recursion.

As remarked above, the presence of separate σ and δ constructors, or a single $\sigma\delta$ constructor doesn't change the expressive power of induction recursion as each system can be defined in terms of the other. But the use of $\sigma\delta$, and more generally, the containerification of induction recursion, allows us to import some sophisticated mathematics concerning containers to help us understand induction recursion. We also remark that one can go beyond Thorsten's remark in that there is nothing special about containers. Let $\mathcal{K}$ be any class of operators on sets. We can define the inference rule

$$\frac{K \in \mathcal{K} \quad F : KI \to \mathsf{IR}\ I\ O}{\sigma\delta^{\mathcal{K}}_{K} F \in \mathsf{IR}\ I\ O}$$

and give this constructor an associated semantics via a decoding function. The natural questions to ask here regards the nature of the transformation of a class of operators $\mathcal{K}$ into the class of operators definable using the form of induction recursion with $\sigma\delta^{\mathcal{K}}$ as a constructor. Is it inflationary? What is its fixed point? Having asked such natural questions, we will not pursue them in this paper.

5. Indexed Induction Recursion

Dybjer and Setzer did more than simply define induction recursion — they also defined indexed induction recursion (Dybjer and Setzer(2006)). To motivate indexed induction recursion, recall that within induction recursion we can define universes $T : U \to D$ where D can be, say, Set or $\mathsf{Set} \to \mathsf{Set}$. As a result, every code $u : U$ in a universe $T : U \to D$ decodes to an element of the same type, namely D. But what if we want to define a universe containing codes codes some of which decode to Set and some of which decode to operators on Set. Indexed induction recursion gives us exactly the technology to do this, *i.e.* to define universe with codes that decode to inhabitants of different types.

As with induction recursion, the theory of indexed induction recursion is fairly delicate and intricate type theory. In this section we aim to do for indexed induction recursion what we have already done for induction recursion, *i.e.* produce an algebraic foundation of indexed induction recursion and use it to derive a clean and concise implementation of indexed induction recursion. To do this, we will reuse the methodology developed in the previous two sections; that is we will use large families to build the codes of indexed induction recursion, define an fold operator and then use localisation to model their decoding. The notion of large families we use is also interesting. To study induction recursion we used large families built from containers and hence it is pleasing that our study of indexed induction recursion uses large families built from indexed containers (Morris and Altenkirch(2009)). Indeed, it is pleasing that this move from containers to indexed containers (and the associated notion of localisation) is the only change that is needed to move generalise our treatment of induction recursion to indexed induction recursion. As a technical point, we point out that we are actually treating Dybjer and Setzer's restricted form of indexed induction recursion.

5.1. *Indexed Containers:*

Containers are designed to model those operations on sets which conform to the shapes and positions metaphor. In the introduction, we descibed operators on sets as forming the simplest format of data type definition and suggested that the next level of sophistication arises when one considers data types arising as least fixed points of operators on I-indexed sets for a fixed set I. A natural question is: which operators on I-indexed sets conform to the shapes and positions metaphor and indexed containers provide an answer to that question. It transpires that it is easier to describe indexed containers in a two stage process where in the first stage we index only the input and then, in the second stage, we index both the input and output. So we start with this half-way house which we call *input indexed containers*.

Definition 14 (Input Indexed Container and its Extension). Let I be a set. An input I-indexed container consists of a pair (S, P) where $S : \mathsf{Set}$ and $P : S \to I \to \mathsf{Set}$. The extension of an input I-indexed container (S, P) is the operation $[S, P] : (I \to \mathsf{Set}) \to \mathsf{Set}$ defined by

$$[S, P]\ X = \Sigma s\!:\!S.\ \ \Pi i\!:\!I.\ \ P\ s\ i \to X\ i$$

Notice how, as with containers, an input indexed container has a set S of operators and a function P which assigns to each operator $s : S$ some positions that will store data. However, with an input indexed container, these positions are I-sorted which leads to the above type. The extension of an input indexed container therefore maps an I-indexed set X to the set consisting of the choice of a shape s and, for each sort i and position for s with sort i, an element of Xi. If we introduce the abbreviation $X \Rightarrow X'$ for the type $\Pi i\!:\!I.\ Xi \to X'i$, then the extension of an input indexed container looks just like that of a container.

$$[S, P]\ X = \Sigma s\!:\!S.\ P\ s \Rightarrow X$$

We can implement input indexed containers and their extension in Agda as follows

$\mathsf{ICont} : \mathsf{Set} \to \mathsf{Set}_1$
$\mathsf{ICont}\ \ I = \Sigma\ \ \mathsf{Set}\ \ (\lambda S \to (S \to I \to \mathsf{Set}))$

$\_ \Rightarrow \_ : \forall\{a\ \ b\} \to \{I : \mathsf{Set}\} \to (P : I \to \mathsf{Set}\ \ a) \to (Q : I \to \mathsf{Set}\ \ b) \to \mathsf{Set}(a \cup b)$
$P \Rightarrow Q = \forall\{i\} \to P\ \ i \to Q\ \ i$

$[\![-]\!]^I_0 : \forall\{a\} \to \{I : \mathsf{Set}\} \to \mathsf{ICont}\ \ I \to (I \to \mathsf{Set}\ \ a) \to \mathsf{Set}\ \ a$
$[\![(S, P)]\!]^I_0\ \ X = \Sigma\ \ S\ \ (\lambda s \to Ps \Rightarrow X)$

As with containers, input indexed contianers lift maps between their inputs to maps between their outputs. We implement that in Agda as follows

$$\begin{aligned}
&[\![-]\!]^I_1 : \forall\{a\ \ b\} \to \{I : \mathsf{Set}\} \to (H : \mathsf{ICont}\ \ I) \to \{P : I \to \mathsf{Set}\ \ a\} \to \{Q : I \to \mathsf{Set}\ \ b\}\\
&\qquad \to (f : P \Rightarrow Q) \to [\![H]\!]^I_0 P \to [\![H]\!]^I_0 Q\\
&[\![(S,T)]\!]^I_1\ \ f\ \ (s\ \ ,\ \ g) = (s\ \ ,\ \ \lambda x \to f\ \ (g\ \ x))
\end{aligned}$$

5.2. *From Indexed Containers to Indexed Induction Recursion*

Recall that from the previous section, containers were used to define large container families which in turn were used to define the syntax of IR-codes. Interestingly the same process works here - now that we have defined input indexed containers, we define the following notion of a *large, input indexed container family* as follows

$$\begin{aligned}
&\mathsf{LICFam} : (D : I \to \mathsf{Set}_1) \to \mathsf{Set}_1 \to \mathsf{Set}_1\\
&\mathsf{LICFam}\ \ D\ \ E = \Sigma\ \ (\mathsf{ICont}\ \ I)\ \ (\lambda H \to [\![H]\!]^I_0\ \ D \to E)
\end{aligned}$$

Notice how the definition of a large input indexed container family is almost exactly the same as for a large container family. The fact that D is no longer a type but an I-indexed set of types is carefully abstracted within the notion of an input indexed container and its extension. As with our previous notions of large families, LICFam is negative in its first argument and positive in its second argument. This means we can define the following lifiting of functions between its second argument. As with our treatment of IR and container based IR, this lifting will be used to define a fold operator for the associated set of codes.

$$\begin{aligned}
&\mathsf{LICFam}_1 : \{I : \mathsf{Set}\} \to \{D : I \to \mathsf{Set}_1\} \to \{X\ \ Y : \mathsf{Set}\} \to\\
&\qquad (X \to Y) \to \mathsf{LCFam}\ \ D\ \ X \to \mathsf{LCFam}\ \ D\ \ Y\\
&\mathsf{LICFam}_1\ \ f\ \ (H\ \ ,\ \ F) = (H\ \ ,\ \ f \circ F)
\end{aligned}$$

In the previous section we defined IR-codes to be the least fixed point of large container families and the same approach works here. That is, we define the codes of input indexed induction recursion to be the following least fixed point

$$\begin{aligned}
&\mathsf{data}\ \ \mathsf{IIR}\ \ \{I : \mathsf{Set}\}\ \ (D : I \to \mathsf{Set}_1)\ \ (E : \mathsf{Set}_1) : \mathsf{Set}_1\ \ \mathsf{where}\\
&\quad \iota^I : E \to \mathsf{IIR}\ \ D\ \ E\\
&\quad \sigma\delta^I : \mathsf{LICFam}\ \ D\ \ (\mathsf{IIR}\ \ D\ \ E) \to \mathsf{IIR}\ \ D\ \ E
\end{aligned}$$

Note how, just as input indexed containers allow the input to be I-indexed, so too does the above definition of input indexed induction recursion. Continuing with our general approach, we define a fold-operator for IIR D E as follows

$$\begin{aligned}
&\mathsf{fold}^I : \{I : \mathsf{Set}\} \to \{D : I \to \mathsf{Set}_1\} \to \{E : \mathsf{Set}_1\} \to \{X : \mathsf{Set}_1\} \to\\
&\qquad (E \to X) \to (\mathsf{LICFam}\ \ D\ \ X \to X) \to \mathsf{IIR}\ \ D\ \ E \to X\\
&\mathsf{fold}^I\ \ i\ \ k\ \ (\iota^I\ \ e) = i\ \ e\\
&\mathsf{fold}^I\ \ i\ \ k\ \ (\sigma\delta^I\ \ F) = k\ \ (\mathsf{LICFam}_1\ \ (\mathsf{fold}^I\ \ i\ \ k)\ \ F)
\end{aligned}$$

Once we have a fold-operator, all we need to do to define the decoding funnction for codes of type IIR is to introduce a notion of localisation for LICFam. The following is the natural generalisation of the localisation function for large container families.

$$
\begin{aligned}
&\rho^I : \{I : \mathsf{Set}\} \to \{D : I \to \mathsf{Set}_1\} \to \{E : \mathsf{Set}_1\} \to \Pi\ \ I\ \ (\mathsf{Fam} \circ D) \to \mathsf{LICFam}\ \ D\ \ E \to \mathsf{Fam}\ \ E \\
&\rho^I\ \ \{I\}\ \ \{D\}\ \ \{E\}\ \ (\lambda'\ \ \Phi)\ \ (H\ \ ,\ \ F) = ([\![H]\!]^I_0 V\ \ , F \circ [\![H]\!]^I_1 W) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ \ V : I \to \mathsf{Set} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad V = \pi_0 \circ \Phi \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad W : V \Rightarrow D \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad W\{i\} = \pi_1(\Phi\ \ i)
\end{aligned}
$$

Having a fold operation and a localisation means of course we can define the decoding function for these codes as follows

$$
\begin{aligned}
&[\![-]\!]^I : \{I : \mathsf{Set}\} \to \{D : I \to \mathsf{Set}_1\} \to \{E : \mathsf{Set}_1\} \to \mathsf{IIR}\ \ D\ \ E \to \Pi\ \ I\ \ (\mathsf{Fam} \circ D) \to \mathsf{Fam}\ \ E \\
&[\![c]\!]^I\ \ m = \mathsf{fold}^I\ \ \eta\ \ (\mu \circ \rho^I m)\ \ c
\end{aligned}
$$

5.3. *Fully Indexed Induction Recursion*

In the previous section we indexed the input of a container to derive input indexed containers and use them to derive a notion of input indexed induction recursion where input families were indexed. We finish the paper by indexing both the input and output of firstly indexed containers and then indexed induction recursion. There is not, however, any need to define a new large families functor *etc.* Instead, we simply define

$$
\begin{aligned}
&\mathsf{DICont} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\
&\mathsf{DICont}\ \ I\ \ J = J \to \mathsf{ICont}\ \ I
\end{aligned}
$$

Thus indexing on the output doesn't require the definition of any new types, but can be built modularly from the input indexed contianers of the last subsection. A fully indexed contianer defines a map of indexed sets as follows

$$
\begin{aligned}
&\langle - \rangle_0 : \forall\{a\}\ \ \{I\ \ J : \mathsf{Set}\} \to \mathsf{DICont}\ \ I\ \ J \to (I \to \mathsf{Set}\ \ a) \to J \to \mathsf{Set}\ \ a \\
&\langle \Phi \rangle_0\ \ X\ \ j = \langle \Phi j \rangle_0\ \ X
\end{aligned}
$$

As ever, this extension of a fully indexed contianer has an action of maps between indexed families.

$$
\begin{aligned}
&\langle - \rangle_1 : \forall\{a\ \ b\} \to \{I\ \ J : \mathsf{Set}\} \to (H : \mathsf{DICont}\ \ I\ \ J) \to \{P : I \to \mathsf{Set}\ \ a\} \to \{Q : I \to \mathsf{Set}\ \ b\} \\
&\qquad \to (f : P \to Q) \to \langle H \rangle_0 P \Rightarrow \langle H \rangle_0 Q \\
&\langle \Phi \rangle_1\ \ f\ \ \{j\} = [\![\Phi\ \ j]\!]^I_1 f
\end{aligned}
$$

This then gives us fully indexed IR codes as follows

$$
\begin{aligned}
&\mathsf{DIIR} : \{I\ \ J : \mathsf{Set}\} \to (D : I \to \mathsf{Set}_1) \to (E : J \to \mathsf{Set}_1) \to \mathsf{Set}_1 \\
&\mathsf{DIIR}\ \ \{I\}\ \ \{J\}\ \ D\ \ E = (j : J) \to \mathsf{IIR}\ \ I\ \ D\ \ (E\ \ j)
\end{aligned}
$$

These codes then have the following decoding function

$$
\begin{aligned}
&\langle - \rangle : \{I\ \ J : \mathsf{Set}\} \to \{D : I \to \mathsf{Set}_1\} \to \{E : J \to \mathsf{Set}_1\} \to \\
&\qquad\qquad\qquad\qquad \mathsf{DIIR}\ \ D\ \ E \to \Pi\ \ I\ \ (\mathsf{Fam} \circ D) \to \Pi\ \ J(\mathsf{Fam} \circ E) \\
&\langle \Phi \rangle\ \ m = \lambda'\ \ (\lambda\ \ j \to [\![\Phi\ \ j]\!]^I m)
\end{aligned}
$$

6. A Categorical Perspecitve on These Matters

This paper has taken a deliberately non-categorical approach to the foundations of induction recursion so as to make the paper as accessible as possible to functional programmers. While recognising the merits of this approach - it is our opinion that functional programmers would benefit from understanding induction recursion and hence one must write in a language which they are familiar with - we can't help but feel that something is lost without the precision of category theory. So, in this section, we briefly recast our work in more categorical terms for those with a categorical background. A full categorical treatment will be the subject of a different paper.

Firstly, of course, the theory of data types is best understood via initial algebra semantics within which data types a specified by functors $F : \mathcal{C} \to \mathcal{C}$ on a category $\mathcal{C}$. The heirarchy of data types mentioned in the introduction can be seen as a choice of sophistication of the category $\mathcal{C}$: i) for simple inductive types one may choose $\mathcal{C}$ to be Set or some other basic category such as ω-cpo; ii) for inductive families one considers $\mathcal{C}$ to typically be a slice category, *e.g.* a categopry of the form $\mathcal{B}/I$ or a functor category of the form $\mathcal{B}^I$; and iii) for induction recursion one takes $\mathcal{C}$ to be a category of the form $\mathsf{Fam}(D)$ where morphisms are the cartesian maps.

That we require F to be a functor within initial algebra semantics formalises the requirement that the various operators we consider within this paper come with a lifting of functions between types to functions between the action of the operator on those types. The data type itself is modelled as the carrier of the inital algebra μF, while the constructors form the actual inital algebra $in : F(\mu F) \to \mu F$. Initiality ensures that for any other F-algebra $h : FA \to A$ there is a unique F-algebra homomorphism $\mathit{fold}\, h$ from the initial algebra $in : F(\mu F) \to \mu F$ to that algebra. For each functor F, the map $\mathit{fold} : (FA \to A) \to \mu F \to A$ is the fold operator for the data type μF. This construction underlies the fold-operators we derived for the various different data types presented within this paper.

The next piece of light shed upon induction recursion by category theory has already been alluded to briefly in this paper. Using container based induction recursion, we see that $\mathsf{IR}\ I\ O$ is the free monad on $\mathsf{LCFam}\ I$. This is reassuring as it means induction recursion can be dealt with with the usual mathematical tools for dealing with algebraic theories built from operations with given arities. Indeed, one starts to see monads in many different parts of the theory of induction recursion, *e.g.* not only is $\mathsf{IR}\ I\ O$ the free monad over $\mathsf{LCFam}\ I$, but also Fam is a monad and the unit and multiplication of this monad play a key role in the decoding function for IR-codes. One may even wonder what is so special about the families monad and whether other monads could be used.

Indeed, when one looks at these questions categorically, a remarkably simple picture emerges. Given a mixed variance functor F and a monad M, a localisation is then a family of maps $\rho : F\ I\ O \to MI \to MO$ which are natural in O. As expected, in this setting the IR codes are given once more by the free monad construction

$$\begin{array}{l} \mathsf{data}\ \ \mathsf{IR}(I\ \ O : \mathsf{Set}) : \mathsf{Set}\ \ \mathsf{where} \\ \quad \iota : O \to \mathsf{IR}\ \ I\ \ O \\ \quad \sigma\delta : F\ \ I\ \ (\mathsf{IR}\ \ I\ \ O) \to \mathsf{IR}\ \ I\ \ O \end{array}$$

To understand the decoding function in this abstract setting, recall that for a given set X, the functor $X \to -$ is called the reader monad with state X and is written R_X. The monad transformer for R_X takes a monad M as input and returns the monad $X \to M-$. Thus $M\ I \to M-$ is the reader monad transformer with state $M\ I$ applied to M. Thus a localiastion $\rho : F\ I\ O \to M\ I \to M\ O$ is a natural transformation from the functor $F\ I$ to the monad $M\ I \to M-$ and as such induces a monad morphism from the free monad on $F\ I$ to this monad. This monad morphism is exactly the decoding function! Wow!

7. Conclusion and Further Work

We think that Dybjer and Setzer's work on induction recursion is of outstanding potential as it opens the way to the next generation of data type definitions which are capable of creating universes closed under depedently typed operations. We have sought to help develop the field of induction recursion in two ways:

- We have produced a conceptual analysis of induction recursion based upon the notions of i) large families from which IR-codes can be derived; ii) liftings of functions between types to functons between the large families built from those types from which fold-operators can be built; and iii) localisation from which the decoding function for IR-codes can be defined.

- We have produced an implementation of our conceptual analysis in the prgramming language Agda so as to demonstrate the robustness of the conceptual analysis described above. Indeed, our analysis was proven to be able to decribe induction recursion, container based induction recursion and indexed induction recursion in a uniform manner. This suggests that the concepts highlighted above are of some importance.

Future Work: There is certainly much more work required to fully understand induction recursion. It is an enormously powerful definitional principle, arguable one that takes us to the very limits of the realm of data structures and code whose semantics can be defined predicatively (Kahle and Setzer(2010)) – beyond which lies the bottomless abyss of system F, F^ω, impredicative higher-order logic, topos-theory, and the like. At the theoretical level we wish to deepen the connection between containers and induction recursion by emulating the key theorem from containers which states that containers and their morphisms have a full and faithfull embedding into the category of endofunctors on Set. That is, we wish to define morphisms between inductive recursive definitions and show that they uniquely represent natural transformations. This requires some sophisticated mathematics, centrally the characterisation of the semantics of the δ-constructor as a left Kan extension.

In another direction, we wish to render that mapping in a computationally relevant form so that it can be exploited by functional programmers. Once we have a category $\mathsf{IR}\ I\ O$ of IR codes and IR morphisms, we will wish to understand its properties better. The category of containers is in fact, slightly surprisingly, cartesian closed (Altenkirch et al.(2010)Altenkirch, Levy, and Staton; Hasegawa(2002)) and hence the question arises whether the same can be said of $\mathsf{IR}\,I\,O$. A positive answer here will allow the technology of higher-order functions familiar to functional programmers to be deployed when dealing with IR codes. Further, once we start asking about cartesian closure, we will inevitably need to focus on other categorical structures possesed by IR-codes. Note: this will be far from merely a handle-turning exercise. The structure of the category $\mathsf{Fam}\,I$ is much more austere than that of the familiar categories Set, Set^I, $\mathsf{Set} \to \mathsf{Set}$ and the like. For example, it lacks a terminal object in general (unless I is small).

Perhaps, the key contribution in this paper is to reveal the central role localisation plays in induction recursion. But from where does that localisation arise? Why is it that we can turn *large* families into the action of functions on *small* families. We conjecture this is a very deep issue linked to the local smallness of the category Set. While sets are 'without number', between any two sets there is only a small set of functions. We believe this is fundamentally what makes induction recursion tick and aim to substantiate this view. And, finally, as with all good research, and hopefully with this paper, we wish to expoloit theoretical results such as the above to help the programmer write and structure better code. This is both our ultimate test and ambition.

References

M. Abott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

T. Altenkirch, P. Levy, and S. Staton. Higher Order Containers. *Computability in Europe*, 2010.

A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, June 2000.

P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Log.*, 124 (1-3):1–47, 2003.

P. Dybjer and A. Setzer. Indexed induction-recursion. *J. Log. Alg. Prog.*, 66:1–49, 2006.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Types for proofs and programs*, volume 3085 of *LNCS*, pages 210–225. Springer-Verlag, 2004.

R. Hasegawa. Two applications of analytic functors. *Theor. Comput. Sci.*, 272(1-2):113–175, 2002. ISSN 0304-3975. .

R. Kahle and A. Setzer. An extended predicative definition of the Mahlo universe. In *Ways of ProofTheory. Festschrift on the occasion of Wolfram Pohler's retirement*. Ontos Verlag, 2010.

S. Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer, 2nd edition, September 1998. ISBN 0387984038.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

P. Morris and T. Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

J. M. Smith. Propositional functions and families of types. *Notre Dame J. Form. Log.*, 30(3): 442–458, 1989.