Special issue: Advanced Programming Techniques for Construction of Robust, General and Evolutionary Programs

Research Paper

Wander types : A formalization of coinduction-recursion*

Venanzio CAPRETTA[1]
[1]*School of Computer Science, University of Nottingham*

ABSTRACT

Wander types **are a coinductive version of inductive-recursive definitions. They are defined by simultaneously specifying the constructors of the type and a function on the type itself. The types of the constructors can refer to the function component and the function itself is given by pattern matching on the constructors.**

Wander types are different from inductive-recursive types in two ways: the structure of the elements is not required to be well-founded, so infinite applications of the constructors are allowed; and the recursive calls in the definition of the function are not required to be on structurally smaller arguments.

Wander types generalize several known type formers. We can use the functional component to control the way the data branch. This allows not only the implementation of coinduction, but also of induction, by imposing well-foundedness through an appropriate function definition. Special instances of wander types are: plain inductive and coinductive types, inductive-recursive types, mixed inductive-coinductive types, continuous stream processors.

KEYWORDS

Type Theory, Coinduction, Induction-Recursion

1 Introduction

Wander types are inspired by induction-recursion and coinduction: they combine both data operators in a new type builder.

Induction-recursion, introduced by Peter Dybjer [6], is a type-definition scheme in which an inductive type is defined simultaneously with a recursive function on it. They may depend on each other. The origin of the idea comes from Per Martin-Löf's definition of universes à la Tarski [15]. By *universe* we mean a type whose elements can themselves represent types. In the version à la Tarski, the members of the universe are codes and there is a function that decodes them into types. If the universe is to be closed under dependent products and other operators on type families, the decoding function needs to be used in the definition of the codes themselves. Erik Palmgren realized that this universe construction could be iterated and generalized [18], leading to a uniform way to generate universes over a set of types and the definition of a super-universe closed under this construction.

Michael Rathjen, Edward R. Griffor and Palmgren [19]–[21] studied the logical strength of the type system with the addition of such a super-universe and found that it corresponds to the addition of certain large cardinals (Mahlo numbers) in set theory.

Dybjer distilled the main idea into a general definition scheme. Besides the original application in the foundation of type theory, induction-recursion has other interesting instances. It can be used to define types with constructors

Received May 5, 2012; Revised September 30, 2012; Accepted December 19, 2012.

* This research is part of the project *Programming and Reasoning on Infinite Data Structures,* supported by EPSRC grant EP/I038713/1.

1) *venanzio.capretta@nottingham.ac.uk*

DOI: 10.2201/NiiPi.2013.10.4

©2013 National Institute of Informatics

that need to satisfy some constraint, for example lists without repetition, as shown by Catarina Coquand (cited in [6]). Some standard data structures, for instance leftist heaps and red-black trees [17 for example], have a natural inductive-recursive structure; this will be the topic of a forthcoming article. Ana Bove and I [2] used it to implement nested recursive functions simultaneously with their domain predicate.

Not all mutually dependent type and function specifications give a sensible definition. We need to impose constraints on the syntactic form of the types of the constructors and the recursive equations. Dybjer and Anton Setzer [7] found an elegant way of characterizing the acceptable instances by a simple inductive type of codes. Each code is interpreted as a functor on a category of families of types. The pair of inductive-recursive type and function is the initial algebra of this functor.

The idea of wander types is to investigate what the final coalgebras of those same functors are. In some cases they turn out to be very familiar advanced data structures. Some previously known constructions, induction-recursion itself, mixed induction-coinduction, continuous stream processors, can also be obtained as instances of wander types.

There is one limitation with respect to induction-recursion: we cannot use wander types to define universes. In fact, the coinductive definition of type classes easily leads to inconsistency. For instance, we could define a proposition consisting of the infinite application of the negation connective. Such a proposition would be its own negation, leading to a contradiction. A different approach to the coinductive generation of types was proposed by Martin-löf [16]: non-standard type theory. In that system we can consistently define an infinite sequence of propositions, each being the negation of the next. Here we put aside these foundational issues and limit ourselves to *small* wander types, in which the target type of the function component is always a set.

Let's illustrate wander types by a simple instance. We want to define a type of binary trees; they don't need to be well-founded, so they may have infinite paths. However, we want to impose some structural restriction about how the branching progresses. Let's choose the simple property that there cannot be any infinite zig-zag subpath inside the tree: it must not be possible to start at a node and then descend by going left-right-left-right-... forever (or right-left-right-left-...). Conventional coinductive types don't allow such a constraint. We need to define first a type of non-well-founded trees with no restriction, and then characterize a subset of them by a predicate. But we can realize it by simultaneously defining two functions that count the number of consecutive zig-zags. In the formalism of wander types, which we're going to define precisely later, we write:

$$
\begin{array}{ll}
\textsf{Wander}\ \ \textsf{type ZigZag} : \textsf{Set} & \textsf{function zigs} : \textsf{ZigZag} \to \mathbb{N} \\
 & \textsf{function zags} : \textsf{ZigZag} \to \mathbb{N} \\
 & \\
\textsf{leaf} : \textsf{ZigZag} & \textsf{node} : \textsf{ZigZag} \to \textsf{ZigZag} \to \textsf{ZigZag} \\
\quad \textsf{zigs leaf} = 0 & \quad \textsf{zigs}\,(\textsf{node}\, t_1\, t_2) = (\textsf{zags}\, t_1) + 1 \\
\quad \textsf{zags leaf} = 0 & \quad \textsf{zags}\,(\textsf{node}\, t_1\, t_2) = (\textsf{zigs}\, t_2) + 1.
\end{array}
$$

Read the definition as follows: *"We are defining simultaneously a type* ZigZag *and two functions* zigs *and* zags*. The constructors of the type are* leaf *and* node *and the functions must satisfy the given equations."* We want ZigZag to be the largest such type, therefore infinite sequences of constructors are allowed. However, the value of the two functions must be a (finite) natural number, which restricts how the infinite branches are allowed to progress.

For example, let's attempt to define some trees by the following recursive equations:

$$t_1 = \textsf{node leaf}\, t_1; \quad t_2 = \textsf{node}\, t_2\, t_2; \quad t_3 = \textsf{node}\, t_3\, t_4,\ t_4 = \textsf{node leaf}\, t_3.$$

We see that t_1 is well-defined, because it branches only to the right and therefore there are no zigzagging paths of any kind: $\textsf{zigs}\, t_1 = 1$, $\textsf{zags}\, t_1 = 2$. On the other hand t_2 is not well-defined because it contains zigzagging paths and $\textsf{zigs}\, t_2$ and $\textsf{zags}\, t_2$ are undefined: they should satisfy

$$
\begin{aligned}
\textsf{zigs}\, t_2 &= \textsf{zigs}\,(\textsf{node}\, t_2\, t_2) = (\textsf{zags}\, t_2) + 1 \\
&= (\textsf{zags}\,(\textsf{node}\, t_2\, t_2)) + 1 = ((\textsf{zigs}\, t_2) + 1) + 1 \\
&= (\textsf{zigs}\, t_2) + 2
\end{aligned}
$$

which is impossible in $\mathbb{N}$. The mutually dependent trees t_3 and t_4 are well-defined: $\textsf{zigs}\, t_3 = 3$, $\textsf{zags}\, t_3 = 2$, $\textsf{zigs}\, t_4 = 1$, $\textsf{zags}\, t_4 = 4$.

This ZigZag type may seem paradoxical for two reasons.

First, the dependency of the type itself on the two functions is not explicitly evident in the types of its constructors. However, there is an essential implicit constraint: if we had defined ZigZag as a plain coinductive type and then tried to define zigs and zags as recursive functions on it, we would have failed because of the lack of a structural induction principle.

The second puzzling feature is that well-definedness of objects specified by equations like the above is in general undecidable. With plain coinductive types, definition by guarded recursion guarantees productivity. The equations above are all guarded, but nevertheless t_2 is unacceptable. Therefore, when we lay down explicitly the formal rules for the definition of elements of a wander type, we must require that the user give some extra information guaranteeing the computability of the function components.

This is the structure of the article: Section 2 introduces induction-recursion and IR codes, it illustrates them using a new application to advanced data structures (leftist heaps); Section 3 introduces formally Dybjer-Setzer codes and their interpretation as functors on a slice category; Section 4 characterizes the final coalgebras of these functors; Section 5 reduces several known type constructors to wander types; Section 6 shows how to represent wander types as coinductive families; Section 7 discusses on-going work and open questions.

The article contains both an expository part explaining inductive-recursive definitions in a simple and accessible way and an original part introducing and studying wander types.

The review material, contained mostly in Section 3, describes the work by Dybjer and Setzer [7] following the exposition by Ghani, Hancock and others [8], [9]. In addition, Section 6, which gives the implementation of wander types as coinductive families, is a straightforward adaptation of the similar representation for inductive-recursive definitions. The idea is due to Conor McBride but has never being published before.

The original contributions of this work are:

- The new application of induction-recursion to define leftist heaps and, by analogy, other advanced data structures;
- The definition of wander types and their characterization as final coalgebras of functors associated to IR codes;
- The representation of induction-recursion, mixed induction-coinduction and continuous stream processors as wander types;
- The existence of wander types in the category of sets (via their representation as coinductive families).

Terminology and Notation In recent years, type theory has seen an exciting proliferation of new type constructors and their combinations in various guises. The terminology can be a bit confusing, using terms like *induction*, *coinduction*, *recursion*, *corecursion*, *simultaneous*, *mixed*, *indexed* in many combinations. For this reason I decided to give *coinduction-recursion* a more memorable name: *wander types*. The idea behind the terminology is that these definitions allow us to specify in fine detail how paths can travel, wander and meander through the structure.

We are going to use an intuitive notation for dependent products and sums that sees them as generalizations of function types and Cartesian products. It will be used to make the types of constructors more readable. When writing the types by themselves, we often use the standard notation for compactness.

Let $A : \mathsf{Set}$ be a type and $B : A \to \mathsf{Set}$ be a family of types indexed on it. We write $(x : A) \to (B\,x)$ for the dependent product of B. It is the type whose elements are functions f with domain A, such that, for every $a : A$, $(f\,a) : (B\,a)$. This type is often denoted by $\Pi x : A.B\,x$.

We write $(x : A) \times (B\,x)$ for the dependent sum of the family B. It is the type whose elements are pairs $\langle a, b\rangle$ with $a : A$ and $b : (B\,a)$. This type is often denoted by $\Sigma x : A.B\,x$.

We use canonical types with a finite number of elements: if $n : \mathbb{N}$, the type $\mathbb{N}_n$ has exactly n elements. We will denote them simply by $0, \ldots, n-1$. In set theory this notation can be exact, since we can define $\mathbb{N}_n$ as a subset of $\mathbb{N}$. In type theory the notation is ambiguous and we should really write $0_n, \ldots, (n-1)_n$. However, the ambiguity will always be dispelled by the context. Sometimes $\mathbb{N}_1$ is denoted by $\{\bullet\}$ and 0_1 by $\bullet$; $\mathbb{N}_2$ will be identified with the type of Booleans $\mathbb{B}$, 0_2 with false and 1_2 with true.

There are bijections between function types with a finite domain and Cartesian products. Therefore $\mathbb{N}_n \to A$ can be identified with $A^n = A \times \cdots \times A$. If $a_0, \ldots, a_{n-1} : A$, we may write $\langle a_0, \ldots, a_{n-1}\rangle$ for the function in $\mathbb{N}_n \to A$ mapping i to a_i. If $v : \mathbb{N}_n \to A$ and $i : \mathbb{N}_n$, we may write v_i for $(v\,i)$.

2 Induction-recursion

Many standard data types are naturally represented by inductive-recursive definitions. As an example, let's define *leftist heaps*, an efficient implementation of priority queues devised by Donald E. Knuth [13]. Here is their definition from Chris Okasaki's book:

> Heaps are often implemented as *heap-ordered* trees, in which the element at each node is no larger than the elements at its children. Under this ordering, the minimum element in a tree is always at the root.
>
> Leftist heaps are heap-ordered binary trees that satisfy the *leftist property*: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its *right spine* (i.e., the rightmost path from the node in question to an empty node). [17 pg.17]

Given an ordered type $\mathcal{A} = \langle A, \leqslant \rangle$, leftist heaps on it are defined as a recursive tree structure simultaneously with the notions of heap-ordering and rank. Here is their formalization, where we use a function returning the root of a node (with a default element for leaves) to specify the ordering.

$$\begin{array}{lll}
\mathsf{IndRec} & \mathsf{type}\ \mathsf{LHeap}_{\mathcal{A}} : \mathsf{Set} & \mathsf{function}\ \mathsf{root} : \mathsf{LHeap}_{\mathcal{A}} \to A \to A \\
 & & \mathsf{function}\ \mathsf{rank} : \mathsf{LHeap}_{\mathcal{A}} \to \mathbb{N}
\end{array}$$

$$\begin{array}{ll}
\mathsf{Empty} : \mathsf{LHeap}_{\mathcal{A}} & \mathsf{Fork} : \ (a : A; t_1, t_2 : \mathsf{LHeap}_{\mathcal{A}}) \to \\
\quad \mathsf{root}\,\mathsf{Empty}\,x = x & \qquad a \leqslant \mathsf{root}\,t_1\,a \to a \leqslant \mathsf{root}\,t_2\,a \to \\
\quad \mathsf{rank}\,\mathsf{Empty} = 0 & \qquad \mathsf{rank}\,t_2 \leq \mathsf{rank}\,t_1 \to \mathsf{LHeap}_{\mathcal{A}} \\
 & \quad \mathsf{root}\,(\mathsf{Fork}\,a\,t_1\,t_2\,p_1\,p_2\,q)\,x = a \\
 & \quad \mathsf{rank}\,(\mathsf{Fork}\,a\,t_1\,t_2\,p_1\,p_2\,q) = (\mathsf{rank}\,t_2) + 1
\end{array}$$

In this definition we used an informal way to specify inductive-recursive types. As in the case of wander types, the specification consists in four parts: the declaration of the type ($\mathsf{LHeap}_{\mathcal{A}}$), the declaration of the function components and their type (root and rank), the constructors of the type (Empty and Fork) and the recursive equations for the functions.

The use of a default value for the root function doesn't affect the invariants. In the constructor Fork, we apply it to check whether the new root a is smaller or equal to all elements of the subtrees t_1 and t_2. The default value appears only when a subtree is empty; in that case a is compared with itself, correctly giving a true condition. There are two possible alternatives. The first consists in assuming that A has a top element, replacing the default; but this is not always the case, for example for natural numbers; we could simply add it, slightly complicating the development. The second alternative would be to use, in place of root, a relation $(\leqslant_{\mathsf{all}}) : \mathcal{A} \to \mathsf{LHeap}_{\mathcal{A}} \to \mathsf{Set}$; this, however, would require a large target type, which is problematic.

Dybjer and Setzer invented an inductive type of codes for inductive-recursive definitions with three simple constructors [7]–[9] (the choice of names for the constructors is mine).

Definition 1 *For every set D, the set of* inductive-recursive codes *over the domain D is the type recursively generated by the following inductive specification.*

$$\begin{array}{ll}
\mathsf{Inductive} & \mathsf{type}\ \mathsf{IR}_D : \mathsf{Set} \\
 & \mathsf{element} : D \to \mathsf{IR}_D \\
 & \mathsf{choose} : (A : \mathsf{Set}) \to (A \to \mathsf{IR}_D) \to \mathsf{IR}_D \\
 & \mathsf{recurse} : (I : \mathsf{Set}) \to ((I \to D) \to \mathsf{IR}_D) \to \mathsf{IR}_D.
\end{array}$$

Every element $c : \mathsf{IR}_D$ is a code representing an inductive-recursive definition of a type Elem_c and a function $\mathsf{fun}_c : \mathsf{Elem}_c \to D$. Specifically:

- We use the first constructor to generate elements and to specify the value of the function on them by a constant. If $c = (\mathsf{element}\,d)$, then $\mathsf{Elem}_c = \{\bullet\}$ is a singleton with $\mathsf{fun}_c(\bullet) = d$;
- We use the second constructor to make the disjoint union of a family of inductive-recursive types, with the function computed as in each of the components. If $c = (\mathsf{choose}\,A\,u)$, then $\mathsf{Elem}_c = (a : A) \times \mathsf{Elem}_{(u\,a)}$ and $\mathsf{fun}_c(\langle a, x \rangle) = \mathsf{fun}_{(u\,a)}(x)$.

- We use the third constructor to generate recursive structures that can depend on the the functional component. An object has a tree structure given by a family $t : I \to \mathsf{Elem}_c$ of children indexed by a type I. For each child we assume that we know how to compute the functional part, that is, we know $\mathsf{fun}_c \circ t$. We can use these recursive values to specify the type of a further component: you can think of it as the content of the node or as a constraint. This extra component is given by a code $v(\mathsf{fun}_c \circ t)$ that can depend of the values of fun_c on t. If $c = (\mathsf{recurse}\, I\, v)$, then $\mathsf{Elem}_c = (t : I \to \mathsf{Elem}_c) \times \mathsf{Elem}_{v(\mathsf{fun}_c \circ t)}$ and $\mathsf{fun}_c(\langle t, y\rangle) = \mathsf{fun}_{v(\mathsf{fun}_c \circ t)}(y)$.

Notation To make the code a bit more readable, we use several typographic conventions.

We use the convention that λ-abstractions extend a far as possible to the right, even across several lines. In particular, there is no need to put parentheses around them when giving them as arguments to higher-order operators. We show this immediately in the displayed formulas below.

An application of the choose constructor with the Booleans as indexing set will be written as a sum, that is, we write

$$c_1 + c_2 \quad \text{for} \quad \mathsf{choose}\, \mathbb{B}\, \lambda b.\mathsf{if}\, b\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2.$$

An application of the recurse constructor with a finite type as indexing set will be written using tuple notation instead of functions: if $I = \mathbb{N}_n$ and c is a term of type IR_D depending on variables $d_0, \ldots, d_{n-1} : D$, then we write

$$\mathsf{rec}\, \lambda\langle d_0, \ldots, d_{n-1}\rangle.c \quad \text{for} \quad \mathsf{recurse}\, \mathbb{N}_n\, \lambda h.c[d_0 := h\, 0, \ldots, d_{n-1} := h\, (n-1)].$$

If we abstract over a function variable h of type $\mathbb{B} \to X_1 \times X_2$, we pattern-match it as a pair of pairs, rather that using applications and projections everywhere, that is, we write $\lambda\, \langle\langle x_{11}, x_{12}\rangle, \langle x_{21}, y_{22}\rangle\rangle.e[x_{11}, x_{12}, x_{21}, x_{22}]$ for

$$\lambda h.e[\pi_1\, (h\, \mathsf{true}), \pi_2\, (h\, \mathsf{true}), \pi_1\, (h\, \mathsf{false}), \pi_2\, (h\, \mathsf{false})].$$

As an illustrative application, we can represent the type of leftist heaps as an inductive-recursive definition by a Dybjer-Setzer code.

Definition 2 *The* inductive-recursive code for leftist heaps *is the element* lheap *of* $\mathsf{IR}_{(A\to A)\times\mathbb{N}}$ *defined by*[1]*:*

$$\begin{aligned}\mathsf{lheap} = \;& \mathsf{element}\, \langle \mathsf{id}, 0\rangle + \\ & \mathsf{choose}\, A\, \lambda a.\mathsf{rec}\, \lambda\langle\langle \mathsf{vroot}_1, \mathsf{vrank}_1\rangle, \langle \mathsf{vroot}_2, \mathsf{vrank}_2\rangle\rangle. \\ & \quad \mathsf{choose}\;\; (a \preccurlyeq (\mathsf{vroot}_1\, a) \wedge a \preccurlyeq (\mathsf{vroot}_2\, a) \wedge (\mathsf{vrank}_2 \leq \mathsf{vrank}_1)) \\ & \qquad\qquad \lambda\_.\, \mathsf{element}\, \langle \lambda x.a, 1 + \mathsf{vrank}_2\rangle.\end{aligned}$$

(We used the above conventions. In particular, the lambda abstraction $\lambda a. \ldots$ extends over the next to lines to the very end of the term and so does the next abstraction of the tuple.)

The code should be read as follows. *There are two constructors. The first gives an element with the identity as root function and rank 0. The second is a family indexed on the set A. To construct one of its elements we must choose an a and two recursive elements with root and rank* vroot_1, vrank_1, vroot_2, vrank_2*; we must give a proof of the ordering and leftist predicate; finally we obtain a new element with root (the constant) a and rank* $1 + \mathsf{vrank}_2$.

3 IR codes as functors

We interpret IR codes as functors in a slice category. Assume that types are interpreted in the base category Set. Also assume that the domain D used in IR codes is *small*, that is, it is interpreted as an object of Set. In the original application to generate universes, D is allowed to be the whole class Set itself. Originally, Dybjer and Setzer developed an interpretation in categories of *families*, where D is allowed to be large. The model we describe is essentially the same as theirs in the simpler case where D is a set.

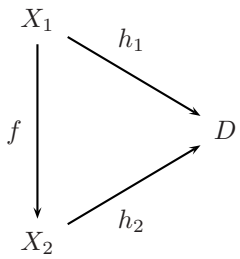
Every element of IR_D can be interpreted as a functor on the category of sets with mappings to D.

Definition 3 *The* slice *category* $\mathsf{Set} \downarrow D$ *has:*

objects: *pairs* $\langle X, h\rangle$ *of an object* X *of* Set *and a morphism* $h : X \to D$ *in* Set*;*

[1] My thanks to Peter Hancock, who helped me get this definition right and gave me enlightening insights about IR codes.

morphisms: *$f : \langle X_1, h_1\rangle \to \langle X_2, h_2\rangle$ is a morphism $f : X_1 \to X_2$ in* Set *such that the following triangle commutes:*



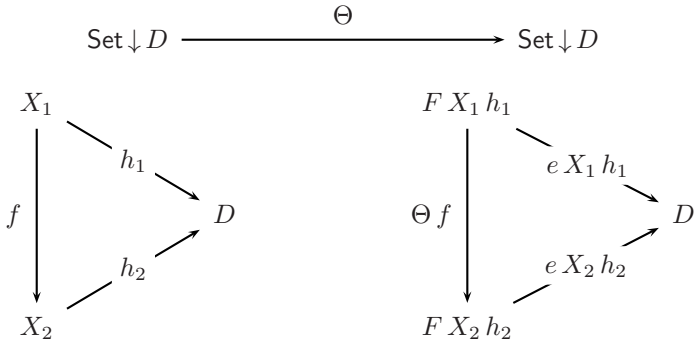
$$h_2 \circ f = h_1.$$

Lemma 4 *An endofunctor Θ : Set$\downarrow D \to$ Set$\downarrow D$ is characterized by the following components:*

object mapping: *For every object X of* Set *and morphism $h : X \to D$ in* Set*, an object $(F\ X\ h)$ of* Set*;*

arrow mapping: *For every object X of* Set *and morphism $h : X \to D$ in* Set*, a morphism $(e\ X\ h) : F\ X\ h \to D$ in* Set*;*

functorial lifting: *For every morphism $f : \langle X_1, h_1\rangle \to \langle X_2, h_2\rangle$ in* Set$\downarrow D$*, a morphism $\Theta\ f : \langle F\ X_1\ h_1, e\ X_1\ h_1\rangle \to \langle F\ X_2\ h_2, e\ X_2\ h_2\rangle$.*



Finally, the functor Θ must preserve identities and compositions (which we can check in the base category Set*).*

Proof. It's a direct unfolding of the definition of functor for the slice category. Notice, in passing, that Θ is not necessarily an extension of a functor in Set, unless the object component F doesn't depend on the h argument. □

Every code in IR_D defines an endofunctor on the slice category Set$\downarrow D$. We give the general construction first and then we illustrate it with the leftist heaps code.

Definition 5 *For every code $c : \mathsf{IR}_D$ we define a functor Θ_c : Set$\downarrow D \to$ Set$\downarrow D$ with components F_c and e_c by recursion on the structure of c as follows.*

$c =$ element d**:**

$$\begin{aligned}\mathsf{F}_c\ X\ h &= \{\bullet\}\\ \mathsf{e}_c\ X\ h\ \bullet &= d\\ \Theta_c\ f\ \bullet &= \bullet;\end{aligned}$$

$c =$ choose $A\ u$**:**

$$\begin{aligned}\mathsf{F}_c\ X\ h &= \Sigma a : A.\mathsf{F}_{(u\,a)}\ X\ h\\ \mathsf{e}_c\ X\ h\ \langle a, x\rangle &= \mathsf{e}_{(u\,a)}\ X\ h\ x\\ \Theta_c\ f\ \langle a, x\rangle &= \langle a, \Theta_{(u\,a)}\ f\ x\rangle;\end{aligned}$$

$c =$ recurse $I\ v$**:**

$$\begin{aligned}\mathsf{F}_c\ X\ h &= \Sigma t : I \to X.\mathsf{F}_{v(h\circ t)}\ X\ h\\ \mathsf{e}_c\ X\ h\ \langle t, y\rangle &= \mathsf{e}_{v(h\circ t)}\ X\ h\ y\\ \Theta_c\ f\ \langle t, y\rangle &= \langle f \circ t, \Theta_{v(h_1\circ t)}\ f\ y\rangle.\end{aligned}$$

In the last case of the definition, notice that the result has the correct type because f is a morphism in the slice category, that is $h_2 \circ f = h_1$. In fact we have that:

$$\Theta_{v(h_1 \circ t)}\, f : \mathsf{F}_{v(h_1 \circ t)}\, X_1\, h_1 \to \mathsf{F}_{v(h_1 \circ t)}\, X_2\, h_2$$
$$\Theta_{v(h_1 \circ t)}\, f\, y : \mathsf{F}_{v(h_1 \circ t)}\, X_2\, h_2 = \mathsf{F}_{v(h_2 \circ f \circ t)}\, X_2\, h_2.$$

Theorem 6 *For every code $c : \mathsf{IR}_D$, the operator Θ_c is an endofunctor on $\mathsf{Set}\downarrow D$.*

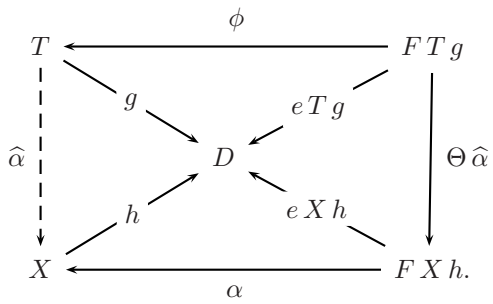
Proof. Verifying that the above definition of Θ_c satisfies all the conditions of Lemma 4 is routine. □

Inductive-recursive definitions represent initial algebras of such functors. Let's see what an algebra for a functor $\Theta : \mathsf{Set}\downarrow D \to \mathsf{Set}\downarrow D$ with components F and e looks like. An algebra is a triple $\langle X, h, \alpha\rangle$ with X a set, $h : X \to D$, and α a morphism in $\mathsf{Set}\downarrow D$ of type

$$\begin{aligned}\alpha : \quad & \Theta\,\langle X\, h\,\rangle \to \langle X, h\rangle \\ & \langle F\, X\, h, e\, X\, h\rangle \to \langle X, h\rangle,\end{aligned}$$

that is, $\alpha : F\, X\, h \to X$ such that $h \circ \alpha = e\, X\, h$.

An initial algebra $\langle T, g, \phi\rangle$ is such that for every algebra $\langle X, h, \alpha\rangle$, there is a unique morphism $\widehat{\alpha} : \langle T, g, \phi\rangle \to \langle X, h, \alpha\rangle$ making the following diagram commute:



Let's look at the functor associated to leftist heaps, $\Theta_{\mathsf{lheap}} : \mathsf{Set}\downarrow((A \to A)\times\mathbb{N}) \to \mathsf{Set}\downarrow((A \to A)\times\mathbb{N})$. If $\langle X, h\rangle$ is any object of the slice category, we write $h = \langle h_a, h_r\rangle$. We beautify the expression by using some obvious notational simplifications. The components of the functor are:

$$\begin{aligned}&\mathsf{F}_{\mathsf{lheap}}\, X\, h = \{\bullet\} + \Sigma a : A.\Sigma x_1, x_2 : X. \\ &\qquad\qquad (a \preccurlyeq h_a\, x_1\, a) \times (a \preccurlyeq h_a\, x_2\, a) \times (h_r\, x_2 \le h_r\, x_1) \\ &\mathsf{e}_{\mathsf{lheap}}\, X\, h\, (\mathsf{inl}\, \bullet) = \langle \mathsf{id}, 0\rangle \\ &\mathsf{e}_{\mathsf{lheap}}\, X\, h\, (\mathsf{inr}\, \langle a, x_1, x_2, p_1, p_2, q\rangle) = \langle \lambda\_.a, (h_r\, x_2) + 1\rangle \\ &\Theta_{\mathsf{lheap}}\, f\, (\mathsf{inl}\, \bullet) = (\mathsf{inl}\, \bullet) \\ &\Theta_{\mathsf{lheap}}\, f\, (\mathsf{inr}\, \langle a, x_1, x_2, p_1, p_2, q\rangle) = (\mathsf{inr}\, \langle a, (f\, x_1), (f\, x_2), p_1, p_2, q\rangle).\end{aligned}$$

Notice that the proof components in the functorial lifting don't need to be changed, because of the equalities $h_{2,a} \circ f = h_{1,a}$ and $h_{2,r} \circ f = h_{1,r}$.

The functor Θ_{lheap} has an initial algebra. In Section 6 this will be proved in general by using a reduction to standard type theory by Conor McBride. Here, let's look at a specific model in classical set theory. Its purpose is just to show that the inductive-recursive definition of leftist heaps corresponds to a traditional mathematical representation. However, the moral is that one should not work with such set-theoretic characterization but assume as primitive the existence of the more natural inductive-recursive types.

We adopt the usual set theoretic definition of well-founded binary trees as subsets of paths with the constraints corresponding to the heap ordering and leftist property.

Definition 7 *A well-founded "naked" binary tree is a set of finite paths ($\mathbb{N}_2^*$ is the set of finite sequences of Booleans, with ϵ the empty sequence; if $p \in \mathbb{N}_2^*$ and $b \in \mathbb{N}_2$, then $pb \in \mathbb{N}_2^*$ is the concatenation if p and b):*

$$\begin{aligned}\mathcal{T} = \{t \subseteq \mathbb{N}_2^* \mid \quad & \forall p \in \mathbb{N}_2^*.\forall b \in \mathbb{N}_2.pb \in t \Rightarrow p \in t \quad \wedge \\ & \forall \beta : \mathbb{N} \to \mathbb{N}_2.\exists n : \mathbb{N}.\beta(0)\cdots\beta(n) \notin t\}.\end{aligned}$$

Note that if $t \in \mathcal{T}$, the elements of t represent paths to nodes. Leaves are implicitly attached below non-extensible paths. So if $p \in t$ and $p1 \notin t$, then the right child of the node denoted by p is a leaf. The empty set $\emptyset \in \mathcal{T}$ denotes the tree consisting of a single leaf. If $t \in \mathcal{T}$, let's denote by $\tilde{t}$ the extension of t with paths to leafs:

$$\tilde{t} = t \cup \{\epsilon\} \cup \{p0 \mid p \in t\} \cup \{p1 \mid p \in t\}.$$

Then we define $\mathsf{rank}(t) = \max\{n \in \mathbb{N} \mid 1^n \in \tilde{t}\}$.

If $p \in \tilde{t}$ then we define the subtree at p *by*

$$t|p = \{p' \in \mathbb{N}_2^* \mid pp' \in t\} \in \mathcal{T}.$$

A leftist heap is a tree with a labelling function satisfying the heap-ordering and leftist heap property:

$$\mathcal{H}_A = \{\langle t, l\rangle \mid t \in \mathcal{T} \wedge l : t \to A \ \wedge \ \forall p, p' \in \mathbb{N}_2^*.pp' \in t \to l(p) \preccurlyeq l(pp') \wedge \forall p \in t.\mathsf{rank}(t|p1) \leq \mathsf{rank}(t|p0)\}.$$

Define $\mathsf{root} : \mathcal{H}_A \to A \to A$ *by*

$$\mathsf{root}(\langle t, l\rangle) = \begin{cases} \mathsf{id} & \text{if } t = \emptyset \\ \lambda\_.l(\epsilon) & \text{otherwise.} \end{cases}$$

Providing the set $\mathcal{H}_A$ with a Θ_{lheap}-algebra structure simply means defining the leaf tree and the operation of joining two children to their parent node. We already remarked that the leaf is represented by the empty set $\emptyset$.

Definition 8 *Define the empty heap as* $\mathsf{Empty} = \langle \emptyset, !_A\rangle \in \mathcal{H}_A$, *where $!_A$ is the unique function from $\emptyset$ to A.*

For every $a \in A$, $\langle t_1, l_1\rangle, \langle t_2, l_2\rangle \in \mathcal{H}_A$, we define a new pair of a tree and a labelling function:

$$\begin{array}{l} \mathsf{Fork}\, a\, \langle t_1, l_1\rangle\, \langle t_2, l_2\rangle = \langle t, l\rangle \in \mathcal{H}_A \text{ with} \\ \quad t = \{\epsilon\} \cup \{0p \mid p \in t_1\} \cup \{1p \mid p \in t_2\} \\ \quad l\,\epsilon = a, \quad l\,(0p) = l_1\, p, \quad l\,(1p) = l_2\, p. \end{array}$$

Lemma 9 *If $a \in A$, $\langle t_1, l_1\rangle, \langle t_2, l_2\rangle \in \mathcal{H}_A$ satisfy the properties*

$$a \preccurlyeq \mathsf{root}\,\langle t_1, l_1\rangle\, a \quad \wedge \quad a \preccurlyeq \mathsf{root}\,\langle t_2, l_2\rangle\, a \quad \wedge \quad \mathsf{rank}(t_2) \leq \mathsf{rank}(t_1),$$

then $\mathsf{Fork}\, a\, \langle t_1, l_1\rangle\, \langle t_2, l_2\rangle \in \mathcal{H}_A$.

Proof. A simple verification. □

Theorem 10 *The triple $\langle \mathcal{H}_A, \langle \mathsf{root}, \mathsf{rank} \circ \pi_1\rangle\rangle$ is the carrier of an initial algebra of Θ_{lheap}.*

Proof. The initial algebra is given by the morphism

$$\begin{array}{rl} \phi_{\mathsf{lheap}} = & [\lambda \bullet .\mathsf{Empty}, \lambda\langle a, x_1, x_2, p_1, p_2, q\rangle.\mathsf{Fork}\, a\, x_1\, x_2] \\ & : \mathsf{F}_{\mathsf{lheap}}\, \mathcal{H}_A\, \langle \mathsf{root}, \mathsf{rank} \circ \pi_1\rangle \to \mathcal{H}_A. \end{array}$$

First of all, we must check that this is an algebra in the slice category, that is, it satisfies the equality

$$\langle \mathsf{root}, \mathsf{rank} \circ \pi_1\rangle \circ \phi_{\mathsf{lheap}} = \mathsf{e}_{\mathsf{lheap}}\, \mathcal{H}_A\, \langle \mathsf{root}, \mathsf{rank} \circ \pi_1\rangle.$$

In fact, by definition and simple calculation, both sides of the equality return $\langle \mathsf{id}, 0\rangle$ when applied to $(\mathsf{inl}\, \bullet)$ and $\langle \lambda\_.a, (\mathsf{rank}\, t_2) + 1\rangle$ when applied to $(\mathsf{inr}\, \langle a, \langle t_1, l_1\rangle, \langle t_2, l_2\rangle, p_1, p_2, q\rangle)$.

Let $\langle X, h, \alpha\rangle$ be any Θ_{lheap}-algebra. We must construct a morphism $\widehat{\alpha} : \mathcal{H}_A \to X$. Let's first define the left and right children of any nonempty heap. If $\langle t, l\rangle : \mathcal{H}_A$ is such that $t \neq \emptyset$, then put

$$\mathsf{lchild}\, \langle t, l\rangle = \langle \{p \mid 0p \in t\}, \lambda p.l\, 0p\rangle, \qquad \mathsf{rchild}\, \langle t, l\rangle = \langle \{p \mid 1p \in t\}, \lambda p.l\, 1p\rangle.$$

We can now define the catamorphism recursively, exploiting the fact that each $t \in \mathcal{T}$ is finite by König's lemma: $\widehat{\alpha}$ is defined on nonempty heaps in terms of its values on the left and right children.

At the same time, one can inductively prove that $\widehat{\alpha}$ is a morphism in the slice category, that is:

$$h_a \circ \widehat{\alpha} = \mathsf{root}, \quad h_r \circ \widehat{\alpha} = \mathsf{rank} \circ \pi_1.$$

Again, these equalities can be proved for a given non-empty heap from the corresponding equalities for its children.

$$\widehat{\alpha}\,\langle t, l\rangle = \begin{cases} \alpha\,(\mathsf{inl}\,\bullet) & \text{if } t = \emptyset \\ \alpha\,(\mathsf{inr}\,a\,x_1\,x_2\,p_1\,p_2\,q) & \text{otherwise} \end{cases}$$

where $a = (l\,\epsilon)$, $x_1 = \widehat{\alpha}\,(\mathsf{lchild}\,\langle t, l\rangle)$, $x_2 = \widehat{\alpha}\,(\mathsf{rchild}\,\langle t, l\rangle)$. The proofs p_1, p_2, q follow from the corresponding properties of $\langle t, l\rangle$ by the above morphism equalities.

It is now a question of unfolding definitions and routine calculations to show that $\widehat{\alpha}$ is a morphism of Θ_{lheap}-algebras.

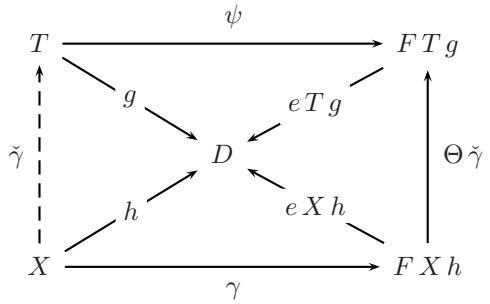
That it is the unique such morphism can be proved by verifying that its definition is forced by the commutativity of the algebraic morphism diagram. The only subtle point is the possibility that there could be different values for the proofs p_1, p_2, q. However, in the set-theoretic framework we are adopting here, proof-irrelevance holds and α is not supposed to depend on those proofs. □

4 Final coalgebras

The starting point of this work was the study of the final coalgebras of functors associated with IR codes. The data structures that we obtain are similar to the ones given by the initial algebras, except that they're not required to be well-founded. However, the well-definedness of the functional part of the coalgebra may still impose that some parts of the structure is well-founded. For example, the final coalgebra for the functor Θ_{lheap} consists of potentially infinite binary trees, but the right spines must still be finite for the rank function to be defined.

Given and endofunctor $\Theta : \mathsf{Set}\downarrow D \to \mathsf{Set}\downarrow D$ with components F and e, a Θ-coalgebra is a triple $\langle X, h, \gamma\rangle$ where X is an object of Set, $h : X \to D$ and $\gamma : X \to F\,X\,h$ such that $h = e\,X\,h \circ \gamma$.

A final coalgebra $\langle T, g, \psi\rangle$ is such that for every coalgebra $\langle X, h, \gamma\rangle$, there is a unique morphism $\check{\gamma} : \langle X, h, \gamma\rangle \to \langle T, g, \psi\rangle$ making the following diagram commute:



Let's consider the zig-zag type informally described in the introduction. Its IR code is:

$$\begin{aligned} &\mathsf{zigzag} : \mathsf{IR}_{\mathbb{N}\times\mathbb{N}} \\ &\mathsf{zigzag} = \mathsf{element}\,\langle 0, 0\rangle + \mathsf{rec}\,(\lambda\,\langle\langle i_1, j_1\rangle, \langle i_2, j_2\rangle\rangle.\mathsf{element}\,\langle j_1 + 1, i_2 + 1\rangle) \end{aligned}$$

with associated endofunctor on $\mathsf{Set}\downarrow \mathbb{N}\times\mathbb{N}$ (assuming $h = \langle h_1, h_2\rangle$):

$$\begin{aligned} &\mathsf{F}_{\mathsf{zigzag}}\,X\,h = \{\bullet\} + X \times X \\ &\mathsf{e}_{\mathsf{zigzag}}\,X\,h\,(\mathsf{inl}\,\bullet) = \langle 0, 0\rangle \\ &\mathsf{e}_{\mathsf{zigzag}}\,X\,h\,(\mathsf{inr}\,\langle x_1, x_2\rangle) = \langle (h_2\,x_1) + 1, (h_1\,x_2) + 1\rangle \\ &\Theta_{\mathsf{zigzag}}\,f\,(\mathsf{inl}\,\bullet) = (\mathsf{inl}\,\bullet) \\ &\Theta_{\mathsf{zigzag}}\,f\,(\mathsf{inr}\,\langle x_1, x_2\rangle) = \mathsf{inr}\,\langle (f\,x_1), (f\,x_2)\rangle. \end{aligned}$$

Since the function component h is not used in the definition of $\mathsf{F}_{\mathsf{zigzag}}$, this is an extension of the functor $F\,X = \{\bullet\} + X \times X$ on Set. However, the final coalgebra is not obtained from the final coalgebra of F in Set, which consists of all binary trees (finite and infinite). The final coalgebra of Θ_{zigzag} consists of those trees that don't contain any

infinite zig-zagging path. The functional components of the final coalgebra count the lengths of the zigzagging paths starting to the left and right, respectively.

As we have done for the leftist heaps data structure, we can give a formal set-theoretic definition of the type and then prove that it is a final coalgebra for Θ_{zigzag}.

Definition 11 *The set $\mathcal{T}_\infty$ of all trees is defined by saying that a tree is a set of paths; we simply drop the finiteness requirement from the definition of $\mathcal{T}$:*

$$\mathcal{T}_\infty = \{t \subseteq \mathbb{N}_2^* \mid \forall p \in \mathbb{N}_2^*.\forall b \in \mathbb{N}_2.pb \in t \Rightarrow p \in t\}$$
$$\mathcal{Z} = \{t \in \mathcal{T}_\infty \mid \forall p \in t.\exists k \in \mathbb{N}.p(01)^k \notin t\}.$$

The functional components can be defined as

$$z_1\, t = \mathsf{max}\,(\{2k+1 \mid (01)^k \in t\} \cup \{2k+2 \mid (01)^k 0 \in t\})$$
$$z_2\, t = \mathsf{max}\,(\{2k+1 \mid (10)^k \in t\} \cup \{2k+2 \mid (10)^k 1 \in t\}).$$

The coalgebra on this set returns just the subtrees:

$$\phi : \mathcal{Z} \to \{\bullet\} + \mathcal{Z}^2$$
$$\phi\, t = \begin{cases} \mathsf{inl}\,\bullet & \textit{if } t = \emptyset \\ \mathsf{inr}\,\langle t|0, t|1\rangle & \textit{otherwise} \end{cases}$$

Lemma 12 *$\langle \mathcal{Z}, \langle z_1, z_2\rangle, \phi\rangle$ is a final coalgebra for the functor Θ_{zigzag}.*

Proof. Let (X, h, γ) be a Θ_{zigzag}-coalgebra. We observed earlier that it is the extension of a coalgebra $\gamma : X \to \{\bullet\} + X \times X$ in Set. Therefore there exists a unique coalgebra morphism $\check{\gamma} : X \to \mathcal{T}_\infty$, because $\langle \mathcal{T}_\infty, \phi\rangle$ is a final coalgebra in Set.

The decision of whether $p \in \check{\gamma}(x)$ is taken by following the path p through γ starting at x. If that is the case, we obtain an element $x_p \in X$ such that $\check{\gamma}(x_p) = \check{\gamma}(x)|p$. This is done by recursion on the length of p. If $p = \epsilon$, then $p \in \check{\gamma}(x)$ if and only $\gamma(x) = \mathsf{inr}\,\langle y_0, y_1\rangle$ for some $y_0, y_1 \in X$ and $x_p = x$. If $p = p'b$, we have two cases. If $p' \notin \check{\gamma}(x)$, then also $p \notin \check{\gamma}(x)$. If $p' \in \check{\gamma}(x)$, then we have $x_{p'} \in X$ such that $\check{\gamma}(x_{p'}) = \check{\gamma}(x)|p'$. Now, if $\gamma(x_{p'}) = \mathsf{inl}\,\bullet$, then $p \notin \check{\gamma}(x)$; if $\gamma(x_{p'}) = \mathsf{inr}\,\langle y_0, y_1\rangle$, then $p \in \check{\gamma}(x)$ and $x_p = y_b$.

To verify that $\check{\gamma}$ is an anamorphism for Θ_{zigzag}, we just need to check that for every $x \in X$, $\check{\gamma}(x) \in \mathcal{Z}$, that is, we must prove that

$$\forall x \in X.\forall p \in \check{\gamma}(x).\exists k \in \mathbb{N}.p(01)^k \notin \check{\gamma}(x).$$

Observe, first of all, that we can reduce the proof to the case when p is the empty path ϵ, because we know that for every path $p \in \check{\gamma}(x)$ there exists $x_p \in X$ such that $\check{\gamma}(x_p) = \check{\gamma}(x)|p$, so that $p(01)^k \in \check{\gamma}(x)$ if and only if $(01)^k \in \check{\gamma}(x_p)$.

In the case that $p = \epsilon$, choose k such that $2k + 1 > h_1(x)$; the statement follows from the fact that $z_1(\check{\gamma}(x)) = h_1(x)$ and by maximality of z_1. □

The zig-zag type shows that the carrier of the final coalgebra in the slice category is not always the final coalgebra in the base category, even when the functor is lifted from the base. In that case we obtained a subset. In other cases we may get a refinement. Let's consider, for example, the type specified by:

$$\mathsf{Wander}\ \ \mathsf{type}\ \mathsf{InfNot} : \mathsf{Set} \quad \mathsf{function}\ \mathsf{invalue} : \mathsf{InfNot} \to \mathbb{B}$$

$$\mathsf{nons} : \mathsf{InfNot} \to \mathsf{InfNot}$$
$$\mathsf{invalue}\,(\mathsf{nons}\,x) = \overline{\mathsf{invalue}\,x}$$

where $\overline{b}$ is the Boolean negation of $b : \mathbb{B}$. Its IR code is $\mathsf{infnot} = \mathsf{rec}\,(\lambda b.\mathsf{element}\,\overline{b})$.

The associated functor on $\mathsf{Set}\downarrow\mathbb{B}$ is an extension of the identity functor: $\mathsf{F}_{\mathsf{infnot}}\,X\,h = X$, $\mathsf{e}_{\mathsf{infnot}}\,X\,h\,x = \overline{h\,x}$. The final coalgebra of the identity functor in Set is the unit type $\{\bullet\}$. However, that doesn't lift to the slice category: $\bullet$ would have to be its own nons, but then $\mathsf{invalue}\,\bullet = \mathsf{invalue}\,(\mathsf{nons}\,\bullet) = \overline{\mathsf{invalue}\,\bullet}$, which can be neither true nor false. The temptation is, as before, to exclude the elements on which the functional component is undefined, obtaining the empty set. In fact that's incorrect as well. The final Θ_{infnot}-coalgebra is, instead, $\langle \mathbb{B}, \lambda b.\overline{b}, \lambda b.\overline{b}\rangle$. You

should think of its two elements as infinite sequences of nonss, each being the nons of the other, one having value true, the other having value false.

This shows that the correct intuition about wander types is that, in the structure of elements, applications of the constructors should be labelled by the values of the function component. It is not necessary that all nodes are so decorated: the equations determine the value at a node from the values at its children; there need only be a label on some node along every infinite path. Finding a good programming paradigm that minimizes the need for decoration is one of the important challenges in implementing wander types in functional programming.

5 Reducing other data constructors

Wander types generalize several well-know constructions in type theory. As we have seen, they allow the definition of structures with very fine control on the branching behaviour: we can specify subtle kinds of well-foundedness properties.

They can, obviously, represent simple coinductive types by not using the functional component, for example defining it to be a constant. They can also represent simple inductive types, by having a function component that requires all paths to be finite. To do this, however, the target type must already have some well-founded structure, but it can be much simpler. For example, natural numbers are enough to obtain all finitary inductive types. In any case, some form of induction must be provided independently of wander types.

We can go further and represent mixed induction-coinduction, as described by Danielsson and Altenkirch [5]. A mixed inductive-coinductive type has two kinds of constructors, those that are required to be well-founded and those that are not so restricted. A very simple example is a type of infinite binary sequences where we forbid infinite consecutive ones. In other words, the sequence cannot be eventually constantly one. To realize this, we should have two unary constructors, one appending a Zero in front of a sequence, the other appending a One. The One constructor must be inductive, the Zero constructor is coinductive. We can realize this with wander types by just having a function component that counts the number of consecutive Ones:

$$\mathsf{Wander}\ \ \mathsf{type\ ZeroOne} : \mathsf{Set} \quad \mathsf{function\ countOnes} : \mathsf{ZeroOne} \to \mathbb{N}$$

$$\begin{array}{ll} \mathsf{Zero} : \mathsf{ZeroOne} \to \mathsf{ZeroOne} & \mathsf{One} : \mathsf{ZeroOne} \to \mathsf{ZeroOne} \\ \mathsf{countOnes}\,(\mathsf{Zero}\,x) = 0 & \mathsf{countOnes}\,(\mathsf{One}\,x) = \mathsf{countOnes}\,x + 1. \end{array}$$

with IR code: $\mathsf{zeroone} = \mathsf{rec}\,(\lambda c.\mathsf{element}\,0) + \mathsf{rec}\,(\lambda c.\mathsf{element}\,(c+1))$. In the original presentation by Danielsson and Altenkirch, arguments of constructors along which well-foundedness is not required are marked by an ∞ symbol. So the type of Zero is $\mathsf{ZeroOne}^{\infty} \to \mathsf{ZeroOne}$. See the article [5] for the formal rules.

A more interesting example of mixed induction-coinduction is the type of stream processors defined by Neil Ghani, Peter Hancock and Dirk Pattinson [10]–[12]. We want to define a type whose elements represent continuous functions from the type Stream_A of infinite sequences over A to Stream_B, for given types A and B. Such a function can perform two operations: read an element from the input stream or print an element in the output stream. We can specify it using two constructors. However, reading forever without producing any output should be forbidden because it leads to non-productive definitions. The reading constructor must therefore be inductive, while the writing constructor is coinductive.

We use an auxiliary type Tree_A of well-founded trees with branching type A, that is, whose nodes have a child for every element of A:

$$\begin{array}{ll} \mathsf{Inductive} & \mathsf{type\ Tree}_A : \mathsf{Set} \\ & \mathsf{leaf} : \mathsf{Tree}_A \\ & \mathsf{node} : (A \to \mathsf{Tree}_A) \to \mathsf{Tree}_A. \end{array}$$

The type of stream processors is now a wander type: the well-foundedness of the read constructor is specified by a function component to trees:

$$\mathsf{Wander}\ \ \mathsf{type\ StProc}_{A,B} : \mathsf{Set} \quad \mathsf{function\ tree} : \mathsf{StProc}_{A,B} \to \mathsf{Tree}_A$$

$$\begin{array}{l} \mathsf{read} : (A \to \mathsf{StProc}_{A,B}) \to \mathsf{StProc}_{A,B} \\ \mathsf{tree}\,(\mathsf{read}\,f) = \mathsf{node}\,(\lambda a.\mathsf{tree}\,(f\,a)) \end{array}$$

$$\begin{array}{l} \mathsf{write} : B \to \mathsf{StProc}_{A,B} \to \mathsf{StProc}_{A,B} \\ \mathsf{tree}\,(\mathsf{write}\,b\,p) = \mathsf{leaf}. \end{array}$$

The IR code for this wander type is:

$$\begin{aligned}&\mathsf{stproc}_{A,B} : \mathsf{IR}_{\mathsf{Tree}_A}\\ &\mathsf{stproc}_{A,B} = \ \mathsf{recurse}\, A\, (\lambda f.\mathsf{element}\,(\mathsf{node}\, f)) +\\ &\qquad\qquad\ \mathsf{choose}\, B\, (\lambda b.\mathsf{rec}\,(\lambda t.\mathsf{element}\,\mathsf{leaf})).\end{aligned}$$

Also in this case, in Danielsson and Altenkirch's implementation, non-well-foundedness of write was imposed by giving it the type $B \to \mathsf{StProc}^{\infty}_{A,B} \to \mathsf{StProc}_{A,B}$, while read is automatically well-founded. In the wander type implementation write is automatically non-well-founded, while well-foundedness of read is imposed by the condition that tree is well defined.

The interpretation of elements of $\mathsf{stproc}_{A,B}$ as continuous stream processors is given by the evaluation function:

$$\begin{aligned}&\mathsf{eval} : \mathsf{StProc}_{A,B} \to \mathsf{Stream}_A \to \mathsf{Stream}_B\\ &\mathsf{eval}\,(\mathsf{read}\, f)\,(a : s) = \mathsf{eval}\,(f\, a)\, s\\ &\mathsf{eval}\,(\mathsf{write}\, b\, p)\, s = b : \mathsf{eval}\, p\, s\end{aligned}$$

Obviously, we must justify the well-definedness of this function.

This is done by exploiting the finality of Stream_B together with initiality of Tree_A, that is, we use well-founded recursion on the value of tree to give a productive definition of the result.

Theorem 13 *There exists a unique function satisfying the two equations of* eval.

Proof. First generalize the definition of trees to $\mathsf{Tree}_{A,C}$: A-branching trees with leaves labelled by elements of C, where C is any type. It has constructors leaf_C and node_C similar to those of Tree_A, except that the leaf is labelled, that is, $\mathsf{leaf}_C : C \to \mathsf{Tree}_{A,C}$. In our specific case we choose $C = B \times \mathsf{StProc}_{A,B}$. We can overload the result type of tree to give more information in the leaf case:

$$\mathsf{tree}_B : \mathsf{StProc}_{A,B} \to \mathsf{Tree}_{A,B\times\mathsf{StProc}_{A,B}}$$

can be defined by well-founded recursion of the result of tree so that the following equations are satisfied:

$$\begin{aligned}&\mathsf{tree}_B\,(\mathsf{read}\, f) = \mathsf{node}_C\,(\lambda a.\mathsf{tree}_B\,(f\, a))\\ &\mathsf{tree}_B\,(\mathsf{write}\, b\, p) = \mathsf{leaf}_C\,\langle b, p\rangle.\end{aligned}$$

Formally, we would use an auxiliary function

$$\mathsf{tree}^{+}_B : \Pi t : \mathsf{Tree}_A.\Pi p : \mathsf{StProc}_{A,B}.t = \mathsf{tree}(p) \to \mathsf{Tree}_{A,B\times\mathsf{StProc}_{A,B}}$$

by recursion on t. Thinking forward to the representation as coinductive families given in Section 6, we see this as recursion on the index of the family.

Notice that it is not in general possible to define a function by structural recursion on a wander type, even when the target type of the function component is inductive, because there is no guarantee that the value of the recursive calls decreases. The case of tree is special, because the structure of its result closely mirrors that of the input.

An important observation, although not needed in this proof, is that tree_B is in fact an isomorphism.

We can then get three functions: the first produces the first entry in the output stream, the second gives the stream processor for the tail, and the third returns the unread part of the input:

$$\begin{aligned}&\mathsf{outh} : \mathsf{Tree}_{A,C} \to \mathsf{Stream}_A \to B,\\ &\mathsf{nextsp} : \mathsf{Tree}_{A,C} \to \mathsf{Stream}_A \to \mathsf{StProc}_{A,B},\\ &\mathsf{inputr} : \mathsf{Tree}_{A,C} \to \mathsf{Stream}_A \to \mathsf{Stream}_A.\end{aligned}$$

All three are easily defined by well-founded recursion on $\mathsf{Tree}_{A,C}$.

We finally obtain a coalgebra:

$$\begin{aligned}&\mathsf{evalco} : \mathsf{StProc}_{A,B} \times \mathsf{Stream}_A \to B \times \mathsf{StProc}_{A,B} \times \mathsf{Stream}_A\\ &\mathsf{evalco}\,\langle p, s\rangle = (\lambda t.\langle \mathsf{outh}\, t\, s, \mathsf{nextsp}\, t\, s, \mathsf{inputr}\, t\, s\rangle)\,(\mathsf{tree}_B\, p).\end{aligned}$$

There exists a unique morphism to the final coalgebra Stream_B. The reader can verify that this morphism must satisfy the equations of eval. □

We developed the previous proof in some detail, defining the function eval with a combination of recursion on Tree_A and corecursion on Stream_B. In practice, the user should be able to just give the definition of eval by the two equations and the system will be able to automatically check productivity/well-foundedness.

A natural but incorrect idea would be to use eval itself as the functional component of $\mathsf{StProc}_{A,B}$, in place of tree. Unfortunately, this would fail to impose a well-foundedness condition on the read constructor. The problem is similar to the one encountered with the InfNot type: for every function $f : \mathsf{Stream}_A \to \mathsf{Stream}_B$ we would have an element of $\mathsf{StProc}_{A,B}$ consisting of infinitely many occurrences of read and returning f as its evaluation.

Ghani, Hancock and Pattinson proved that the stream functions representable by elements of $\mathsf{StProc}_{A,B}$ are exactly the continuous ones. Here we see a slightly stronger result, that constructs directly a final coalgebra of $\Theta_{\mathsf{stproc}_{A,B}}$ from the set of all continuous functions. A continuous function from streams to streams has the property that every finite part of the output is determined by a finite part of the input. It will then have a *modulus of continuity*: a function determining how much of the input needs to be read to compute a given amount of output.

In the following the notation $s|n$ denotes the list of the first n elements of a stream s.

Definition 14 *Let* $f : \mathsf{Stream}_A \to \mathsf{Stream}_B$ *and* $m : \mathsf{Stream}_A \to \mathbb{N} \to \mathbb{N}$. *We say that* m is a modulus of continuity for f *if the following condition holds:*

$$
\begin{aligned}
&\mathsf{Continuity}(m, f) = \\
&\quad \forall s : \mathsf{Stream}_A . \forall n : \mathbb{N}. \\
&\qquad (\forall s' : \mathsf{Stream}_A .\ \ s'|(m\, s\, n) = s|(m\, s\, n) \to \\
&\qquad\qquad (f\, s')|(n+1) = (f\, s)|(n+1) \wedge (m\, s'\, n) = (m\, s\, n)) \\
&\qquad \wedge (m\, s\, (n+1)) \geq (m\, s\, n).
\end{aligned}
$$

We define the set of continuous functions as the pairs consisting of a function and a modulus of continuity:

$$\mathcal{SP}_{A,B} = \{\langle f, m\rangle \mid \mathsf{Continuity}(m, f)\}.$$

The definition of modulus of continuity contains three conditions. The first says that the first $n+1$ entries of the output are determined by the first $(m\, s\, n)$ entries of the input, that is, $(f\, s)|(n+1)$ is determined by $s|(m\, s\, n)$. So for any other stream s' beginning with the same $(m\, s\, n)$ elements, the function will produce the same $m+1$ elements as output. We also require that m is consistent, that is, it gives the same value $(m\, s\, n)$ to $(m\, s'\, n)$. The third condition states that m is monotone: to produce more output we need to read more input.

The set we just defined is a good model for stream processors: we can define reading and writing functions and their inverse. In the following definition we assume that A is non-empty and therefore there is a canonical element $s_0 : \mathsf{Stream}_A$. This is used in the definition of the action operation. The reader can verify that the definition does not depend on s_0: any stream would give the same result.

Definition 15 *The reading and writing operations of* $\mathcal{SP}_{A,B}$ *are defined by:*

$$
\begin{aligned}
&\mathsf{read} : (A \to \mathcal{SP}_{A,B}) \to \mathcal{SP}_{A,B} \\
&\mathsf{read}\, g = \langle f, m\rangle \quad \textit{where} \\
&\quad f\, (a : as) = \pi_1\, (g\, a)\, as \\
&\quad m\, (a : as)\, n = (\pi_2\, (g\, a)\, as\, n) + 1
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathsf{write} : B \to \mathcal{SP}_{A,B} \to \mathcal{SP}_{A,B} \\
&\mathsf{write}\, b\, \langle f', m'\rangle = \langle f, m\rangle \quad \textit{where} \\
&\quad f\, as = b : f'\, as \\
&\quad m\, as\, 0 = 0 \\
&\quad m\, as\, (n+1) = m'\, as\, n.
\end{aligned}
$$

A read operation will always need to get an element from the input before deciding what function to apply to the rest of the input; thus the modulus of continuity will be one plus the modulus of that function. A write operation produces a new output element b without reading anything; thus the argument of the modulus of continuity is decreased by one because we get b for free.

There is an inverse to the pair of functions read and write. Every continuous function is in the form of a reading or writing operation.

Definition 16 *The operation determining the input/output action associated with a continuous function is defined*

by:

$$\begin{array}{l}\mathsf{action} : \mathcal{SP}_{A,B} \to (A \to \mathcal{SP}_{A,B}) + (B \times \mathcal{SP}_{A,B}) \\ \mathsf{action}\,\langle f, m\rangle = \left\{\begin{array}{ll} \mathsf{inr}\,\langle b, \langle f', m'\rangle\rangle & \mathit{if}\ (m\ s_0\ 0) = 0 \\ \mathit{where}\ \ b = \mathsf{head}\,(f\ s_0) & \\ \qquad f'\ as = \mathsf{tail}\,(f\ as) & \\ \qquad m'\ as\ n = m\ as\ (n+1) & \\ \mathsf{inl}\,(\lambda a.\langle f'_a, m'_a\rangle) & \mathit{if}\ (m\ s_0\ 0) > 0 \\ \mathit{where}\ \ f'_a\ as = f\ (a{:}as) & \\ \qquad m'_a\ as\ n = (m\ (a{:}as)\ n) - 1. & \end{array}\right.\end{array}$$

The subtraction of 1 in the last line is well-defined since, under the condition of that branch and because of the consistency and monotonicity of m, $m\,(a{:}as)\,n$ is strictly positive. In fact, the condition for the branching is independent of the canonical element s_0 used to test it: $(m\ s_0\ 0) = 0$ if and only if $(m\ s\ 0) = 0$ for any stream s. In words: if the first element of the output of f on s_0 is determined without reading any element of s_0, the same is true for every other input stream s.

Lemma 17 *The operation* action *is the inverse of the pair of operations* read *and* write*, that is:*

$$\begin{array}{l}\mathsf{action} \circ [\mathsf{read}, \mathsf{write}] = \mathsf{id}_{(A\to\mathcal{SP}_{A,B})+B\times\mathcal{SP}_{A,B}} \\ [\mathsf{read}, \mathsf{write}] \circ \mathsf{action} = \mathsf{id}_{\mathcal{SP}_{A,B}}\end{array}$$

Proof. The two equalities are easily verified by unfolding the definitions and simplifying. □

Definition 18 *We can associate to every continuous function (with a continuity modulus) the tree of its "input readings":*

$$\begin{array}{l}\mathsf{tree} : \mathcal{SP}_{A,B} \to \mathsf{Tree}_A \\ \mathsf{tree}\,\langle f, m\rangle = \left\{\begin{array}{ll} \mathsf{leaf} & \mathit{if}\ (m\ s_0\ 0) = 0 \\ \mathsf{node}\,(\lambda a.\mathsf{tree}\langle f'_a, m'_a\rangle) & \mathit{if}\ (m\ s_0\ 0) > 0 \end{array}\right.\end{array}$$

where f'_a and m'_a are defined as for action*.*

Note that tree is well-defined, that is, the tree produced is well-founded, because m'_a decreases the value of $(m\,(a{:}as)\,n)$ and therefore every path through the tree is finite. More specifically, every maximal path through the tree can be extended to an element $s : \mathsf{Stream}_A$. The path has length $(m\ s\ 0)$ and is therefore finite.

Theorem 19 *The triple $\langle \mathcal{SP}_{A,B}, \mathsf{tree}, \mathsf{action}\rangle$ is a final coalgebra for $\Theta_{\mathsf{stproc}_{A,B}}$.*

Proof. First of all, let's spell out explicitly how $\Theta_{\mathsf{stproc}_{A,B}}$ operates and what are its coalgebras.

$$\begin{array}{l}\Theta_{\mathsf{stproc}_{A,B}} : \mathsf{Set}\downarrow\mathsf{Tree}_A \to \mathsf{Set}\downarrow\mathsf{Tree}_A \\ \Theta_{\mathsf{stproc}_{A,B}}\,\langle X, t\rangle = \langle (A \to X) + (B \times X), [\lambda f.\mathsf{node}\,(t \circ f), \lambda\langle b, x'\rangle.\mathsf{leaf}]\rangle.\end{array}$$

So a $\Theta_{\mathsf{stproc}_{A,B}}$-coalgebra is a triple $\langle X, t, \alpha\rangle$, where $\alpha : X \to (A \to X) + (B \times X)$ such that, for every $x : X$, if $(\alpha\ x) = \mathsf{inl}\ f$ then $(t\ x) = \mathsf{node}\,(t \circ f)$ and if $(\alpha\ x) = \mathsf{inr}\,\langle b, x'\rangle$ then $(t\ x) = \mathsf{leaf}$.

By definition and simple calculation, $\langle \mathcal{SP}_{A,B}, \mathsf{tree}, \mathsf{action}\rangle$ is such a coalgebra. We sketch the construction of the unique morphism $\check{\alpha}$ from any other coalgebra $\langle X, t, \alpha\rangle$ to it.

We can extend t in the same way we extended tree in Theorem 13 and define a similar stream coalgebra:

$$\begin{array}{l}t_B : X \to \mathsf{Tree}_{A,B\times X} \\ \mathsf{outh}_X : \mathsf{Tree}_{A,B\times X} \to \mathsf{Stream}_A \to B \\ \mathsf{nextsp}_\alpha : \mathsf{Tree}_{A,B\times X} \to \mathsf{Stream}_A \to X \\ \mathsf{inputr}_\alpha : \mathsf{Tree}_{A,B\times X} \to \mathsf{Stream}_A \to \mathsf{Stream}_A \\ \mathsf{evalco}_\gamma : X \times \mathsf{Stream}_A \to B \times X \times \mathsf{Stream}_A.\end{array}$$

This gives us a unique coalgebra morphism $\mathsf{eval}_\gamma : X \times \mathsf{Stream}_A \to \mathsf{Stream}_B$.

It is also easy to define a function $\mathsf{tl} : \mathsf{Tree}_A \to \mathsf{Stream}_A \to \mathbb{N}$ that computes the length of the branch of a well-founded tree determined by a stream.

Every element $x : X$ is mapped to $(\check{\alpha}\, x) = \langle f_x, m_x \rangle$ where $f_x = \lambda s.\mathsf{eval}_\gamma \langle x, s \rangle$ and m_x is defined by recursion on its second argument:

$$m_x\, s\, 0 = \mathsf{tl}\,(t\, x)\, s$$
$$m_x\, s\, (n+1) = \mathsf{tl}\,(t\, x)\, s + m_{(\mathsf{nextsp}_\alpha\,(t\,x)\,s)}\,(\mathsf{inputr}_\alpha\,(t\,x)\,s)\,n.$$

It can be verified that this gives the unique coalgebra morphism. □

6 Representation as coinductive families

In most well-known versions of type theory, wander types are not directly allowed. They cannot be defined in the type-theoretic system Coq [1], [22], although they can be realized in Agda [3].

If the target type of the functional component is *small*, there is a well-known encoding invented by Conor McBride. Induction-recursion can be represented by inductive families and wander types by coinductive families. I learnt this device directly from McBride. An accessible reference to it is in a post to the Coq mailing list, 23 October 2002, but the idea dates back to several years earlier. Recently it was generalized by Lorenzo Malatesta, Altenkirch, Ghani, Hancock and McBride to a proof of equivalence between small induction-recursion, dependent polynomials and indexed containers [14].

Suppose we want to define a wander type T with functional component $f : T \to D$. If D is a small type, then we can define a coinductive family $\overline{T} : D \to \mathsf{Set}$. The idea is that the elements of $(\overline{T}\, d)$ are those elements $t : T$ such that $(f\, t) = d$.

For example, the zigzag type can be defined as follows (consult Chapter 13 of the book by Yves Bertot and Pierre Casteran [1] for a good introduction to coinductive families).

$$\begin{array}{ll} \mathsf{CoInductive} & \mathsf{type}\ \overline{\mathsf{ZigZag}} : \mathbb{N} \times \mathbb{N} \to \mathsf{Set} \\ & \overline{\mathsf{leaf}} : \overline{\mathsf{ZigZag}}\,\langle 0, 0\rangle \\ & \overline{\mathsf{node}} : (n_1, m_1, n_2, m_2 : \mathbb{N}) \to \\ & \qquad \overline{\mathsf{ZigZag}}\,\langle n_1, m_1\rangle \to \overline{\mathsf{ZigZag}}\,\langle n_2, m_2\rangle \\ & \qquad \to \overline{\mathsf{ZigZag}}\,\langle m_1 + 1, n_2 + 1\rangle \end{array}$$

We can then define our wander type as the disjoint union of the family:

$$\begin{array}{ll} \mathsf{ZigZag} = \Sigma d : \mathbb{N} \times \mathbb{N}.\overline{\mathsf{ZigZag}}\, d : \mathsf{Set} & \\ \mathsf{zigs}\,\langle d, t\rangle = \pi_1\, d & \mathsf{zags}\,\langle d, t\rangle = \pi_2\, d \\ \mathsf{leaf} = \langle\langle 0, 0\rangle, \overline{\mathsf{leaf}}\rangle & \mathsf{node} = \lambda\langle\langle n_1, m_1\rangle, t_1\rangle, \langle\langle n_2, m_2\rangle, t_2\rangle. \\ & \qquad \langle\langle m_1 + 1, n_2 + 2\rangle, \overline{\mathsf{node}}\, n_1\, m_1\, n_2\, m_2\, t_1\, t_2\rangle. \end{array}$$

Similar translations are possible for every wander type: we can associate to every IR code a coinductive family whose disjoint union correctly encodes the final coalgebra of the associated functor in the slice category.

Definition 20 *Every IR code defines a constructor with strictly positive type for an inductive or coinductive family:*

$$\begin{array}{l} \mathsf{Constr} : \mathsf{IR}_D \to (D \to \mathsf{Set}) \to \mathsf{Set} \\ \mathsf{Constr}\,(\mathsf{element}\, d)\, F = F\, d \\ \mathsf{Constr}\,(\mathsf{choose}\, A\, u)\, F = \Pi a : A.\mathsf{Constr}\,(u\, a)\, F \\ \mathsf{Constr}\,(\mathsf{recurse}\, I\, v)\, F = \Pi \vec{d} : I \to D.\Pi \vec{i} : (\Pi i : I.F\,(\vec{d}\, i)).\mathsf{Constr}\,(v\, \vec{d})\, F. \end{array}$$

Read this definition as a syntactic construction of a term $(\mathsf{Constr}\, c\, F)$. In particular, we want to talk about its formal structure and specific occurrences of the variable F. This allows us to formulate the following simple result.

Lemma 21 . *For every $c : \mathsf{IR}_D$ and every family $F : D \to \mathsf{Set}$, the type $\mathsf{Constr}\, c\, F$ has the form*

$$\Pi x_1 : T_1 \ldots . \Pi x_n : T_n.F\, d[x_1, \ldots, x_n]$$

with F occurring only strictly positively in each of the T_is.

Proof. Simple verification by induction on the structure of c. □

The lemma empowers us to use the constructor in the definition of inductive and coinductive families in a system like Coq.

Definition 22 *For every* $c : \mathsf{IR}_D$*, define the two families of types indexed on* D*:*

$$\begin{array}{ll} \text{Inductive} & \mathsf{IndFam}_c : D \to \mathsf{Set} \\ & \mathsf{indfam}_c : \mathsf{Constr}\, c\, \mathsf{IndFam}_c \end{array}$$

$$\begin{array}{ll} \text{CoInductive} & \mathsf{CoIndFam}_c : D \to \mathsf{Set} \\ & \mathsf{coindfam}_c : \mathsf{Constr}\, c\, \mathsf{CoIndFam}_c. \end{array}$$

They are well-defined by the previous lemma.

Theorem 23 *The pairs* $\langle \Sigma d : D.\mathsf{IndFam}_c\, d, \pi_1 \rangle$ *and* $\langle \Sigma d : D.\mathsf{CoIndFam}_c\, d, \pi_1 \rangle$ *are the carriers of an initial algebra and a final coalgebra of* Θ_c*, respectively.*

Proof. We define the Θ_c-coalgebra structure for $\langle \Sigma d : D.\mathsf{CoIndFam}_c\, d, \pi_1 \rangle$. The construction of the Θ_c-algebra for $\langle \Sigma d : D.\mathsf{IndFam}_c\, d, \pi_1 \rangle$ is similar. The universal property for the coalgebra (algebra) can be derived from the corresponding universal property of the coinductive (inductive) family $\mathsf{CoIndFam}_c$ (IndFam_c).

We must define a function:

$$\psi : (\Sigma d : D.\mathsf{CoIndFam}_c\, d) \to \mathsf{F}_c\, (\Sigma d : D.\mathsf{CoIndFam}_c\, d)\, \pi_1.$$

We do it by recursion on c and case analysis on the elements of $(\mathsf{CoIndFam}_c\, d)$. These elements have the form $(\mathsf{coindfam}_c\, t_1 \cdots t_n)$, where $t_1, \ldots, t_n$ are terms of the types $T_1, \ldots, T_n$ from Lemma 21. This means that every element of $(\Sigma d : D.\mathsf{CoIndFam}_c\, d)$ has the form $\langle d[x_1 := t_1, \ldots, x_n := t_n], \mathsf{coindfam}_c\, t_1 \cdots t_n \rangle$. Since the first component is determined by the t_is, $(\Sigma d : D.\mathsf{CoIndFam}_c\, d)$ is isomorphic to $\Sigma x_1 : T_1 \ldots \Sigma x_{n-1} : T_{n-1}.T_n$. A further transformation can be performed in the case where $c = (\mathsf{recurse}\, I\, v)$ by replacing the components $\vec{d} : I \to D$ and $\vec{t} : (\Pi i : I.F\, (\vec{d}\, i))$ with a single $\vec{x} : I \to (\Sigma d : D.F\, d)$. Finally, comparison of Definitions 5 and 20 shows that $(\Sigma d : D.\mathsf{CoIndFam}_c\, d)$ is isomorphic to $\mathsf{F}_c\, (\Sigma d : D.\mathsf{CoIndFam}_c\, d)\, \pi_1$, giving the desired coalgebra. □

7 Future work

Several interesting lines of research are being pursued.

Although wander types can encode inductive types, we already need to have a type with a logically equivalent well-founded structure as the target of the function component. This is evident from the stream processor example: we needed to have a inductive type of trees to impose the constraint about termination of sequences of reading operations.

This raises the question of how much extra ordinal machinery is necessary to give wander types the full power of type theory. For example, an interesting question is in what conditions the type of continuous functions between two wander types can also be encoded as a wander type, generalizing the case of stream processors.

We are also interested in indexed typed extensions. We want to define and study coinductive-recursive indexed types (*wander families*), along the lines of similar work by Dybjer and Setzer on inductive-recursive families.

The definition of functions towards wander types should be improved, taking inspiration from the notion of guardedness for coinductive types [4]. As the encoding in coinductive types clearly shows, using the universal property of final coalgebras to define functions requires that every application of a constructor is labelled by the value of the functional component on the corresponding subterm. However, it is clear that such values can be computed from those of the sub-subterms. This means that it is only necessary that along every path from each node there is at least one constructor labelled by a function result. This will lead to a notion of definition by equations that are *guarded by values*.

References

[1] Y. Bertot and P. Castéran. *Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions*, Springer, 2004.

[2] A. Bove and V. Capretta, "Nested general recursion and partiality in type theory," In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, vol.2152 of *Lecture Notes in Computer Science*, pp.121–135, Springer-Verlag, 2001.

[3] A. Bove, P. Dybjer, and U. Norell, "A brief overview of Agda - a functional language with dependent types," In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, vol.5674 of *Lecture Notes in Computer Science*, pp.73–78, Springer, 2009.

[4] T. Coquand, “Infinite objects in type theory,” In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES’93*, vol.806 of *Lecture Notes in Computer Science*, pp.62–78, Springer-Verlag, 1993.

[5] N. Danielsson and T. Altenkirch, “Subtyping, declaratively,” In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction (MPC 2010)*, vol.6120 of *Lecture Notes in Computer Science*, pp.100–118. Springer Berlin/ Heidelberg, 2010.

[6] P. Dybjer, “A general formulation of simultaneous inductive-recursive definitions in type theory,” *Journal of Symbolic Logic*, vol.65, no.2, 2000.

[7] P. Dybjer and A. Setzer, “A finite axiomatization of inductive-recursive definitions,” In *Proceedings of TLCA 1999*, vol.1581 of *LNCS*, pp.129–146, Springer-Verlag, 1999.

[8] N. Ghani and P. Hancock, “An algebraic foundation and implementation of induction recursion and indexed induction recursion,” Draft paper, 2011.

[9] N. Ghani, P. Hancock, L. Malatesta, and A. Setzer, “Fibred data types,” Draft paper, 2012.

[10] N. Ghani, P. Hancock, and D. Pattinson, “Continuous functions on final coalgebras,” *Electr. Notes Theor. Comput. Sci.*, vol.164, no.1, pp.141–155, 2006. *Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science* (*CMCS 2006*).

[11] N. Ghani, P. Hancock, and D. Pattinson, “Continuous functions on final coalgebras,” *Electr. Notes Theor. Comput. Sci.*, vol.249, pp.3–18, 2009. *Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics* (*MFPS 2009*).

[12] N. Ghani, P. Hancock, and D. Pattinson, “Representations of stream processors using nested fixed points,” *Logical Methods in Computer Science*, vol.5, no.3, 2009.

[13] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[14] L. Malatesta, T. Altenkirch, N. Ghani, P. Hancock, and C. McBride, “Small induction recursion, indexed containers and dependent polynomials are equivalent,” Submitted for publication, 2012.

[15] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[16] P. Martin-Löf, “Mathematics of infinity,” In P. Martin-Löf and G. Mints, editors, *Conference on Computer Logic*, vol.417 of *Lecture Notes in Computer Science*, pp.146–197, Springer, 1988.

[17] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[18] E. Palmgren, “On universes in type theory,” In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, vol.36 of *Oxford Logic Guides*, pp.191–204, Oxford Science Publications, 1998. *Proceedings of a Congress Held in Venice*, Oct. 1995.

[19] M. Rathjen, “The strength of Martin-Löf type theory with a superuniverse. Part I,” *Arch. Math. Log.*, vol.39, no.1, pp.1–39, 2000.

[20] M. Rathjen, “The strength of Martin-Löf type theory with a superuniverse. Part II,” *Arch. Math. Log.*, vol.40, no.3, pp.207–233, 2001.

[21] M. Rathjen, E. R. Griffor, and E. Palmgren, “Inaccessibility in constructive set theory and type theory,” *Annals of Pure and Applied Logic*, vol.94, no.1–3, pp.181–200, 1998.

[22] The Coq Development Team. *The Coq Proof Assistant. Reference Manual. Version 8*. INRIA, 2011: http://coq.inria.fr/ refman/index.html

Venanzio CAPRETTA

Venanzio Capretta received a Laurea in Mathematics from the University of Padova, Italy, and a PhD in Computer Science from the Radboud University Nijmegen, The Netherlands. He worked as a researcher at INRIA Sophia Antipolis, France and the University of Ottawa, Canada. He is currently a lecturer and a member of the Functional Programming Lab at the School of Computer Science of the University of Nottingham, UK. His main fields of interest are mathematical logic, type theory, typed lambda calculus and functional programming. He has worked on the development of proof assistants, specifically with applications to algebraic algorithms, the logic of general recursion and higher-order abstract syntax. At present, the focus of his research in on coinduction and the formalization and logic of infinite data structures.