# IR for the Masses

... well, the FP Masses at least

Neil Ghani
University of Strathclyde
ng@cis.strath.ac.uk

Peter Hancock
University of Strathclyde
hancock@spamcop.net

## Abstract

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 offers the possibility of defining universes of objects (primarily, types) closed under given dependently typed operations. The key feature of induction recursion is that the codes in the universe of types are built up inductively at the same time as the recursive definition of their decoding function.

Despite this potential, induction recursion has not become as widely understood or used as it should have been. We believe this is in part because: i) there is still scope for analysing the foundation of the induction recursion; and ii) because 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 presentation of induction recursion to complement the original type theoretic one; and ii) reflect this new understanding into implementations which broaden the accessibility of induction recursion to non-theorists.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features-Data types and structures

***General Terms*** Dependently Typed Programming, Category Theory, Type Theory

***Keywords*** Induction Recursion

## 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, 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 [5–8], that first appeared with the notion of universe, or type of types, introduced by Martin-Löf in the early 70's.

It is our opinion that members of the TLDI audience 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. At the simplest level, one models data types as initial algebras of functors on the category $\mathsf{Set}$ of sets. Natural numbers, lists storing data of a given type, binary trees etc fit into this framework. However, many important data types do not. As a result, various, more sophisticated, categories have been suggested as a place where initial algebra semantics can be deployed so as to capture such advanced data types. For example, many-sorted data types can be modelled as initial algebras of functors over categories of the form $\mathsf{Set}^I$ for a fixed set $I$, data types with variable binding have been studied as initial algebras of functors over various presheaf categories, nested types can be modelled as initial algebras of functors over the category of endofunctors $\mathsf{Set} \to \mathsf{Set}$. These semantic treatments have been accompanied by syntactic grammars for generating functors over such categories. Examples of such grammars are polynomial functors, strictly positive functors, containers, indexed containers, dependent polynomials, analytic functors, inductive families etc.

Now, each of the above theories of data types can be seen as attempting to define indexed data types $T : U \to X$ where $U$ is a collection of indexes and $X$ is something like $\mathsf{Set}$. Induction recursion is the next step in this line of research where, for the first time, the set $U$ is defined simultaneously with the function $T$. Formalising this idea, induction recursion proposes to i) represent data types as initial algebras of functors over categories of families; and ii) develops a syntactic system of codes for representing such functors. 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 such universes exactly require the simultaneous definition of their index set and decoding function. Such examples cannot be easily represented within any of the theories of data types mentioned in the previous paragraph. Examples of such inductive recursive definitions will be given in the body of the paper.

However, despite the fundamental analysis 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 theoretical foundations and pragmatic aspects of induction recursion need further development. This paper proposes to address this by i) developing a categorical presentation of induction recursion to complement the type-theoretical formulations of Dybjer and Setzer; and ii) following in the footsteps of the algebra of programming school and reflecting this categorical structure into code. This methodology has proved to be very fruitful in the past probably because both functional programming and category theory seek to understand computational phenomena using high levels of abstraction which can be reflected into structured programming idioms. More concretely, providing implementations of categorical concepts helps make our paper more accessible to those whose category theory is not as strong as it could be and it also gives programmers code to experiment and play with. On the other hand, the implementation of our ideas also guarantees their partial correctness via type checking.

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 investigate the families monad so as to work uniformly over families rather than to first deal with indexing sets and then with decoding functions. We also separate out co- and contravariant aspects of induction recursion which allows us to give a universal property characterising the set of IR codes. Finally, we introduce briefly the key notion of reflection. [1]
- In section 4 we show how induction recursion can be reformulated using containers. This allows us to compactify the definition of IR codes and to define morphisms between inductive recursive definitions which represent natural transformations in the same way that IR codes represent functors.
- In section 5 we return to the topic of reflection and show how the essence of induction recursion is contained within the concept of reflection. This allows us to understand the essence of induction recursion as a question of size and to parameterise induction recursion by the choice of a reflection.
- In section 6 we show that the theory of induction recursion can be localised to a universe. This allows us to avoid having to talk vaguely about large sets and instead use the well known territory of the category $\mathsf{Set}$ of small sets as the basis for our work.
- Finally, in section 7 we show there is nothing special about families and replace them with an arbitrary strong monad which we combine with the reader monad.

This step by step process is designed to explain a number of innovations which we have introduced so as to simplify definition of IR codes to that of being a free monad and whose decoding function is just the unique mediating morphism from this free monad to another monad generated by a reflection function.

In this paper we assume a category $\mathsf{SET}$ of *large* sets, a category $\mathsf{CAT}$ of *large* categories which contains as an object $\mathsf{Set}$, the category of *small* sets. These size issues reflect analogously delicate size issues which arise in Dybjer and Setzer's work. We assume the reader is familiar with only simple categorical constructions such as categories, functors, natural transformations, initial algebras, and monads. We denote the one point set and the category with one object and one morphism by $1$ - which meaning is intended can be inferred from the context. If a category $\mathcal{C}$ has $\mathsf{Set}$-indexed coproducts, $A$ is a set and $B : A \to |\mathcal{C}|$ is an $A$-indexed collection of objects of $\mathcal{C}$, then we denote by $\Sigma a : A.\ Ba$ the coproduct of this collection. If $f : A \to C$ and $g : B \to C$ are morphisms in a category, we denote by $[f, g] : A+B \to C$ the cotuple of $f$ and $g$.

In terms of programming, we use the dependently typed programming language Agda to present a functional programming perspective on this paper. Those not familiar with this dependently typed programming language can either find more details at [3] 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 "$\{x : A\} \to$ ..." 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`. As explained above, the inclusion of this Agda code serves several purposes in terms of increasing the accessibility, usability and correctness of our paper and we feel these benefits outweigh the overhead of requiring the reader to read the Agda code. Its a testament to the progress in dependently typed programming that we now have a language which is abstract enough to closely mirror the mathematical foundations of induction recursion with relatively minor overhead. One last question: Despite our best efforts to provide computational intuition by providing Agda code, is this work still too theoretical for TLDI? While it may remain too theoretical for some, we hope that we have made the case that the subject of induction recursion needs to be brought more firmly to the attention of at least a segment of the TLDI audience, and that this paper, using familiar ideas from the algebra of programming school, does as good a job as possible given the inherently technical nature of the subject. And, afterall, the value of a conference papers need not lie in its ability to say something to all participants, but may lie in its ability to say something significant to some of its participants.

## 2. Induction Recursion in A Nutshell

Dybjer and Setzer defined a system of codes for defining functors as follows:

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

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

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

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

Note that the constructor $\delta$ allows for the possibility that $A = 0$. Since $0 \to D = 1$, we have that if $F \in \mathsf{IR}\ D$, then ( by regarding an element of $\mathsf{IR}\ D$ as a function $1 \to \mathsf{IR}\ D$, we have that $\delta_0 F \in \mathsf{IR}\ D$.

Understanding IR codes, and thus each of the IR-constructors, is best done by understanding the semantics of each IR-code. This semantics was given by Dybjer and Setzer and associates to each

[1] This use of reflection is based upon the set-theoretic use of reflection rather than the categorical notion of reflection arising within the theory of adjoints.

IR code a functor over the category of families of the discrete category over $D$. This semantics is sometimes called the *decoding* of a code. First we define the families functor, and then we define the decoding function itself.

DEFINITION 2. *Let* $\mathcal{C} \in \mathsf{CAT}$ *be a large category. The large category* $\mathsf{Fam}(\mathcal{C}) \in \mathsf{CAT}$ *is defined as follows.*

- *The objects of* $\mathsf{Fam}(\mathcal{C})$ *are pairs* $(U : \mathsf{Set}, T : U \to |\mathcal{C}|)$. *Here,* $|\mathcal{C}|$ *refers to the objects of the category* $\mathcal{C}$. *Note here that the set* $U$ *is a* small *set. We call* $U$ *the index set of the family and* $T$ *the decoding function of the family.*

- *The morphisms of* $\mathsf{Fam}(\mathcal{C})$ *between* $(U, T)$ *and* $(U', T')$ *consist of pairs* $(g, g^*)$ *where* $g : U \to U'$ *is a function between the index sets of the respective families and* $g^*$ *is a function assigning to each* $u \in U$, *a morphism* $g^*_u : Tu \to T'(ga)$ *in* $\mathcal{C}$. *Equivalently,* $g$ *is a natural transformation between the functors* $T$ *and* $T' \circ g$.

When $I$ is a large set, we write $\mathsf{Fam}\ I$ for the action of $\mathsf{Fam}$ on the discrete category over $I$. Indeed, within the theory of induction recursion, we are typically interested in categories of the form $\mathsf{Fam}I$ where $I$ is a large set. We also will need a version of the families functor which is an endofunctor on $\mathsf{SET}$. This version of the families functor, which we also denote $\mathsf{Fam}$ takes as input a large set $I$ and returns the large set of objects in $\mathsf{Fam}\ I$. Which version of $\mathsf{Fam}$ is intended can be inferred from the context. The reader may ask why we need the variations of $\mathsf{Fam}$. The answer is that firstly $\mathsf{Fam}$ must return a category so that the decoding of an IR code can be a functor between categories of families, while the input must be a category so that $\mathsf{Fam}$ can form a monad. Secondly, the ability of induction recursion to define universes closed under mixed variant constructors such as $\Pi$-types only works when we decode IR codes as functors over $\mathsf{Fam}\ I$ where $I$ is a large set. And, finally, we need a version of $\mathsf{Fam}$ as an endofunctor over the category $\mathsf{SET}$ as for most of the paper, we will consider IR codes to form an initial algebra of an endofunctor which, thus, must be an endofunctor over $\mathsf{SET}$. For the moment, observe that if $I$ is a large set, then a morphism $(U, T) \to (U', T')$ in $\mathsf{Fam}\ I$ consists of a $\alpha : U \to U'$ such that $T = T' \circ \alpha$.

EXAMPLE 1. $\mathsf{Fam}\ 1$**:** *A degenerate case occurs when* $\mathcal{C} = 1$, *the one object category. In this case,* $\mathsf{Fam}\ 1$ *is isomorphic to the category* $\mathsf{Set}$ *of sets. As we shall see, this allows us to study functors* $F : \mathsf{Set} \to \mathsf{Set}$ *within induction recursion. More precisely, it will allow us to see containers as a special case of induction recursion.*

EXAMPLE 2. $\mathsf{Fam}(\mathsf{Set})$*: When* $\mathcal{C}$ *is the category* $\mathsf{Set}$, *the category* $\mathsf{Fam}(\mathsf{Set})$ *has as objects families of sets. This category is equivalent to the arrow category* $\mathsf{Set}^{\to}$ *and is widely used as a model of type theory as it is locally cartesian closed. See for example [10]*

EXAMPLE 3. $\mathsf{Fam}(\mathsf{Set}^{op})$**:** *When* $\mathcal{C}$ *is the category* $\mathsf{Set}^{op}$, *we get the category* $\mathsf{Cont}$ *of containers [4], [14], [2], [1]. This category has a full and faithful embedding* $[-] : \mathsf{Cont} \to [\mathsf{Set}, \mathsf{Set}]$ *defined by*

$$\begin{array}{rcl} [S, P] & : & \mathsf{Set} \to \mathsf{Set} \\ {}[S, P]X & = & \Sigma s : S.\ Ps \to X \end{array}$$

*See section 4 for more details.*

The next part of Dybjer and Setzer's work is to define a decoding function which assigns to every large set $D$ and IR code $c \in \mathsf{IR}\ D$ a functor $[\![c]\!] : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$. Dybjer and Setzer notice that any object $P$ of $\mathsf{Fam}\ D$, is given by an object $\pi_0 P : \mathsf{Set}$ and $\pi_1 P : \pi_0 P \to D$. Indeed $\pi_0 : \mathsf{Fam}\ D \to \mathsf{Set}$ is a functor. As a result of this decomposition, the object part of the functor $[\![c]\!]$ can be given in two parts: i) a function $F_0 : \mathsf{Fam}\ D \to \mathsf{Set}$; and ii) a function $F_1$ which assigns to each object $P$ of $\mathsf{Fam}\ D$ a function $F_0 P \to D$. The morphism part of $[\![c]\!]$ is given, of course, by mapping morphisms to morphisms.

DEFINITION 3 (Dybjer and Setzer's Decoding Function). *Let* $D$ *be a large set and* $c \in \mathsf{IR}\ D$. *Define the functor* $[\![c]\!] : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$ *as follows:*

- *When* $c = \iota d$,

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

- *When* $c = \sigma_A f$,

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

- *When* $c = \delta_A F$,

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

We note the special case $[\![\delta_0 F]\!] = [\![F]\!]$. Also, in the definition of decoding notice that various coproducts in $\mathsf{Set}$ are used. Crucially, these coproducts are small and hence exist. For example, the definition of $[\![\sigma_A f]\!]$ is a coproduct over elements of the small set $A$, while in the definition of $[\![\delta_A f]\!](U, T)$, the coproduct is over elements of $A \to U$. Since $A$ and $U$ are small sets, such elements also form a set.

Infact, Dybjer and Setzer proved that if $c$ is an IR code, then $[\![c]\!]$ not only maps families to families, but is also a functor. We give the action of $[\![c]\!]$ on morphisms below:

LEMMA 1. *Let* $D$ *be a large set and* $c \in \mathsf{IR}\ D$ *be an IR code. Then* $[\![c]\!]$ *defines a functor* $\mathsf{Fam}\ D \to \mathsf{Fam}\ D$.

PROOF 1. *Let* $(U, T)$ *and* $(U', T')$ *be objects of* $\mathsf{Fam}\ D$. *As remarked above, since* $D$ *is a (large) set, a morphism between* $(U, T)$ *and* $(U', T')$ *consists of an* $\alpha : U \to U'$ *such that* $T = T' \circ \alpha$. *Then* $[\![c]\!]\alpha : [\![c]\!](U, T) \to [\![c]\!](U', T')$ *can be defined by giving a function* $[\![c]\!]_0\alpha : [\![c]\!]_0(U, T) \to [\![c]\!]_0(U', T')$ *such that* $[\![c]\!]_1(U, T) = [\![c]\!]_1(U', T') \circ [\![c]\!]_0\alpha$. *Induction on the structure of* $c$ *can be used to define* $[\![c]\!]_0\alpha$ *as follows:*

- *If* $c$ *is* $\iota d$, *then* $[\![c]\!]_0(\alpha) = id$
- *If* $c$ *is* $\sigma(A\ ,\ f)$, *then* $[\![c]\!]_0\ \alpha\ (a, t) = (a,\ [\![fa]\!]_0\ \alpha\ t)$
- *If* $c$ *is* $\delta(A\ ,\ F)$, *then* $[\![c]\!]_0\ \alpha\ (g, t) = (\alpha \circ g,\ [\![F(T \circ g)]\!]_0\ \alpha\ t)$

Those functors in the image of $[\![-]\!]$ are given a special name:

DEFINITION 4 (IR Functors). *Let* $D$ *be a large set and* $F : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$ *be a functor. If there is a code* $c$ *such that* $F \cong [\![c]\!]$, *then we say that* $F$ *is an IR-functor.*

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 functor $[\![c]\!]$ has an initial algebra. As a result, induction recursion is a principle asserting the existence of various set-indexed families. It clearly goes beyond $W$-types as these are the fixed points of containers and (as we shall see in the next section on examples) all containers are IR-functors.

DEFINITION 5 (IR Datatypes). *A family* $T : U \to D$ *is inductive recursive iff there is an IR-code* $c$ *such that* $(U, T) \cong \mu[\![c]\!]$

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, and in particular its presentation via familiar and known categorical concepts, together with its implementation in Agda, will help to rectify this situation. But before we delve into our own results, we present some examples to show both the power, and the inscrutability, of induction recursion. note that examples 5, 7, 8 and 9 are original and hence part of the contribution of this paper. These examples get ever more sophisticated so, if examples aren't your thing, skip straight to section 3.

## 2.1 Examples of IR Datatypes

If $D = 1$, then $\mathsf{Fam}\ D$ is essentially the category $\mathsf{Set}$ of sets. Hence functors on $\mathsf{Set}$ can be discussed within the framework of induction recursion.

EXAMPLE 4 (Natural numbers). *The functor mapping $X$ to $X+1$ is the decoding of the $IR$-code*

$$\iota * + \delta_1(\lambda x.\, \iota *) : \mathsf{IR}\ 1$$

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

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

In the last example, we coded-up the inductively defined datatype $\mathsf{Nat}$ 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 [15]. One of the main reasons (stated in [13]) 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.

We now do something more general, for the so-called W-type of Martin-Löf type theory. This is its canonical mechanism for defining inductive data types. Given a family of sets $(S, P) : \mathsf{Fam\,Set}$, the type $W(S, P)$ represents the initial algebra of the container functor $[S, P]$. Its elements are wellfounded trees, in which the nodes are labelled by elements $s$ of $S$, and each such node is assigned a $P\ s$ indexed family of immediate subtrees.

EXAMPLE 5 (Container functors and W-types). *More generally, if $(S, P)$ is any container, the functor $[S, P] : \mathsf{Set} \to \mathsf{Set}$ is the decoding of the* $\mathsf{IR}$*-code*

$$\sigma_S(\lambda s.\, \delta_{Ps}(\lambda \_.\, \iota *)) : \mathsf{IR}\ 1$$

*Indeed, it is an easy inductive proof to see that the class of IR-functors when $D = 1$ is exactly the class of container-functors.*

*A simple embellishment of this code assigns to each element of a W-type the set of paths through it from the root to some subtree.*

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

Since we have shown that all container functors are IR-functors, we also have that all $W$-types are IR datatypes. We illustrate in a later example that induction recursion is a comparatively powerful principle, which allows us to define data structures more complex than those that can be defined using fixed points of containers, or even indexed containers.

Though it is powerful, it is also useful at an extremely humdrum level, for example to define a datatype such as a type of labelled binary trees, that we wish to make sure satisfy some invariant. By using induction recursion we can build the invariant into the type.

EXAMPLE 6 (Binary sorted trees). *Suppose we have a set $A$ of labels on which there is a decidable total order $(\leq) : A^2 \to 2$. Now we define a set $T$ of $A$-labelled binary trees in which the labels are non-decreasing as we traverse the tree in-order.*

$$\begin{array}{l|l}
\multicolumn{2}{c}{T = 1 + (\Sigma\, t_1 : T, a : A, t_2 : T)\, t_1 \leq a \wedge a \leq t_2} \\
\hline
\Diamond \leq a = \mathrm{true} & a \leq \Diamond = \mathrm{true} \\
t_1, a, t_2, \_, \_ \leq a' & a' \leq t_1, a, t_2, \_, \_ \\
\quad = a \leq a' \wedge t_2 \leq a' & \quad = a' \leq t_1 \wedge a' \leq a
\end{array}$$

*The structure has type* $\mathsf{Fam}((A \to 2)^2)$. *The code for the endofunctor is as follows.*

$$\begin{array}{ll}
 & \iota\, \langle\, \lambda \_.\, \mathrm{true}, \lambda \_.\, \mathrm{true}\, \rangle \\
+ & \sigma\, A\, (\lambda a. \\
 & \delta_1\, (\lambda \langle\, X, Y\, \rangle. \\
 & \iota\, \langle\, \lambda a'.\, a \leq a' \wedge X\, a', \lambda a'.\, Y\, a' \wedge a' \leq a\, \rangle)) \\
: & \mathsf{IR}((A \to 2)^2)
\end{array}$$

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 (Three kinds of closure under $\Sigma$). *Assume that we have a family $(S, E)$ :* $\mathsf{Fam\,Set}$. *We define three different indexed families that extend this given family (in the sense that there are indices for each set of the given family, and are moreover closed under $\Sigma$, each in a slightly different sense. These families are respectively the initial algebras of the following functors, that are endofunctors on* $\mathsf{Fam\,Set}$.

$$\begin{array}{lcl}
\begin{pmatrix} U \\ T \end{pmatrix} & \mapsto & \begin{pmatrix} S \\ E \end{pmatrix} + \begin{pmatrix} [U, T]U \\ \lambda(u, f).\, \Sigma\, (T\, u)(T \cdot f) \end{pmatrix} \\
\begin{pmatrix} U \\ T \end{pmatrix} & \mapsto & \begin{pmatrix} S \\ E \end{pmatrix} + \begin{pmatrix} [U, T]S \\ \lambda(u, s).\, \Sigma\, (T\, u)(E \cdot s) \end{pmatrix} \\
\begin{pmatrix} U \\ T \end{pmatrix} & \mapsto & \begin{pmatrix} S \\ E \end{pmatrix} + \begin{pmatrix} [S, E]U \\ \lambda(s, u).\, \Sigma\, (E\, s)(T \cdot u) \end{pmatrix}
\end{array}$$

*These functors are represented by the following IR codes, of type* $\mathsf{IR(Set)}$*).*

$$\begin{array}{lcl}
\sigma\iota\, (S, E) & + & \delta_1\, (\lambda U.\, \delta_U\, (\lambda T.\, \iota(\Sigma\, U\, T))) \\
\sigma\iota\, (S, E) & + & \delta_1\, (\lambda U.\, \sigma\iota\, (U \to S,\ \lambda s.\, \Sigma\, U\, (E \cdot s))) \\
\sigma\iota\, (S, E) & + & \sigma_S\, (\lambda s.\, \delta_{E\, s}\, (\lambda T.\, \iota(\Sigma\, (E\, s)\, T)))
\end{array}$$

*Note that the third of these codes is 'degenerate' (strongly fibred), in that though large-valued recursion is involved, there is no need to define the two parts of the family simultaneously.*

We have already seen examples of 'small' induction recursion, where the values of the recursively defined function are sets, or small types such as the set 2 of booleans, or the set $A \to 2$ of boolean predicates over a set. Small induction recursion can be extremely useful to obtain a desired structure on the inductively defined indices. This is illustrated in the following example.

EXAMPLE 8 (Consecutive transitions). *A* transition system *is a 4-tuple:*

$$\begin{array}{l}
S : \mathsf{Set}, \\
T : S \to \mathsf{Set}, \\
d : (s : S) \to T\, s \to S \\
s_0 : S
\end{array}$$

*Think of the elements of $S$ as states. Think of the pair of functions $(T, d)$ as (co-operatively) assigning to each $s : S$ the set-indexed family* $\mathsf{SET}d\,s\,tt : T\,s$ *of states immediately downstream from $s$, which means those that can be reached from $s$ by a single transition in $T\,s$. The last piece of data is an initial state $s_0$.*

*We may be interested to the family of states that are* reachable *from the initial state of a transition system via zero or more transitions. To define an index set for this family, we need to define the set of sequences (possibly empty) of consecutive transitions from the initial state. Simultaneously with this we define a function whose domain is the set of such composite transitions, that gives the state to which the system is brought by the last transition. The notion of a sequence of* consecutive *transitions has to be expressed in terms of this function. This mutual interdependency between a function and its domain, where we must think of both entities as built-up simultaneously, is characteristic of induction recursion.*

*The things we are defining, that I write $T_* :$* $\mathsf{Set}$ *and $d_* : T_* \to S$, are given as the two coordinates of the fixed point of the following endofunctor on* $\mathsf{Fam}\,S$.

$$\left(\begin{array}{c}\Gamma : \mathsf{Set} \\ f : \Gamma \to S\end{array}\right) \mapsto \left(\begin{array}{c}1 \\ \lambda\_.\, s_0\end{array}\right) + \left(\begin{array}{c}\Sigma\,\Gamma\,(T f) \\ \lambda(\gamma, t).\,(f\,\gamma)\,t\end{array}\right)$$

*This functor can be coded as follows:*

$$\iota\, s_0 \,+\, \delta_1\,(\lambda s.\,\sigma\iota\,(T\,s, d\,s)) : \quad \mathsf{IR}(S)$$

*I shall write the constructor for the empty sequence $\Diamond : T_*$, and write the binary constructor that corresponds to the second summand of the functor (where we prolong a given sequence by a further atomic transition) using an infix operator $\_\frown\_ : (ts : T_*) \to T(d_*\,ts) \to T_*$.*

*Now we put these things together to define the reachable part of a given transition system.*

$$\begin{array}{l} S' = T_* \\ T'\,\mathcal{I}ts = T\,(d_*\,\mathcal{I}ts) \\ d'\,\mathcal{I}ts\,t = (\mathcal{I}ts \frown t) \\ s_0' = \Diamond \end{array} \quad .$$

The following defines a set of well-founded structures by induction recursion of type $\mathsf{Fam}^2\,\mathsf{Set}$. It can be shown that this set is strictly more extensive than can be obtained by use of W-types alone, in other words by using induction recursion only at type $\mathsf{Fam}\,1$).

EXAMPLE 9 (An inaccessible). *An inaccessible will be a structure of type* $\mathsf{Fam}\,(\mathsf{Fam}\,\mathsf{Set})$. *Such an object is a triple*

$$\begin{array}{ll} \Omega : \mathsf{Set} & - \quad \textit{index set} \\ \left.\begin{array}{l} I : \Omega \to \mathsf{Set} \\ J : (\alpha : \Omega) \to I\,\alpha \to \mathsf{Set} \end{array}\right\} & \Omega \to \mathsf{Fam}\,\mathsf{Set} \end{array}$$

*Informally, the definition can be written*

$$\begin{array}{l|l} \multicolumn{2}{l}{\Omega = 1 + \Omega + (\Sigma\,\alpha : \Omega)\,([I\,\alpha, J\,\alpha]\Omega} \\ \hline I\,0 = \mathbf{0} & \_ \\ I\,\alpha^+ = I\,\alpha + 1 & J\,\alpha^+\,i = J\,\alpha\,i \\ & J\,\alpha^+\,\Diamond = W(I\,\alpha)(J\,\alpha) \\ I(\sqcup_{\alpha,i} f) = (\Sigma\,j : J\,\alpha\,i)\,I(f\,j) & J(\sqcup_{\alpha,i} f)\,(j, i') = J(f\,j)\,i' \end{array}$$

*where the equation above the line defines the index set inductively, while the columns of equations below the line define $I : \Omega \to$* $\mathsf{Set}$ *and $J : (\alpha : \Omega) \to I\,\alpha \to$* $\mathsf{Set}$ *by recursion. I use $0$, $\_^+$ and $\sqcup_{\_,\_}\_$ as constructors for the respective summands of the right-hand side in the inductive definition of $\Omega$.*

*To see what is going on, it is enlightening to figure out the first few families of sets that arise. The key is that these families of sets are used to define containers, and the W-types that are the initial algebras of these are in turn used to define exponential functors $X \mapsto X^E$.*

- *The family $I\,0, J\,0$ is the empty family $\{\,\}$. As a container, this is the constant functor with value $\mathbf{0}$, for which the initial algebra $W(I\,0, J\,0)$, is the empty set $\mathbf{0}$.*
- *The family $I\,0^+, J\,0^+$ is the singleton family $\{\,\mathbf{0}\,\}$. As a container, this is the constant functor with value $X^{\mathbf{0}} = 1$, for which the initial algebra $W(I\,0^+, J\,0^+)$ is the singleton set $1$. So far so boring.*
- *The family $I\,0^{++}, J\,0^{++}$ is $\{\,\mathbf{0}, 1\,\}$. As a container, its value at a set $X$ is value $X^{\mathbf{0}} + X^1$, and for this the initial algebra $W(I\,0^+, J\,0^+)$ is* $\mathsf{Nat}$.
- *The family $I\,0^{+++}, J\,0^{+++}$ is $\{\,\mathbf{0}, 1, \mathsf{Nat}\,\}$. As a container functor this takes $X$ to $1 + X + X^{\mathsf{Nat}}$, and for this the initial algebra is the set of countable Brouwer ordinals.*

*And so on. Successive family members (the values of $J$) are essentially set-level implementations of the successive regular ordinals (provided we allow that $\mathbf{0}$ and $1$ are, though finite, both regular). For each $\alpha : \Omega$, the elements of $I\alpha$ serve as indices for the predecessors of $\alpha$. The sequence of sets* $\mathsf{SET}J\,\beta\beta : I\alpha$ *exhaust the first $\alpha$ successive regular ordinals. By construction $J\,\alpha^+$ is the least common fixed point of the functors* $\mathsf{SET}X \mapsto X^{J\,\alpha\,\beta}\beta : I\,\alpha$. *The long and the short of it is that the set $\Omega$ is regular, yet closed under the step to the next regular, and hence an in a sense that can be made precise, inaccessible.*

*Finally, we write the code that represents the endofunctor on* $\mathsf{Fam}^2\mathsf{Set}$ *displayed informally in the table above. Taking the liberty of using pattern-matching abstraction, it can be written as the following code of type* $\mathsf{IR}\,(\mathsf{Fam}\,\mathsf{Set})$.

$$\left(\begin{array}{ll} & \iota(\mathbf{0}, \_) \\ + & \delta_1\,(\lambda(I, J).\,\iota(I + 1, [\,J\,|\,W\,I\,J\,])) \\ + & \delta_1\,(\lambda(I, J). \\ & \sigma\delta\,(I, J)\,(\lambda(i, (\lambda j : J\,i.\,(K\,j, L\,j))). \\ & \iota\,(\Sigma\,(J\,i)\,K, \lambda(j, k).\,L\,j\,k))) \end{array}\right)$$

Finally, an example of an operation on families which is not functorial, and hence not IR-encodable.

EXAMPLE 10. *The map sending a family $(U, T)$ to the family $(U + 1, [T, \lambda\_.\,U])$ does not extend to a functor on* $\mathsf{Fam}$ *and hence is not the decoding of an IR code.*

## 3. The Families Monad And Mixed Variant IR

**The Families Monad:** The presentation of the decoding function given above takes as input a code $c$ and a family $(U, T)$ and then 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. Raising the level of abstraction is well known to illuminate fundamental structure and category theory has proven itself well equipped for this task. Indeed, the ability to trade complex constructions in simple categories for simpler constructions in more sophisticated categories is of course a hall mark of category theory. As a concrete pay-off we will be able to remove the six clauses defining the decoding function and replace them by three. Not only will this increase tractability, this gives us the new possibility of high level reasoning by using the properties of the category $\mathsf{Fam}\ D$.

In this section we develop the properties of families we need and then use them to give a different presentation of induction recursion based upon this structure. One key concept we need is that the functor $\mathsf{Fam}$ is a strong monad. We begin by recalling this definition and then showing that $\mathsf{Fam}$ is indeed a strong monad. Note the definition of a strong monad relies on working with a monoidal category. Were we to spell out all the details regarding monoidal categories and strength, these details would not add much to the paper but would be a significant diversion from our main topic of interest. Hence we refer to [12] for more details regarding monoidal categories and strength should the reader wish to have complete definitions to hand.

DEFINITION 6. *Let $\mathcal{C}$ be a category. A monad on $\mathcal{C}$ consists of a functor $T : \mathcal{C} \to \mathcal{C}$ together with natural transformations called the unit $\eta : Id \to T$ and multiplication $\mu : T \circ T \to T$. These natural transformations are required to satisfy the associativity and unit laws: $\mu \circ \eta T = Id = \mu \circ T\eta$ and $\mu \circ T\mu = \mu \circ \mu T$.*

*A strong monad on a monoidal category $(\mathcal{C}, \otimes, I)$ is is a monad $(T, \eta, \mu)$ on $\mathcal{C}$ together with a natural transformation $\tau : TX \otimes Y \to T(X \otimes Y)$ satisfying the obvious laws.*

We note that in this paper we use $\mu$ for two operations: i) the initial algebra of a functor; and ii) the multiplication of a monad. While unfortunate, we have used this dual meaning since both are standard in the literature and the intended meaning of $\mu$ can be deduced from the context. We also note that the above definition of a monad is the "categorical" definition of a monad which is known to be equivalent to the functional programming definition based upon return and bind. The above reference contains more details of this equivalence.

LEMMA 2. $\mathsf{Fam}(-)$ *is a strong monad on* $\mathsf{CAT}$ *equipped with its cartesian structure.*

PROOF 2. *The unit $\eta_D : D \to \mathsf{Fam}\ D$ maps $d$ to the family $(1, \lambda\_.d)$. We note this process is functorial, ie that $\eta_D$ is a functor, and hence that $\eta$ is a natural transformation as required.*

*An element of $\mathsf{Fam}(\mathsf{Fam}\ D)$ consists of a set $A$ and a function $f : A \to \mathsf{Fam}\ D$ which, itself, consists of two parts: i) a function $f_0 : A \to \mathsf{Set}$; ii) and a function $f_1$ which takes an $a \in A$ as input and returns a function $f_1 a : f_0 a \to D$. The action of $\mu_D$ on such a family returns the family with index set $\Sigma a : A.f_1 a$ and with decoding function mapping the pair $(a, x)$ to $f_1 a x$. Again, $\mu_D$ is a functor and hence $\mu$ is a natural transformation as required.*

*Strength is given by a natural family of maps $\mathsf{Fam}\ D \times X \to \mathsf{Fam}(D \times X)$ which is defined by mapping a family $(A, f)$ and an $x \in X$ to the family $(A, \lambda a : A.\langle fa, x\rangle)$*

In Agda, we can represent the families monad as follows.

$$
\begin{array}{l}
\mathsf{Fam} : \mathsf{Set}_1 \to \mathsf{Set}_1 \\
\mathsf{Fam} D = \Sigma\ \ \mathsf{Set}\ \ (\lambda A \to (A \to D)) \\[4pt]
\mathsf{Fam}_1 : \{D\ \ D' : \mathsf{Set}_1\} \to (D \to D') \to \mathsf{Fam} D \to \mathsf{Fam} D' \\
\mathsf{Fam}_1\ \ g\ \ (A\ \ ,\ \ \ f) = (A\ \ ,\ \ g \circ f) \\[4pt]
\eta : \{D : \mathsf{Set}_1\} \to D \to \mathsf{Fam} D \\
\eta\ \ d = (\top\ \ ,\ \ \mathsf{const}\ d) \\[4pt]
\mu : \{D : \mathsf{Set}_1\} \to \mathsf{Fam}(\mathsf{Fam} D) \to \mathsf{Fam} D \\
\mu\{D\}(A\ \ ,\ \ B) = (\Sigma\ \ A\ \ (\mathsf{proj}_1 \circ B)\ \ ,\ \ g) \\
\qquad\qquad\qquad\quad \mathsf{where}\ \ g : \Sigma\ \ A\ \ \ (\mathsf{proj}_1 \circ B) \to D \\
\qquad\qquad\qquad\qquad\qquad g\ \ (a\ \ ,\ \ k) = \mathsf{proj}_2\ \ (Ba)\ \ k
\end{array}
$$

We chose not to implement strength in Agda as we need only one consequence of strength in our work, namely the following function

$$
\begin{array}{l}
ev : \mathsf{Fam} I \to \mathsf{Fam}(\mathsf{Fam} I \to \mathsf{Fam} O) \to \mathsf{Fam}(\mathsf{Fam} O) \\
ev\ P\ (A, f) = (A, \lambda a : A.\ f\ a\ P)
\end{array}
$$

We have seen that families are important in induction recursion as IR codes produce functors between categories of families. Further, the input to the $\sigma$-constructor is in fact a family of IR codes. But what about the $\delta$-constructor. If we think about $\sigma$-constructor as taking as input a code parameterised over elements of a set $A$, then (when $A = 1$ and $D = \mathsf{Set}$) the $\delta$-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 polymorphic function which is a function parameterised by sets themselves. Similarly, one can thus see $\sigma$ as building codes from $A$-indexed codes for a specific set $A$, while $\delta$-builds codes from type-indexed codes. To model this we introduce the notion of a large family so called because such a large family existentially quantifies over Set rather than a specific set

DEFINITION 7. *The large families functor* $\mathsf{LFam} : \mathsf{SET}^{op} \times \mathsf{SET} \to \mathsf{SET}$ *is defined by* $\mathsf{LFam}\ \ I\ \ O\ \ =\ \ \Sigma A : \mathsf{Set}.(A \to I) \to O$

In Agda, one may represent large families by

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

**Mixed Variant IR:** Having discussed families for a while, we turn our attention to variance. Notice how large families have a naturally contravariant part and a naturally covariant part. This makes us wonder about IR codes themselves - the reader may have asked whether IR is functorial. That is, does a map $f : D \to D'$ induce a map $\mathsf{IR}\ f : \mathsf{IR}\ D \to \mathsf{IR}\ D'$. The answer is clearly no as the construction of $\mathsf{IR}\ D$ is both covariant and contravariant in $D$. However, we can tease out the covariance and contravariance if we notice that all the covariance comes from the $\iota$ constructor, while all the contravariance comes from the $\delta$-constructor. The constructor $\sigma$ is variance neutral.

Thus we arrive at our first reformulation of IR codes as the following Agda definition

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

This makes it clear that the type $\mathsf{IR}\ \ I\ \ O$ is the initial algebra of the functor $O + \mathsf{Fam} + \mathsf{LFam}\ \ I : \mathsf{SET} \to \mathsf{SET}$. Notice that since IR codes form a large set, this dictates the requirement that it be the initial algebra of a functor on $\mathsf{SET}$ which, in turn, motivates the need to be able to consider Fam as a functor on $\mathsf{SET}$ as motivated briefly in section 2. An equivalent universal property for $\mathsf{IR}\ I\ O$ is that it is the free monad on the functor $\mathsf{Fam} + \mathsf{LFam}\ I$ at $O$. This is the most fundamental pay off for mixed variant IR - by separating out the variance in IR's constructors, we get a universal property characterising the large set of IR-codes.

The reader however should be nervous ... what exactly is the category $\mathsf{SET}$. Its certainly not the usual place we look for semantics and only the foolish would tread blindly into the world of large cardinals, ignoring subtle issues such as local smallness, regularity and inaccessibility without caution. Certainly, things *should* work out, but *should* is not as firm a basis for certainty as we ideally want. Nor should the fact that these definitions type check in Agda fool us into believing these foundations are as secure as we would like them to be. As we shall see in section 6, the role of internal IR will be to exactly resolve this problem by brining us back down to functors and monads over the category Set of small sets.

**Reflection:** What is left to do is give our decoding function for mixed variant IR codes. As the codes $\mathsf{IR}\ \ I\ \ O$ are parameterised by a negatively occurring $I$ and a positively occurring $O$, the reader will not be surprised to discover that in mixed variant IR, each code decodes to a functor $\mathsf{Fam}\ \ I \ \rightarrow\ \mathsf{Fam}\ \ O$. Before we define the decoding function, we will use a new idea to help understand the essence of the decoding function for $\delta$.

DEFINITION 8. *Let* $F : \mathsf{SET}^{op} \times \mathsf{SET} \rightarrow \mathsf{SET}$ *be functor. A reflection for* $F$ *consists of a family of maps* $\rho : F\ \ I\ \ O \rightarrow \mathsf{Fam}\ \ I \rightarrow \mathsf{Fam}\ \ O$ *natural in* $O$*. Note that in this definition* $\mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O$ *should be regarded as the large* set *of functors between the* category $\mathsf{Fam}\ I$ *and* $\mathsf{Fam}\ O$ *where* $I$ *and* $O$ *are large* sets.

A categorically astute reader may wonder why no naturality in $I$ is required by a reflection. The simple answer is we never need any form of naturality in $I$ as $I$ always seems to be fixed in our constructions. Were it to vary, one could of course ask for naturality in $I$. However, the reader may then wonder whether dinaturality would be better deployed if some form of naturality in both $I$ and $O$ were required. Unfortunately this would not seem sufficient as we definitely need maps $F\ \ I\ \ O \rightarrow \mathsf{Fam}\ \ I \rightarrow \mathsf{Fam}\ \ O$ when $I \neq O$.

A reflection for $\mathsf{LFam}$ is given in Agda by defining

$$\begin{aligned}&\rho : \mathsf{LFam}\ I\ O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O\\ &\rho\ (A\ ,\ F)\ (U\ ,\ T) = (A \rightarrow U\ ,\ \lambda g \rightarrow F(T \circ g))\end{aligned}$$

This definition clearly shows that something like reflection is at the heart of the decoding function for $\delta$-codes and, indeed, we can use reflection to recast the decoding function of the $\delta$-constructor. Overall, mixing the ideas of the families monad, mixed variant IR and reflection, we derive the following Agda decoding function for IR codes.

$$\begin{aligned}&[\![-]\!] : \{I\ O : \mathsf{Set}_1\} \rightarrow \mathsf{IR}\ I\ O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O\\ &[\![\ \iota\ d\ ]\!]\ P\ = \eta\ d\\ &[\![\ \sigma\ f\ ]\!]\ P\ = (\mu \circ \mathsf{Fam}_1(\mathsf{ev}\ P \circ [\![-]\!]))\ f\\ &[\![\ \delta\ F\ ]\!]\ P\ = (\mu \circ \mathsf{Fam}_1(\mathsf{ev}\ P \circ [\![-]\!]))\ (\rho\ F\ P)\end{aligned}$$

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. The decoding functions for $\sigma$ and $\delta$-constructors look very similar. Essentially, reflection is being used in the above decoding function to turn a large family of codes into a small family of codes whose semantics can then be given using the semantics for the $\sigma$-constructor. However, there is much more to be said about reflection if the reader will bear with us a little bit longer.

What about functoriality of $[\![c]\!]$ - can this too be given at the level of families. The answer is yes, and the proof is even more immediate with our algebraic presentation of the decoding function.

LEMMA 3. *Let* $c : \mathsf{IR}\ \ I\ \ O$ *be an IR code. Then* $[\![c]\!] : \mathsf{Fam}\ \ I \rightarrow \mathsf{Fam}\ O$ *is a functor*

PROOF 3. *All constructions in the definition of* $[\![-]\!]$*, e.g.* $\mu$*,* $\mathsf{Fam}$*,* $\mathsf{ev}$ *and* $\rho$ *are functorial and so act on morphisms.*

Given an IR code, we have seen how to generate a functor. When $I = O$ we get an endofunctor and, as we mentioned before, its fixed point is called an inductive recursive type. In Agda, we can define

$$\begin{aligned}&\mathsf{mutual}\\ &\quad \mathsf{data}\ U\ \{D : \mathsf{Set}_1\}\ (c : \mathsf{IR}\ D\ D)\ :\ \mathsf{Set\ where}\\ &\quad\quad C : (\mathsf{proj}_1 \circ [\![c]\!])(F\ c) \rightarrow U\ c\\ &\quad T : \{D : \mathsf{Set}_1\} \rightarrow (c : \mathsf{IR}\ D\ D) \rightarrow U\ c \rightarrow D\\ &\quad T\ c\ (C\ a) = (\mathsf{proj}_2 \circ [\![c]\!])\ (F\ c)\ a\\ &\quad F : \{D : \mathsf{Set}_1\} \rightarrow (c : \mathsf{IR}\ D\ D) \rightarrow \mathsf{Fam} D\\ &\quad F\ c = (U\ c\ ,\ T\ c)\end{aligned}$$

Of course, using mixed variant IR has the advantage that we could define a more general fixed point operator $\mathsf{IR}\ \ (I + O)\ \ O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O$ 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.

LEMMA 4. $\mathsf{IR}$ *is a mixed variant functor* $\mathsf{SET}^{op} \times \mathsf{SET} \rightarrow \mathsf{SET}$.

PROOF 4. *Our sketch proof consists of the action of on morphisms which we present in Agda*

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

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

## 4. Container IR

Having three constructors is a pain. And whats more, when the semantics of $\sigma$ and $\delta$ is presented using reflection, these two constructors seem very similar. Such unity is crying out to be formalised! To do this, we note that we could regard Fam as a mixed variant functor which is constant in its contravariant position. As such we could define a reflection for it by

$$\begin{aligned}&\rho : \mathsf{Fam}O \rightarrow \mathsf{Fam}\ I \rightarrow \mathsf{Fam}\ O\\ &\rho\ (A\ ,\ F)\ (U\ ,\ T) = (A\ ,\ F)\end{aligned}$$

then the semantics of both the $\sigma$- and $\delta$-constructors are exactly the same, namely the semantics of both $\sigma(A, F)$ and $\delta(A, F)$ maps $P$ to

$$(\mu \circ \mathsf{Fam}_1(\mathsf{ev}\ P \circ [\![-]\!]))\ (\rho\ F\ P)$$

where $\rho$ is the reflection for $\mathsf{Fam}$ and $\mathsf{LFam}$-respectively. This leads us to wonder if there is a common pattern underlying both $\sigma$ and $\delta$ ... 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 $\sigma$ and $\delta$ can naturally be compressed into one constructor which leads to a containerification of the meta-theory of IR.

In this section we develop container IR and, as an application, use it to define morphisms between IR codes. Our eventual goal is to replicate the fundamental theorem of containers which says that there is a category of containers which has a full and faithful embedding into a category of endofunctors. Container based IR allows us to do this, although the full result requires a paper in its own right. Here, we simply sketch the beginning of the process so as to demonstrate container IR at work.

Recall from section 2 that a container is a set $S$ and a function $P : S \to \mathsf{Set}$. In Agda we may define

$$\begin{array}{l}\mathsf{Cont} : \mathsf{Set}_1 \\ \mathsf{Cont} = \mathsf{Fam}\ \mathsf{Set}\end{array}$$

As mentioned there, containers define functors on the category Set. They also, using the same formula, extend to functors on SET as the following Agda definition shows

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

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

There are a number of constructions on containers which reflect constructions on endofunctors. For example, there is an identity container, constant containers, hom-functors, sums of containers, composition of containers. Once more, in Agda

$$\begin{array}{l}\mathsf{Id} : \mathsf{Cont} \\ \mathsf{Id} = (\top\ ,\ \mathsf{const}\ \top)\end{array}$$

$$\begin{array}{l}\_\oplus\_ : \mathsf{Cont} \to \mathsf{Cont} \to \mathsf{Cont} \\ (S\ ,\ P) \oplus (S'\ ,\ P') = (S + S'\ ,\ [P\ ,\ P'])\end{array}$$

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

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

$$\begin{array}{l}\_\odot\_ : \mathsf{Cont} \to \mathsf{Cont} \to \mathsf{Cont} \\ (S\ ,\ P) \odot (S'\ ,\ P') = ([(S\ ,\ P)]_0 S'\ ,\ g) \\ \qquad\qquad \mathsf{where}\ g : [(S\ ,\ P)]_0 S' \to \mathsf{Set} \\ \qquad\qquad\qquad g\ (s\ ,\ f) = \Sigma\ (Ps)\ (P' \circ f)\end{array}$$

Its easy to check that Id represents the identity functor, $\oplus$ represents the coproduct of functors, $KA$ represents the constantly $A$-valued functor, $RA$ represents the functor mapping $X$ to $A \to X$. Crucially, the embedding of containers as endofunctors maps $\odot$ to the composition of functors. That is, if $H$ and $G$ are containers, then the functors $[H \odot G]$ and $[H] \circ [G]$ are naturally isomorphic. One half of this isomorphism will be needed later and so we give it now in Agda

$$\begin{array}{l}\approx :\ \{H\ K : \mathsf{Cont}\} \to \{X : \mathsf{Set}_1\} \to \\ \quad [H \odot K]_0 X \to [H]_0([K]_0 X) \\ \approx\ \{(S\ ,\ P)\}\ \{(S'\ ,\ P')\}\ ((s\ ,\ f)\ ,\ k) = (s\ ,\ \langle f, \mathsf{curry}\ k\rangle)\end{array}$$

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 $\sigma$ and $\delta$ 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 LCFam functor 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, the functor LCFam is contravariant in its first argument and covariant in its second argument. Note that Fam and LFam are special cases where the chosen container is taken to be firstly a container of $K\ A$ which represents the constantly $A$ valued functor, and secondly $R\ A$ which represents the functor $A \to -$. Thus the container based IR codes have the same expressive power as Dybjer and Setzer's IR codes and container IR simply presents them differently. With these definitions in place, container induction recursion has the following codes

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

As a result, we see that $\mathsf{IR}\ \ I\ \ O$ is the free monad on $\mathsf{LCFam}\ \ I$ at $O$. However the remarks of the previous section on size issues remain valid and caution should be used when making claims such as those above. To give the decoding function, we need a reflection for LCFam and this can be given in Agda by

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

Given this reflection, we can define the decoding function for container IR codes as follows:

$$\begin{array}{l}[\![-]\!] : \{I\ O : \mathsf{Set}_1\} \to \mathsf{IR}\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O \\ [\![\ \iota\ d\ ]\!]\ T\ \ = \eta\ d \\ [\![\ \sigma\delta\ F\ ]\!]\ T\ \ = (\mu \circ \mathsf{Fam}_1(\mathsf{ev}\ T \circ [\![-]\!]))\ (\rho\ F\ T)\end{array}$$

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 effecting other parts. Of course, the reader may want to know concretely what this decoding formula does (and we shall need it later) so we give that now

LEMMA 5. *Let* $\sigma\delta(H\ ,\ F) \in \mathsf{IR}\ I\ O$ *and* $(U,T) \in \mathsf{Fam}\ I$. *Then*

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

PROOF 5. *Direct calculation*

As remarked above, the presence of separate $\sigma$ and $\delta$ 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 and use them to help us understand induction recursion. An example of this occurs in the definition of container morphisms. But first, we remark that one can go beyond Thorsten's remark in that there is nothing special about containers. Let $\mathcal{K}$ be any class of functors $\mathsf{SET} \to \mathsf{SET}$. Then we could define

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

and give this constructor an associated semantics via a decoding function. The natural questions to ask here regard the nature of the transformation of a class of functors $\mathcal{K}$ into the class of functors 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.

## 4.1 IR Morphisms via Container IR

The goal of this section is to define, for each pair of IR codes $c, c' \in \mathsf{IR}\ \ I\ \ O$ a set $\mathsf{IR}(c,c')$ and a decoding function mapping each $f \in \mathsf{IR}(c,c')$ to a natural transformation $[\![c]\!] \to [\![c']\!]$. We will do this from the ground up and so begin by showing that the functor $P\ +\ - : \mathsf{Fam}\ I \to \mathsf{Fam}\ I$ which sends a family $Q$ to $P + Q$ is inductive recursive.

LEMMA 6. *Let P be an object of* $\mathsf{Fam}\ I$. *Then* $P + - : \mathsf{Fam}\ I \to \mathsf{Fam}\ I$ *is inductive recursive*

PROOF 6. *Lets define the function*

$$\begin{aligned}
&\triangle : \{I : \mathsf{Set}_1\} \to \mathsf{Fam}\ I \to \mathsf{LCFam}\ I\ I \\
&\triangle\{I\}(X\ ,\ T) = (KX \oplus \mathsf{Id}\ ,\ g) \\
&\qquad\qquad \text{where } g : [KX \oplus Id]_0\ I \to I \\
&\qquad\qquad\qquad g(\mathsf{inj}_1 x\ ,\ f) = T\ x \\
&\qquad\qquad\qquad g(\mathsf{inj}_2 x\ ,\ f) = f\ x
\end{aligned}$$

*We claim that if $P \in \mathsf{Fam}\ I$ is of the form $B : A \to I$ , and if $\triangle P$ is the pair $(H\ ,\ \phi)$, then the IR code $\sigma\delta(H\ ,\ \iota \circ \phi)$ decodes to the functor $P\ +\ -$. This is established by direct calculation*

$$\begin{aligned}
[\![\sigma\delta(H\ ,\ \iota \circ \phi)]\!](U,T) &= \Sigma x : [H]_0 U . [\![(\iota \circ \phi)([H]_1 Tx)]\!](U,T) \\
&= \Sigma x : A + U.\ \eta(\phi[H]_1 Tx) \\
&= \Sigma a : A.\ \eta(Ba) + \Sigma u : U.\ \eta(Tu) \\
&= (A, B) + (U, T)
\end{aligned}$$

Now that we know that coproduct with a fixed family is an inductive recursive functor, we wish to extend this result to show that if we compose an inductive recursive functor with a functor of the form $P+-$, we still get an inductive recursive functor. The key definition is the following

$$\begin{aligned}
&\mathsf{comp} : \{I\ O : \mathsf{Set}_1\} \to \mathsf{IR}\ I\ O \to \mathsf{LCFam}\ I\ I \to \mathsf{IR}\ I\ O \\
&\mathsf{comp}\ (\iota\ P)\ F =\ \iota\ P \\
&\mathsf{comp}\{I\}\{O\}\ (\sigma\delta(H\ ,\ F))\ (K\ ,\ \phi) = \\
&\qquad \sigma\delta(H \odot K\ ,\ g) \\
&\qquad \text{where } g : [H \odot K]_0 I \to \mathsf{IR}\ I\ O \\
&\qquad\qquad g\ t = \mathsf{comp}\ ((F \circ [H]_0 \phi \circ \approx)t)\ (K\ ,\ \phi)
\end{aligned}$$

We can now prove the main property of comp.

LEMMA 7. *If c is an IR code and $(K, \phi) \in \mathsf{LCFam}\ I\ I$, then*

$$[\![\mathsf{comp}\ c\ (K\ ,\ \phi)]\!]\ (U,T) = [\![c]\!]([\![\sigma\delta(K, \iota \circ \phi)]\!](U,T))$$

PROOF 7. *Let $(U', T') = [\![\sigma\delta(K, \iota\circ\phi)]\!](U,T)$. Direct calculation shows that $U' = [K]_0 U$ and that $T' = \phi \circ [K]_1 T$. Now, when c is an $\iota$-code the lemma is trivial. When c is of the form $\sigma\delta(H\ ,\ F)$, then $[\![c\ ]\!](U', T')$ is equal to each of the following:*

$$\begin{aligned}
&\Sigma\alpha : [H]_0 U'.\ [\![F([H]_1 T'\alpha)]\!](U', T') \\
&= \Sigma\alpha : [H]_0([K]_0 U).\ [\![F([H]_1(\phi \circ [K]_1 T)\alpha)]\!](U', T') \\
&= \Sigma\alpha : [H]_0([K]_0 U).\ [\![(F \circ [H]_1\phi)([H]_1[K]_1 T\alpha)]\!](U', T') \\
&= \Sigma\alpha : [HK]_0 U.\ [\![(F \circ [H]_1\phi)([H]_1[K]_1 T(\approx \alpha))]\!](U', T') \\
&= \Sigma\alpha : [HK]_0 U. \\
&\qquad [\![\mathsf{comp}(F \circ [H]_1\phi)([H]_1[K]_1 T(\approx \alpha))(K, \phi)]\!](U,T) \\
&= [\![\mathsf{comp}\ c\ (K, \phi)]\!]\ (U,T)
\end{aligned}$$

We can now prove the main result of this section: namely, we define IR morphisms and show they decode to natural transformations. To do this, note that if $(S, P)$ is a container and $X\ :\ [S, P]_0 I$, then the second projection of $X$ is an $I$-indexed family - that is $\pi_1 X : \mathsf{Fam}\ I$

DEFINITION 9 (IR-Morphisms). *Define inductive recursive morphisms as follows.*

- *For $\sigma\delta$ codes, define $\mathsf{IR}(\sigma\delta((S,P),F)\ ,\ K)$ to be*

$$\Pi(X : [S,P]_0 I).\ \ \mathsf{IR}(FX, K(\pi_1 X + -))$$

- *For $\iota$-codes define $\mathsf{IR}(\iota d\ ,\ \sigma\delta((S,P),F))$ to be*

$$\Sigma(s,f) : [S,P]0.\ \mathsf{IR}(\iota d\ ,\ F(s, ! \circ f))$$

*and $\mathsf{IR}(\iota d\ ,\ \iota d')$ to be 1 if $d == d'$ and 0 otherwise.*

Note that the first clause of the definition of IR-morphisms is well defined as by the previous two lemmas $K(\pi_1 X + -)$ is an inductive recursive functor. We can now prove that IR-morphisms do indeed generate natural transformations.

LEMMA 8. *Let $\phi \in \mathsf{IR}(c, c')$. Then $\phi$ generates a natural transformation from $[\![c]\!]$ to $[\![c']\!]$*

PROOF 8. *The $\iota$-cases are easy, so we concentrate on the case where $\phi \in \mathsf{IR}(\sigma\delta_{S,P}F, K)$, that is*

$$\phi : \Pi(X : [S,P]I).\mathsf{IR}(FX\ ,\ K(\pi_1 X + -))$$

*By induction, for all $(s\ :\ S\ ,\ Q\ :\ Ps \to I)$, $\phi_{s,Q}$ generates a natural transformation and so we get maps*

$$\begin{aligned}
&\forall s : S, Q : Ps \to I, T : U \to I. \\
&\qquad [\![F(s,Q)]\!]T \to [\![K]\!](Q+T)
\end{aligned}$$

*Its easy to see that from such maps, we can derive a map of type*

$$\begin{aligned}
&\forall s : S, Q : Ps \to I, T : U \to I. \\
&\qquad [\![F(s,Q)]\!]T \otimes (\mathsf{Fam}\ I)(Q,T) \to [\![K]\!]T
\end{aligned}$$

*where if $R\ \in\ \mathsf{Fam}\ \ I$ and $A$ is a set, then $R \otimes A$ is the A-fold coproduct of R which exists as $\mathsf{Fam}\ I$ is closed under $\mathsf{Set}$-indexed coproducts. These maps allow us to define a map of type*

$$\forall s : S, g : Ps \to U, T : U \to I.\ [\![F(s, T \circ g)]\!]T \to [\![K]\!]T$$

*which is exactly a natural transformation as required.*

Finally, we can make good on an earlier promise. Not only are IR 1 1 codes equi-expressive with containers, but the category of containers is equivalent to the category of IR 1 1 codes and morphisms.

LEMMA 9. *When $I\ =\ 0\ =\ 1$, not only do IR codes agree with containers, but IR morphisms are exactly the container morphisms.*

PROOF 9. *We know that a container $(S, P)$ can be represented as an IR functor $\sigma\delta((S,P), \mathsf{const}(\iota *))$. The IR-morphisms between $\sigma\delta((S,P), \mathsf{const}(\iota *))$ and $\sigma\delta((S',P'), \mathsf{const}(\iota *))$ are thus given by*

$$\begin{aligned}
\Pi s : S.\ \Sigma g : [S', P'](Ps).1\ &=\ \Pi s : S.\ \Sigma s' : S'.\ P's' \to Ps \\
&= \Sigma f : S \to S'.\ \Pi s : S.\ P'(fs) \to Ps
\end{aligned}$$

*which are exactly those appearing in the literature.*

## 5. Reflective Induction Recursion

In section 3 we replaced the decoding function given by Setzer and Dybjer with one based upon the notion of reflection. In particular, we defined a reflection for a functor $F\ :\ \mathsf{SET}^{op} \times \mathsf{SET}\ \to\ \mathsf{SET}$ to be a family of maps $\rho_{I\ O}\ :\ F\ \ I\ \ O\ \to\ \mathsf{Fam}\ \ I\ \to\ \mathsf{Fam}\ \ O$ natural in $O$. By giving a reflection for the functor LFam, we were able to rewrite the decoding function for IR codes built from the $\delta$ constructor in a manner similar to that for the $\sigma$ constructor. In section 4 we went further and showed how, by giving a reflection for Fam, the semantics of both $\sigma$ and $\delta$ are essentially the same, varying only in the choice of reflection. Once we had introduced the $\sigma\delta$-constructor, we gave a reflection for the functor LCFam and used it to define the same decoding function for container based IR codes as for non-container IR apart from the different choice of reflection. These reflections do seem to therefore be at the heart of the decoding function for induction recursion. We now give some more examples of reflections:

EXAMPLE 11. *The mixed variant functor* $\mathsf{IR}\ I\ O$ *has a reflection given by its decoding function* $[\![-]\!] : \mathsf{IR}\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O$ *is a reflection*

As an extreme case, we have the following

EXAMPLE 12. *The mixed variant functor mapping $I$ and $O$ to* $\mathsf{Fam}\ I \to \mathsf{Fam}\ O$ *has the identity as a reflection.*

And an obvious mixed variant functor is the hom-functor:

EXAMPLE 13. *The mixed variant functor mapping $I$ and $O$ to $I \to O$ has as a reflection the map sending a function $f : I \to O$ and the family $(U, T) \in \mathsf{Fam}\ I$ to the family $(U, f \circ T)$. This is of course just the action of the functor* Fam *on morphisms.*

So the three reflections appearing in the previous sections, together with the three above, suggest that not only are reflections key to the decoding function for induction recursion, but that moreover they are a natural concept which can be expected to arise widely. The reader may wonder whether a reflection is all that we need to define IR codes and a decoding function. Perhaps surprisingly the answer is yes. Indeed, we can parameterise an Agda module by a functor representing the constructors of induction recursion and a reflection from which the decoding function may be generated. This can be done via the declaration

$\mathsf{open\ \ import\ \ Fam1}$

$\mathsf{module\ \ RefIR}$
$(F : \mathsf{Set}_1 \to \mathsf{Set}_1 \to \mathsf{Set}_1)$
$(F_1 : \{Z\ X\ Y : \mathsf{Set}_1\} \to (X \to Y) \to F\ Z\ X \to F\ Z\ Y)$
$(R : \forall\{I\ O : \mathsf{Set}_1\} \to F\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O)$
$\mathsf{where}$

and then build our IR codes and their decoding function over these parameters. Given the last two sections, it should not be surprising that the IR-codes $\mathsf{IR}\ I\ O$ relative to these parameters are the free monad on $F\ I$ at $O$ as can be given in Agda by

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

while the decode function for these codes uses the reflection parameter just as the specific instances previously have

$$\begin{array}{l}[\![-]\!] : \{I\ O : \mathsf{Set}_1\} \to \mathsf{IR}\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O\\ [\![\ \iota\ d\ ]\!]\ P\ = \eta\ d\\ [\![\ \sigma\delta\ F\ ]\!]\ P\ = (\mu \circ \mathsf{Fam}_1(\mathsf{ev}\ P \circ [\![-]\!]))\ (\rho\ F\ P)\end{array}$$

So, this shows us that given any functor, we can derive codes via the free monad construction and a decoding function via the reflection for the functor. Moreover, this suggests to us that reflection is at the heart of induction recursion. We believe this not just because reflection is all that is needed to define codes and their decodes, but also because reflection contains at its core a reduction of size from big to small, a feature which is central at a conceptual level to induction recursion. Indeed, Dybjer and Setzer's first presentation of induction recursion was as a reflection principle as found in set theory. Roughly speaking, within that setting, a reflection principle is something that turns large things into small things. For example, a function $\mathsf{Set}_1 \to \mathsf{Set}$ in Agda could be thought of as reflection principle as it will have to turn large sets into small sets. Similarly a natural transformation $\mathsf{LFam}\ I\ O \to \mathsf{Fam} O$ could be a thought of as a reflection principle as it would turn a large family into a family indexed by a *specific, small* set.

Other examples of such reflections are given by parametricity as found in System F which can be used to show that types quantified over other types impredicatively are equivalent to small types - eg $\Pi X.X \approx O$. From category theory, one can see the Yoneda lemma as a reflection as it says the large set of natural transformations between the hom functor $\mathcal{C}(X, -)$ and a functor $F$ is equivalent to the small set $FX$. Finally representation theorems such as the container representation theorem which classifies natural transformations between containers as a small set, and the "eating" program of classifying continuous functions between final coalgebras as a small set are also reflections in this sense. Of course, in general, reflections with good properties are hard to come up with precisely because it is hard to turn large things into small things. But their importance is hard to ignore ... indeed one could argue that much of the work of the computer scientist lies in the search for reflections which convert large semantic objects into small objects which can be represented with a computer program and hence be made amenable to computational processes.

Now consider the reflections $\rho : F\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O$ which concern us. Given a family $P \in \mathsf{Fam}\ I$, we can define $\rho\ \_\ P : F\ I\ O \to \mathsf{Fam}\ O$ which can turn a large object as represented by $F\ I\ O$ into a small family indexed by a set. This is exactly why the semantics of $\sigma$ and $\delta$ in section 3 were so similar - reflection turned a large-indexed family of IR-codes into a small indexed family of IR-codes which could be handled by $\sigma$. Another way of looking at $\rho$ is to say that it turns the largeness inherent in $F$, into largeness of the functor-space between small-indexed families. And a final way to consider the use of reflection inherent in induction recursion is to provide the capacity to generate from a family $P$ a function which turns a large object into a small object localised at $P$.

We finish this section with an open question. We have seen how induction recursion can be seen as taking as input a functor $F : \mathsf{SET}^{op} \times \mathsf{SET} \to \mathsf{SET}$ and a reflection $\rho$ and returns the functor $\mathsf{IR}\ I\ O$ with decode function $[\![-]\!]$. We have seen how IR is a mixed variant functor in lemma 4. This makes $[\![-]\!]$ a reflection for IR and so one may play the same process again and again. That is, one may ask for the fixed point of the operation sending the pair $(F\ ,\ \rho)$ to $(\mathsf{IR},\ [\![-]\!])$. More concretely, one may wonder whether building IR codes over the reflection $(\mathsf{IR}\ ,\ [\![-]\!])$ gives us any more definable functors than those arising from the IR codes $\mathsf{IR}\ I\ O$ themselves. Note this is a different question from asking for the relationship between $\mathsf{IR}\ I\ (\mathsf{IR}\ I\ O)$ and $\mathsf{IR}\ I\ O$ - this later relationship is probably best understood by noting that

$$\mathsf{IR}\ I\ (\mathsf{IR}\ I\ O) = (F\ I)^*(FI)^*0 \to (F\ I)^*O = \mathsf{IR}\ I\ O$$

where the middle arrow is the multiplication of the monad $(F\ I)^*$.

## 6. Internal IR

So far we have been happy working, with the odd warning, with large categories, large functors, large this and large that. While we have not been deterred by the inherent dangers of this activity, we have noted that caution is surely required. As another example of the potential pitfalls we are in danger of falling into, consider the principle of doing induction on the structure of IR codes as is found in both the original works of Dybjer and Setzer when defining the decoding function, and in our work when defining IR morphisms. In performing such inductive arguments, one must assume that the code $\delta_A F$ is larger, in some sense, than $FX$ where $X$ is any object of $\mathsf{Fam}\ I$. If we think of $FX$ as a code containing a free type variable, then this induction principle shares marked similarities with the induction principle for types of System F. While this can be done, it is certainly non-trivial.

The key to avoiding this size problem, and to get back to working with the small sets we know and love, is to internalise induction recursion to a universe itself. At first, this seemed like a daunting task, but the authors were very happy to see how the theoretical work we have engaged in so far made this task relatively simple. Now, all we need to do is to consider the right notion of reflection etc.

Concretely, lets ask ourselves what is so special about the universe of small sets. They certainly are special ... they index families and appear in the definition of containers, large families and a plethora of other constructions we have seen already. However, we can replace the universe $(\mathsf{Set}, \mathsf{El})$ (where $\mathsf{El} : \Pi S : Set.\ S \to \mathsf{Set}$) by some other universe $(S_0, E_0)$. To develop the theory of induction recursion wrt a universe $(S_0, E_0)$ we would need to define a families monad wrt this universe. Unfortunately this cannot be done with an arbitrary universe - rather we need a universe closed under $\Sigma$-types and containing a code for the one element set. Of course we can build freely such a universe - using of course - induction recursion. In Agda, this gives us the code:

```
mutual
          data $S$ : Set where
          $one : S$
          $\eta : S_0 \to S$
          $sig : (s : S) \to (Es \to S) \to S$

          $E : S \to$ Set
          $E\ one = \top$
          $E\ (\eta\ s) = E_0 s$
          $E\ (sig\ s\ f) = \Sigma\ (E\ s)\ (E \circ f)$
```

Given a universe such as $(S\ ,\ E)$ which has a code for the one element set and which is closed under $\Sigma$-types, we can define a families monad on Set. This monad indexes families not by sets, but by elements of $S$. Here is the definition in Agda

```
Fam : Set → Set
Fam$D = \Sigma\ S\ (\lambda s \to (Es \to D))$

Fam$_1$ : $\{D\ D'$ : Set$\} \to (D \to D') \to$ Fam$D \to$ Fam$D'$
Fam$_1$ $g\ (A\ ,\ f) = (A\ ,\ g \circ f)$

$\eta : \{D$ : Set$\} \to D \to$ Fam$D$
$\eta\ d = (one\ ,$ const $d)$

$\mu : \{D$ : Set$\} \to$ Fam(Fam$D) \to$ Fam$D$
$\mu\ \{D\}\ (s\ ,\ f) = (sig\ s\ ($proj$_1 \circ f)\ ,\ g)$
                    where $g : \Sigma\ (Es)\ (E \circ$ proj$_1 \circ f) \to D$
                          $g\ (a\ ,\ k) =$ proj$_2\ (fa)\ k$
```

As can be seen from this code, we get an honest to god monad on Set. Given our families monad relative to $(S\ ,\ E)$, we can capitalise on use our previous work to define a notion of internal induction recursion. All we need to do is take as input a functor $F : \mathsf{Set}^{op} \times \mathsf{Set} \to \mathsf{Set}$ and reflection $\rho : F\ I\ O \to \mathsf{Fam}\ I \to \mathsf{Fam}\ O$. Exactly as before! The strength of our previous work now becomes apparent - yet again no new effort is required to define internal induction recursion. Codes are again constructed by the free monad on the functor $F\ I$ at $O$

```
        data IR($I\ O$ : Set) : Set where
          $\iota : O \to$ IR $I\ O$
          $\sigma\delta : F\ I$ (IR $I\ O) \to$ IR $I\ O$
```

while the decode function for these codes uses the reflection parameter just as the specific instances previously have

```
    $[\![-]\!] : \{I\ O$ : Set$\} \to$ IR $I\ O \to$ Fam $I \to$ Fam $O$
    $[\![\ \iota\ d\ ]\!]\ T\ = \eta\ d$
    $[\![\ \sigma\delta\ F\ ]\!]\ T\ = (\mu \circ$ Fam$_1$(ev $T \circ [\![-]\!]))\ (\rho\ F\ T)$
```

## 7. Monadic Induction Recursion

The reader by now may feel we are belabouring the point and that we have made clear what the abstract structure of induction recursion is by demonstrating how a variety of different instances of induction recursion all arise in the same manner using the notion of reflection. But there is one last thing we wish to add.

Lets begin by saying there is one part of our treatment that is still a mystery ... namely the part of the decoding function for the $\sigma\delta$-constructor which is not part of the reflection. Namely the use of the multiplication of the Fam monad, the use of evaluation and the recursive use of the decoding function $[\![-]\!]$. Can we understand this part of induction recursion better? A related question is the following - is the families monad special. Perhaps another monad would suffice? After all, we have already seen how the standard families monad can be internalised to the internal families monad defined with respect to a universe closed under $\Sigma$ and containing a code for the one element type. And, while we are at it, shouldn't the decoding function be defined via a fold? Especially now that we are working over the category Set where traditional initial algebra semantics may be deployed safely.

Of course, we have answers to all these questions. Firstly, yes, one needs only a *strong* monad $(M, \eta, \mu, \tau)$ as opposed to being restricted to working only with the families monad. Given functor $F$ and such a monad, a reflection is then a natural transformation $\rho : F\ I\ O \to MI \to MO$. As expected, in this setting the IR codes are given once more by the free monad construction

```
        data IR($I\ O$ : Set) : Set where
          $\iota : O \to$ IR $I\ O$
          $\sigma\delta : F\ I$ (IR $I\ O) \to$ IR $I\ O$
```

But how will the use of an arbitrary monad $M$ effect the decoding function? Concretely, lets first define the fold for IR codes and then see how to define decoding using the fold operator.

```
fold : $\{I\ O\ X$ : Set$\} \to (O \to X) \to$
                         $(F\ I\ X \to X) \to$ IR $I\ O \to X$
fold $\alpha\ \beta\ (\iota\ o) = \alpha\ o$
fold $\alpha\ \beta\ (\sigma\delta\ K) = (F(fold\ \alpha\ \beta)K)$
```

With this initial algebra semantics in place, the decoding function can be given as a fold as desired.

```
$[\![-]\!] : \{I\ O$ : Set$\} \to$ IR $I\ O \to M\ I \to M\ O$
$[\![-]\!] =$ fold $\alpha\ \beta$ where
    $\alpha : \{I\ O$ : Set$\} \to O \to M\ I \to\ MO$
    $\alpha\ o\ t = \eta\ o$
    $\beta : \{I\ O$ : Set$\} \to F\ I\ (M\ I \to M\ O) \to M\ I \to M\ O$
    $\beta\ t\ m = (\mu \circ M_1(ev\ m))(\rho\ t\ m)$
```

So far we have answered two of the questions posed in this section - namely can the decoding function we written for an arbitrary monad and can it be written via a fold. But one last question remains ... why is the decoding function the way it is?

To answer this question, 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 reflection $\rho : F\ I\ O \to M\ I \to M\ O$ 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!

One final comment. The reader may start to think of arrows rather than monads, as the functor F is a mixed variant functor and so is the map sending $I\ O$ to $\mathsf{Fam}\ I \to \mathsf{Fam}\ O$. Indeed, when viewed like this, a reflection $\rho$ perhaps starts to resemble a arrow morphism. However, such thinking seems misplaced. Indeed, $\mathsf{Fam}\ I \to \mathsf{Fam}\ O$ is not an arrow as we cannot map a functor $\mathsf{Fam}I \to \mathsf{Fam}\ O$ to one $\mathsf{Fam}(I \times D) \to \mathsf{Fam}\ (O \times D)$.

## 8. Conclusions and Future Work

**Conclusions:** Starting from Dybjer and Setzers specific formulation of induction recursion, we have simplified its meta theory in a number of ways. Specifically, we have

- Exploited the families monad to bring out more of the intrinsic algebraic structure in both the constructors for IR codes and also in the decoding function for IR codes. So we can now work directly with families rather than separately with their index sets and then their decoding functions.
- Separated out covariant and contravariant occurrences in the target of the decoding function for IR codes so as to be able to characterise IR codes as an initial algebra arising from a free monad construction.
- Introduced containers into the presentation of induction recursion so as to be able to bring the algebra of containers to bear on that of induction recursion.
- Revealed that not only do induction recursive codes represent functors, but also that between such codes there are morphisms that represent natural transformations.
- Isolated a certain notion of reflection as a key ingredient in the definition of the decoding function for IR codes. In particular, the universal property of IR codes arising from their definition via a free monad construction characterises the decoding function as simply the monad morphism generated by a reflection.
- Shown how induction recursion can be internalised within a universe so as to address issues pertaining to size.
- Shown that in order to develop the theory of induction-recursion, no specific features of $\mathsf{Fam}$ are used beyond the structure it carries as a monad. The IR codes and their decoding as endofunctors can be defined for *any* strong monad.

These are, we hope, substantial contributions to the development of an algebraic treatment of induction recursion, revealing important new structure, specifically the categorical structure of hom-sets between the codes. Our hope is that this algebraic presentation of IR, by teasing out the structure hidden in the type-theoretical presentation, will help to make the subject more accessible to the academic community, as it deserves. To further achieve this goal, and reach those whose operational and computational intuitions are stronger than their categorical intuitions, we have also provided partial implementation of our results in Agda. This seems particularly important for a conference such as TLDI. We hope to have thus struck an appropriate balance between new results and cleaner presentations of known results, as well as delivered a deepening of the theoretical underpinnings of the subject, accompanied by practical constructions.

**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 [11] – beyond which lies the bottomless abyss of system $F$, $F^\omega$, impredicative higher-order logic, topos-theory, and the like. At the theoretical level we wish to fulfil the promise of this paper and show that IR morphisms can be mapped to natural transformations in a full and faithful way. This requires some sophisticated mathematics, centrally the characterisation of the semantics of the $\delta$-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 $IRIO$ 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 [4, 9] 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 structure of the 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 $\mathsf{Set}$, $\mathsf{Set}^I$, $\mathsf{Set} \to \mathsf{Set}$ and the like. For example, it lacks a terminal object in general (unless $I$ is small). As a deeper level, just as we have explained in this paper the key constructions in induction recursion (including the large set of IR codes) using universal properties, we would like to understand the universal property of not just the *set*, but rather the *category* of IR codes and morphisms.

The key contribution in this paper is to reveal the role reflection plays in induction recursion. But from where does that reflection arise? Why is it that we can turn *large* objects into the action of functions on *small* objects. We conjecture this is a very deep issue linked to the local smallness of the category $\mathsf{Set}$. 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.

## References

[1] M. Abott, T. Altenkirch, and N. Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

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

[3] Agda. The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2010.

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

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

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

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

[8] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, Apr. 1999.

[9] R. Hasegawa. Two applications of analytic functors. *Theor. Comput. Sci.*, 272(1-2):113–175, 2002. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(00)00349-2.

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

[11] R. Kahle and A. Setzer. An extended predicative definition of the Mahlo universe. In R. Schindler, editor, *Ways of ProofTheory. Festschrift on the occasion of Wolfram Pohler's retirement*.

[12] G. M. Kelly. Basic concepts of enriched category theory. *Reprints in Theory and Applications of Categories*, (10):1–136, 2005.

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

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

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