Theoretical Computer Science 412 (2011) 5006–5024

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Coalgebras in functional programming and type theory

Venanzio Capretta *

School of Computer Science, University of Nottingham, United Kingdom

A B S T R A C T

This is a survey article on the use of coalgebras in functional programming and type theory. It presents the basic theory underlying the implementation of coinductive types, families and predicates. It gives an overview of the application of corecursive methods to the study of general recursion, formal power series, tabulations of functions on inductive data. It also sketches some advanced topics in the study of the solutions to non-guarded corecursive equations and the design of non-standard type theory.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

During the last decade, coinductive methods have evolved from a specialised research topic to a central area of computer science. Their impact is particularly momentous in dependent functional programming and type theory. Lazy programming languages, foremost Haskell [42,41,55], allow us to manipulate infinite structures, for example infinite lists [32,36,74,75]. Type theory takes the potential still further, allowing coinductive families and predicates. Systems like Coq [72,13] and Agda [54,17] offer an elegant implementation of coinductive types, that yielded in a few years a rich blossoming of applications.

This article is a bird's-eye view of some of the most interesting results. We will sketch the broad design of the topics, formulate some of the most striking results and refer the reader to the appropriate literature for an in-depth treatment.

The first three sections set out the basic theory. In Section 2, we give a cursory overview of the fundamental concepts of type theory, the minimum indispensable to understand the way it is used in this article. Section 3 describes the categorical notions of final coalgebra and bisimulation and gives the most important results about them. Then Section 4 looks at how the categorical notions are implemented in type theory, also detailing the use of coinductive families, predicates and relations. The following sections are about extensions and applications of the basic theory: the solution of non-guarded corecursive equations (Section 5), the application to power series and calculus (Section 6), to the tabulation of functional programs (Section 7) and to recursion theory (Sections 8 and 9).

2. Type theory in a nutshell

In order to make the rest of the paper accessible to as many readers as possible, here is a condensed outline of (intensional Martin–Löf's) type theory [46–48,53,8,70]. It is a formal system with types and terms. Its statements are typing assertions of the form $t : T$, saying that a term t denotes some object which is a member of the set denoted by type T. Typing judgements have a sequence of assumptions, declaring that some variables denote generic elements of certain types:

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash t : T.$$

This judgement says: *"Assuming that the variable x_1 denotes an element of type A_1, that x_2 denotes an element of A_2, etc., we can conclude that the term t denotes an element of type T"*. Types can depend on terms, so that the variables x_i's can occur not

* Tel.: +44 0 115 8232342.
E-mail address: venanzio.capretta@nottingham.ac.uk.

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.04.024

only in t but also in T. Moreover a variable x_i may occur in the types of subsequent assumptions, A_j with $j > i$. For example, assume that we have defined a type family of vectors Vect depending on two parameters, the type of the vector elements and its length. Then the judgement

$$X : \mathsf{Set}, x_1 : X, x_2 : X \vdash \langle x_1, x_2 \rangle : \mathsf{Vect}\, X\, 2$$

says that, assuming that the variable X denotes a set and the variables x_1 and x_2 denote elements of it, then the pair $\langle x_1, x_2 \rangle$ denotes an element of the type $\mathsf{Vect}\, X\, 2$. After defining a vector concatenation function vapp, we can formulate the judgement

$$X : \mathsf{Set}, n_1 : \mathbb{N}, v_1 : \mathsf{Vect}\, X\, n_1, n_2 : \mathbb{N}, v_2 : \mathsf{Vect}\, X\, n_2 \vdash \mathsf{vapp}\, X\, n_1\, v_1\, n_2\, v_2 : \mathsf{Vect}\, X\, (n_1 + n_2)$$

saying that if v_1 denotes a vector of elements of X of length n_1 and v_2 a vector of length n_2, then their concatenation is a vector of length $n_1 + n_2$.

There are some basic types and some type constructors. Every type is defined by rules of three kinds: *introduction* rules tell us how to construct new elements of a type, *elimination* rules tell us how to define functions on the type and *reduction* rules tell us how to simplify applications of those functions.

There are types whose elements are themselves types, they are called *type universes*. In this article, we only use the universe Set, the type of *small* data types. In some formulations, for example in the type theory of Coq, there is a separate universe Prop for logical formulae. Here we use the Curry–Howard correspondence [40,70]: propositions are just data types whose elements are their proofs. So we will identify the universe Prop of logical formulae with Set. In some applications, higher levels of types may be needed, for example to implement some forms of polymorphism. These can be given by an ω-tower of type universes. For example, in Coq, Set and Prop are elements of Type_1, which is in turn an element of Type_2 and so on in an infinite ascending hierarchy. Set itself could be renamed Type_0 for uniformity. A more powerful and general approach is to add rules to generate new universes as they are needed, as shown by Palmgren [58] and generalised by Dybjier [26]. When we define type constructors below, we formulate them as operators inside Set, but they can equally be applied to higher universes.

For example, here are the rules for the two type constructors for (non-dependent) product and sum, defining the Cartesian product and disjoint sum of two given types A, B:

introduction
$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \qquad \frac{a : A}{\mathsf{inl}\, a : A + B} \qquad \frac{b : A}{\mathsf{inr}\, b : A + B}$$

elimination
$$\frac{c : A \times B}{\pi_1\, c : A} \qquad \frac{c : A \times B}{\pi_2\, c : B}$$

$$\frac{d : A + B \quad x : A \vdash u[x] : C \quad y : B \vdash v[y] : C}{\mathsf{case}\, d\, \mathsf{of}\, (\mathsf{inl}\, x \mapsto u[x] \mid \mathsf{inr}\, y \mapsto v[y]) : C}$$

reduction
$$(\pi_1\, \langle a, b \rangle) \rightsquigarrow a \quad (\pi_2\, \langle a, b \rangle) \rightsquigarrow b$$

$$\mathsf{case}\, (\mathsf{inl}\, a)\, \mathsf{of}\, (\mathsf{inl}\, x \mapsto u[x] \mid \mathsf{inr}\, y \mapsto v[y]) \rightsquigarrow u[a]$$
$$\mathsf{case}\, (\mathsf{inr}\, b)\, \mathsf{of}\, (\mathsf{inl}\, x \mapsto u[x] \mid \mathsf{inr}\, y \mapsto v[y]) \rightsquigarrow v[b].$$

Let us explain how these rules work. They are given in a simplified form that ignores contexts. In fact, each assertion should be given as the conclusion of a judgement. For example, the introduction rule for Cartesian product should be

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B},$$

where Γ can be any context. Since, in this case, the context is the same for all the judgements in the rule, we left it out. In other cases, namely when defining binders, the context changes: in the rules we only write the part of the context that is modified. For example, the full elimination rule for sums is

$$\frac{\Gamma \vdash d : A + B \qquad \Gamma, x : A \vdash u[x] : C \qquad \Gamma, y : B \vdash v[y] : C}{\Gamma \vdash \mathsf{case}\, d\, \mathsf{of}\, (\mathsf{inl}\, x \mapsto u[x] \mid \mathsf{inr}\, y \mapsto v[y]) : C}.$$

Here we have a common context Γ that does not change. In the second and third premises it is extended with the variables x and y, respectively. To stress the fact that the conclusion of these judgements depends on the new variable, we explicitly wrote it in square brackets: $u[x]$ is a term u in which the variable x may occur. In the conclusion, these local variables become bound and disappear from the context.

The rules should be a bit more general: the elimination type C for disjoint union could itself depend on a term of $A + B$. We gloss over these issues here: check the extensive literature for all the tricky details.

In the original formulation by Martin–Löf, there is a different form of judgement: *definitional equality*, which can be asserted when two terms denote the same value. Today, this is most often replaced by a reduction relation. Two terms are called *convertible* when they can be reduced to a common descendant. If we reduce a term as much as possible, we always

obtain, after a finite number of steps, a unique *normal form*: a term that cannot be further simplified. Convertible terms are interchangeable: in a judgement, a term can always be replaced by one of its reducts. In *extensional* versions of type theory, definitional equality can be stronger and, for example, identify two functions if their graphs are the same. The proof assistant NuPrl [20] is based on such formulation. However, the price to pay is undecidability of type checking.

Products and sums have dependent versions, Π-types and Σ-types. Suppose that A is a type and B is a family of types indexed over A, which means that $A : \mathsf{Set}$ and $B : A \to \mathsf{Set}$. Then we obtain two more types $\Pi x : A.B\,x$ and $\Sigma x : A.B\,x$. Intuitively, an element $f : \Pi x : A.B\,x$ is a function that maps every element $a : A$ to an element of $B\,a$; an element of $\Sigma x : A.B\,x$ is a pair $\langle a, b\rangle$ with $a : A$ and $b : B\,a$.

$$\text{introduction} \quad \frac{x : A \vdash b[x] : B\,x}{\lambda x.b[x] : \Pi x : A.B\,x} \qquad \frac{a : A \quad b : B\,a}{\langle a, b\rangle : \Sigma x : A.B\,x}$$

$$\text{elimination} \quad \frac{f : \Pi x : A.B\,x \quad a : A}{f\,a : B\,a}$$

$$\frac{c : \Sigma x : A.B}{\pi_1\,c : A} \qquad \frac{c : \Sigma x : A.B}{\pi_2\,c : B\,(\pi_1\,c)}$$

$$\text{reduction} \quad (\lambda x.b[x])\,a \rightsquigarrow b[a] \qquad (\pi_1\,\langle a, b\rangle) \rightsquigarrow a \qquad (\pi_2\,\langle a, b\rangle) \rightsquigarrow b.$$

The expression $b[a]$ indicates the substitution of every free occurrence of x with a, taking care of avoiding variable capture: bound variables in b should be renamed if they coincide with some free variable of a. Sometimes we use the notation $(x : A)(B\,x)$ for $\Pi x : A.B\,x$. Function types are Π types in which B does not depend on A: if $A, B : \mathsf{Set}$, then $A \to B = \Pi x : A.B$.

As an example, suppose we have a type $[\mathbb{N}]$ of lists of natural numbers and a binary relation $\mathsf{OrdPerm}\,l_1\,l_2$ stating that the list l_2 is an ordered permutation of l_1. Then we can give an exact type to a sorting function:

$$\mathsf{sort} : \Pi l : [\mathbb{N}].\Sigma l' : [\mathbb{N}].\mathsf{OrdPerm}\,l\,l'.$$

A term of this type is a program mapping a list l to a pair consisting of a list l' and a proof that l' is an ordered permutation of l. Therefore, correctness of the program is guaranteed by the type itself.

Finally, some basic types: the type of natural numbers $\mathbb{N}$, the type with only one element $\mathbf{1}$, and the type of Boolean values $\mathbb{B}$:

$$\text{introduction} \quad 0 : \mathbb{N} \quad \frac{n : \mathbb{N}}{\mathsf{S}n : \mathbb{N}} \qquad \bullet : \mathbf{1} \qquad \mathsf{true} : \mathbb{B} \quad \mathsf{false} : \mathbb{B}.$$

For elimination and reduction rules, we prefer to use recursive definitions by pattern matching rather than the eliminators [21,50]. So a function on natural numbers (let us say, the one computing the Fibonacci numbers) is defined by a sequence of pattern matching equations:

$$\begin{aligned} &\mathsf{fibpair} : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \\ &\mathsf{fibpair}\,0 = \langle 0, 1\rangle \\ &\mathsf{fibpair}\,\mathsf{S}n = \langle j, i + j\rangle \text{ where } \langle i, j\rangle = \mathsf{fibpair}\,n \end{aligned} \qquad \begin{aligned} &\mathsf{fib} : \mathbb{N} \to \mathbb{N} \\ &\mathsf{fib}\,n = \pi_1\,(\mathsf{fibpair}\,n). \end{aligned}$$

The reduction relation is then just the unfolding of the equations. (The where construction can be realised by abstraction and elimination rules): we could write the second equation for fibpair as

$$\mathsf{fibpair}\,\mathsf{S}n = (\lambda i.\lambda j.\langle j, i + j\rangle)\,((\lambda z.\langle \pi_1\,z, \pi_2\,z\rangle)\,(\mathsf{fibpair}\,n)).$$

Finally, there is an empty type $\mathbf{0}$, without any elements. Pattern matching is very easy on it: there are not patterns, so we can define a function from it to any other type without giving any equation. If $z : \mathbf{0}$, then $!z : A$ for any type A.

Our tour of the basic constructions of type theory will be completed by the introduction of inductive and coinductive types in Section 4.

3. Final coalgebras

Let us start with a short overview of the basic theory of final coalgebras and bisimilarity. These definitions and results are well-established: some of them date back to Lambek [45]; the main pillars were erected by Aczel and Mendler [4,3]. The exposition by Rutten and Turi [65,73] is a good introduction. In the last decade, the field has blossomed into many developments, as witnessed by the proceedings of the CMCS and CALCO conferences.

We will illustrate the general notions by instantiating them for the case of streams (infinite sequences) and infinitely branching trees. The abstract concepts are formulated in any category $\mathcal{C}$ and the main theorems are valid in general. However, because our interest is in functional programming and type theory, we favour the instantiations of the categorical constructions as Coq or Agda types and functions or as Haskell data structures and programs. Hence, for our purposes, it is enough to work in the category of sets, Set.

Haskell makes no distinction between inductive and coinductive types (initial algebras and final coalgebras). Therefore, an algebraically compact category [28,9] might be more suitable as its semantics than Set, but we carefully avoid using this duplicity of Haskell structures.

We want to study types of structured entities: every object can be decomposed into some tree-like frame. We will see that this means that those types are fixed points of a kind of functors called *containers* [1]. Trees have nodes of different *shape*, and each shape has branches stemming from various *positions*. Picture a functor on Set as some data constructor, defining a type of complex data objects containing elements of the argument type. Structured data are characterised by different kinds of shape; every shape contains several positions where substructures can be attached. Most functors on Set used in the literature are *strictly positive* (or even *finitary* or *polynomial*), that is, they may be represented as containers. A container is an endofunctor $F : \mathsf{Set} \to \mathsf{Set}$ of a specific form: $FX = \Sigma s : S.Ps \to X$ for some type S of *shapes* and family of types $P : S \to \mathsf{Set}$ of *positions*.

This means that an element of FX is determined by a *shape* s, which has a set of *positions* Ps, each containing an element of X.

As running examples, let us take two simple functors, depending on type parameters D and A, B respectively: $\mathsf{S}_D X = D \times X$ and $\mathsf{T}_{A,B} X = B + (A \to X)$. The shape type for S_D is D itself and every shape has just one position, so $Pd = \mathbf{1}$. Indeed $\mathsf{S}_D X \cong \Sigma d : D.\mathbf{1} \to X$ with the pair $\langle d, x \rangle$ corresponding to $\langle d, \lambda u.x \rangle$.

The shape type for $\mathsf{T}_{A,B}$ is $B + \mathbf{1}$. The shapes $(\mathsf{inl}\, b)$ have no positions and the shape $(\mathsf{inr}\, \bullet)$ has a position for every element of A.

$$\mathsf{T}_{A,B} X \cong \Sigma s : B + \mathbf{1}.(\mathsf{case}\, s\, \mathsf{of}\, (\mathsf{inl}\, b) \mapsto \mathbf{0} \mid (\mathsf{inr}\, \bullet) \mapsto A) \to X$$

with $(\mathsf{inl}\, b)$ corresponding to $\langle \mathsf{inl}\, b, \lambda z.!z \rangle$ and $(\mathsf{inr} f)$ corresponding to $\langle \mathsf{inr}\, \bullet, f \rangle$.

We study types of objects with a tree-like structure: the shapes are nodes and every position is the stemming point of a branch. Therefore, the leaves of the trees consist of the shapes without positions. Initial algebras comprise the well-founded structures while final coalgebras contain potentially non-well-founded structures. Since S_D has no shape with empty position set, the initial algebra is empty, while the final coalgebra consists of infinite sequences of elements of D. Both the initial algebra and final coalgebra of $\mathsf{T}_{A,B}$ contain trees with B-labelled leaves and A-branching nodes.

Definition 1. Let F be a functor, a *coalgebra* for F is pair $\langle A, \alpha \rangle$ of an object A and a morphism $\alpha : A \to FA$; $\langle A, \alpha \rangle$ is a *final F-coalgebra* if, for every coalgebra $\langle X, \xi : X \to FX \rangle$, there exists a unique coalgebra morphism f from $\langle X, \xi \rangle$ to $\langle A, \alpha \rangle$. We express this by a *corecursive diagram:*

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & FA \\ f \uparrow & & \uparrow Ff \\ X & \xrightarrow[\xi]{} & FX \end{array} \qquad \alpha \circ f = Ff \circ \xi.$$

We will talk separately of the property of *existence of solutions* and of *unicity of solutions to corecursive diagrams* for coalgebras that ensure that there is at least or at most one morphism f making the square above commute.

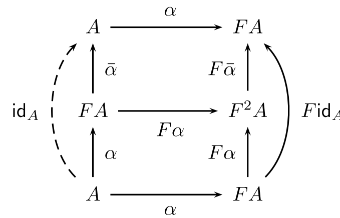
If F has a final coalgebra, it is unique up to isomorphism; it is usually denoted by $\langle \nu F, \mathsf{out}_F \rangle$. If we need to highlight the argument of the functor, we use ν as a binder and write $\nu X.FX$. The first final coalgebra we consider and by far the most used in the literature is, for every type D, $\mathbb{S}_D = \nu X.D \times X$, the coalgebra of streams. We will drop the subscript D when it is clear from the context. The components of the morphism part of the coalgebra are called head and tail: $\langle \mathsf{head}, \mathsf{tail} \rangle = \mathsf{out}_{\nu X.D \times X} : \mathbb{S}_D \to D \times \mathbb{S}_D$. This coalgebra is susceptible to a concrete representation: it is the set of infinite sequences of elements of D, so it is isomorphic to $\mathbb{N} \to D$ (we sidestep for the moment the issue of extensionality, we come back to it in Section 4). So head gives the first element of the sequence (it corresponds to $\lambda f.f\ 0$) and tail gives the stream of elements after the first (it correspond to $\lambda f.\lambda n.f\ (n+1)$). We use the notation ${}^{\mathfrak{h}}s$ for $(\mathsf{head}\, s)$, ${}^{\mathfrak{t}}s$ for $(\mathsf{tail}\, s)$, ${}^{n\mathfrak{t}}s$ for successive applications of tail and ${}^{\mathfrak{h}n}s$ for the nth element of s: ${}^{0\mathfrak{t}}s = s$, ${}^{(n+1)\mathfrak{t}}s = {}^{\mathfrak{t}}({}^{n\mathfrak{t}}s)$, ${}^{\mathfrak{h}n}s = {}^{\mathfrak{h}}({}^{n\mathfrak{t}}s)$. Therefore $\lambda s.{}^{\mathfrak{h}n}s$ corresponds to $\lambda f.f\ n$.

Theorem 1 (*Lambek's Lemma [45]*). *Final coalgebras are invertible.*

Proof. Define $\bar{\alpha} : FA \to A$ as the unique solution of the diagram

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & FA \\ \bar{\alpha} \uparrow & & \uparrow F\bar{\alpha} \\ FA & \xrightarrow[F\alpha]{} & F^2A \end{array} \qquad \alpha \circ \bar{\alpha} = F\bar{\alpha} \circ F\alpha.$$

We want to prove that $\bar{\alpha}$ is the inverse of α, that is, $\bar{\alpha} \circ \alpha = \mathsf{id}_A$ and $\alpha \circ \bar{\alpha} = \mathsf{id}_{FA}$. The following diagram shows that both id_A and $\bar{\alpha} \circ \alpha$ are coalgebra morphisms from $\langle A, \alpha \rangle$ to itself:



By the unicity property of final coalgebras, it must be that $\bar{\alpha} \circ \alpha = \mathsf{id}_A$.

Consequently, we also have that $\alpha \circ \bar{\alpha} = F\bar{\alpha} \circ F\alpha = F(\bar{\alpha} \circ \alpha) = F\mathsf{id}_A = \mathsf{id}_{FA}$. □

In the case of streams, the inverse of the final coalgebra $\langle \mathsf{head}, \mathsf{tail} \rangle$ is the algebra $\mathsf{cons} : D \times \mathbb{S} \to \mathbb{S}$ that prepends a new first element to a sequence. We write $d :: s$ for $\mathsf{cons}\ \langle d, s \rangle$.

Binary relations are represented in category theory through the notion of *span*. In the simplest version, a span is intuitively the object consisting of the ordered pairs of related elements, with the projection functions from that object.

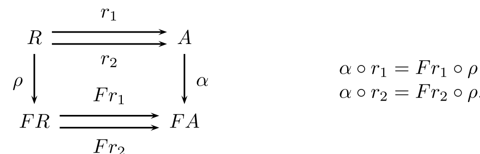
Definition 2. A *span* over an object A is a triple $\langle R, r_1, r_2 \rangle$ of an object R and two *projection* morphisms $r_1, r_2 : R \to A$.
Let F be a functor and $\langle R, r_1, r_2 \rangle$ a span. The *lifting of* $\langle R, r_1, r_2 \rangle$ *by* F is the span $\langle FR, Fr_1, Fr_2 \rangle$.

In the most direct representation of a relation, the idea is that R is the set of pairs of elements $\langle x, y \rangle$ that are in a relation, r_1 and r_2 are the first and second projection. Another, more comprehensive intuition, is that R is a set of proofs of the relation, with r_1 and r_2 specifying what elements the proof relates; so there may be different members of R that relate the same pair of elements (different proofs for the same instance of the relation).

If you think of FA as a set of structures containing elements in A, that is, if F is a container [1], then the lifting relates structures with the same shape and related elements in corresponding positions.

In all specific cases, we work in the category of sets, so we can represent relations directly rather than by a span. The functors will be simple enough that the lifting of a relation can be given explicitly.

Definition 3. Let $\langle A, \alpha \rangle$ be a coalgebra. A span $\langle R, r_1, r_2 \rangle$ is a *bisimulation* if there exists a morphism $\rho : R \to FR$ such that both r_1 and r_2 are coalgebra morphisms from $\langle R, \rho \rangle$ to $\langle A, \alpha \rangle$:



(The diagram here is used only to declare the type of the morphisms: we do not assume that it commutes. The only equalities are the ones stated on the right.)

The idea is that a relation is a bisimulation if, whenever two elements of A are related by it, then their images through α are related by the lifting. If you think of α as giving the structure of an element this says: if two elements are related, then they must have the same shape, with components in corresponding positions also related. This notion of bisimulation was first introduced by Park [59] and Milner [51] as a way of reasoning about processes. Similar concepts were developed earlier in other fields and substantial previous work prepared the background for its appearance. The survey article by Sangiorgi [67] tells the history of the idea. Aczel [2] adopted it as the appropriate notion of equality for non-well-founded sets. There are subtle differences between several notions of bisimulation that are not equivalent in full generality: recent work by Staton [71] investigates their correlations.

On $\mathbb{S}$ a bisimulation is a binary relation $\sim$ such that

$$\forall s_1, s_2 : \mathbb{S}. s_1 \sim s_2 \Rightarrow {}^{\natural}s_1 = {}^{\natural}s_2 \wedge {}^{t}s_1 \sim {}^{t}s_2.$$

Notice that $s_1 \sim s_2$ guarantees that corresponding elements in the infinite sequences defined by s_1 and s_2 are equal, that is, s_1 and s_2 are extensionally equal. In fact, by repeatedly applying the above property, we have that ${}^{\natural n}s_1 = {}^{\natural n}s_2$ for every n.

Definition 4. A coalgebra $\langle A, \alpha \rangle$ is said to satisfy the *coinduction principle* if every bisimulation $\langle R, r_1, r_2 \rangle$ on it has $r_1 = r_2$.

Intuitively, the coinduction principle states that the elements of A are completely characterised by their structure, which can be infinite. There is a well-known connection between finality of a coalgebra and the coinduction principle.

Theorem 2. *The principle of coinduction is equivalent to the unicity of solutions to corecursive diagrams.*

Proof. Assume that the principle of coinduction holds for a coalgebra $\langle A, \alpha\rangle$. We want to prove that it satisfies the unicity part of the definition of final coalgebra. Let f_1, f_2 be two solutions for the same corecursive diagram:

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & FA \\ f_1 \uparrow\uparrow f_2 & & Ff_1 \uparrow\uparrow Ff_2 \\ X & \xrightarrow[\xi]{} & FX \end{array} \qquad \begin{array}{l} \alpha \circ f_1 = Ff_1 \circ \xi \\ \alpha \circ f_2 = Ff_2 \circ \xi. \end{array}$$

The triple $\langle X, f_1, f_2\rangle$ is a bisimulation over $\langle A, \alpha\rangle$. By the coinduction principle, it must then be that $f_1 = f_2$, as desired.

Vice versa, assume that $\langle A, \alpha\rangle$ satisfies the unicity part of the definition of final coalgebra. We want to prove that it also satisfies the coinduction principle. Let $\langle R, r_1, r_2\rangle$ be a bisimulation over $\langle A, \alpha\rangle$, that is

$$\begin{array}{ccc} R & \overset{r_1}{\underset{r_2}{\rightrightarrows}} & A \\ \rho \downarrow & & \downarrow \alpha \\ FR & \overset{Fr_1}{\underset{Fr_2}{\rightrightarrows}} & FA \end{array} \qquad \begin{array}{l} \alpha \circ r_1 = Fr_1 \circ \rho \\ \alpha \circ r_2 = Fr_2 \circ \rho. \end{array}$$

Then both r_1 and r_2 are coalgebra morphisms from $\langle R, \rho\rangle$ to $\langle A, \alpha\rangle$. By unicity, it must be that $r_1 = r_2$, as desired. □

4. Coinductive types and families

In type theory, final coalgebras are ideally implemented as coinductive types. There are, however, some differences between the categorical theory and the type-theoretic realisation. In this presentation, we mostly follow the Coq type system (see Chapter 13 of [13]), but we use a notation more similar to that of Haskell or Agda.

Coinductive types are seen mostly as infinitary extensions of inductive ones. Therefore, they are conceived as being generated by constructors, albeit with the power of being infinitely iterated. Thus, the definitions of inductive and coinductive types are almost indistinguishable apart from a different introductory declaration. Here's an example:

$$\begin{array}{ll} \textbf{data} & \mathsf{List}\,(A : \mathsf{Set}) : \mathsf{Set} \\ & \mathsf{nil} : \mathsf{List}\,A \\ & \mathsf{cons} : A \to \mathsf{List}\,A \to \mathsf{List}\,A \end{array} \qquad \begin{array}{ll} \textbf{codata} & \mathsf{IList}\,(A : \mathsf{Set}) : \mathsf{Set} \\ & \mathsf{Inil} : \mathsf{IList}\,A \\ & \mathsf{Icons} : A \to \mathsf{IList}\,A \to \mathsf{IList}\,A. \end{array}$$

These two declarations both define a type constructor that, given a type A, produces a type, $(\mathsf{List}\,A)$ and $(\mathsf{IList}\,A)$, whose elements are either the empty list, nil and Inil, or are constructed by combining an element of A with another element in the type, $(\mathsf{cons}\,a\,x)$ and $(\mathsf{Icons}\,a\,y)$. The difference lies in the fact that the occurrences of cons must be well-founded and therefore, in this particular case, finite, while Icons can occur in non-well-founded branches an infinite number of times. This is encoded formally by different rules for acceptable recursive functions on List and IList.

The well-foundedness of inductive lists is characterised by allowing the definition of structural recursive programs on them. For example, here's the program that computes the length of a list:

$$\begin{array}{l} \mathsf{length} : \mathsf{List}\,A \to \mathbb{N} \\ \mathsf{length}\;\mathsf{nil} = 0 \\ \mathsf{length}\;(\mathsf{cons}\,a\,x) = \mathsf{S}\;(\mathsf{length}\,x). \end{array}$$

Here is how it works: for every possible constructor form that an element of $\mathsf{List}\,A$ can have, there is an equation giving the value of length. The function length can occur on the right-hand sides of the equations, provided that its argument is a proper subterm of the initial input.

The coinductive type $\mathsf{IList}\,A$ does not enjoy a similar method of definition by structural recursion: since the structure of its elements is not well-founded, it would not guarantee termination of the computation. The definition of a lazy list should be sound if it is *productive*: suppose a list l is defined in terms of itself, $l = \phi[l]$. This equation is said to be productive if the function $\lambda l.\phi[l]$ guarantees that all the entries of the list l will be generated by unfolding and reducing it. In practice, this property is enforced by a principle of definition by *guarded corecursion* [22], which we illustrate with a function computing the infinite list of numbers starting from a given one:

$$\begin{array}{l} \mathsf{from} : \mathbb{N} \to \mathsf{IList}\,\mathbb{N} \\ \mathsf{from}\;n = \mathsf{Icons}\;n\;(\mathsf{from}\;\mathsf{S}n). \end{array}$$

This definition is acceptable because the recursive call to from on the right-hand side of the equation occurs directly as an argument of the constructor Icons: we say that it is *guarded by the constructor*. This guarantees that a call to from always generates at least one constructor and the unfolding of the definition will progressively produce more and more of the structure of the list. Notice that now the complexity of the argument of the recursive call becomes irrelevant: it is $\mathsf{S}n$, which is more complex than the original input n.

The Coq system provides an operator for corecursion, cofix [34]: if C is a coinductive data type, we have a rule

$$\frac{f : A \to C \vdash \Phi[f] : A \to C}{\mathsf{cofix}\,(f \mapsto \Phi[f]) : A \to C}\,\mathcal{C}_{\mathsf{cofix}}(\Phi)$$

stating that the fixed point of Φ is well-defined, provided that the syntactic side requirement $\mathcal{C}_{\mathsf{cofix}}(\Phi)$ is satisfied. This side requirement is an encoding of the guardedness condition. See [33] for the formal definition.

As for recursive definitions, we will not use cofix directly, but rather define corecursive objects by equations, as we did for from. Equations are guarded if recursive calls occur in the right-hand side of each equation only under constructors. In the example, the recursive call from Sn occurs under the constructor lcons. We call objects defined in this way *cofixed points.*

Since unfolding of a coinductive object would lead to an infinite computation, coinductive types are treated lazily: cofixed points are expanded only when they occur under an operator that requires a constructor to compute its result. For example the from function above is not automatically reduced: $(\mathsf{from}\,0) \not\rightsquigarrow (\mathsf{lcons}\,0\,(\mathsf{from}\,1))$. On the other hand, if we give it as an argument to a function defined by cases on its constructor form, it will be lazily unfolded:

$$\begin{array}{l} \mathsf{isNil} : \mathsf{lList}\,A \to \mathbb{B} \\ \mathsf{isNil}\,\mathsf{lnil} = \mathsf{true} \\ \mathsf{isNil}\,(\mathsf{lcons}\,a\,l) = \mathsf{false} \end{array} \qquad \begin{array}{l} (\mathsf{isNil}\,(\mathsf{from}\,0)) \rightsquigarrow (\mathsf{isNil}\,(\mathsf{lcons}\,0\,(\mathsf{from}\,1))) \\ \qquad\qquad\qquad\quad \rightsquigarrow \mathsf{false}. \end{array}$$

Besides single coinductive types or type constructors, we can define coinductively whole families of types and predicates or relations over types. The definitions are like the ones above, except that both the signature of the definition and the type of the constructors can depend on index types.

For example, the following two declarations define two predicates on streams, one inductive, the other coinductive:

$$\begin{array}{ll} \textbf{data} & \mathsf{Eventually}\,(P : D \to \mathsf{Prop}) : \mathbb{S} \to \mathsf{Prop} \\ & \mathsf{now} : (d : D, s : \mathbb{S}) P d \to \mathsf{Eventually}_P\,(d :: s) \\ & \mathsf{later} : (d : D, s : \mathbb{S}) \mathsf{Eventually}_P\,s \to \mathsf{Eventually}_P\,(d :: s) \end{array}$$

$$\begin{array}{ll} \textbf{codata} & \mathsf{ForEver}\,(P : D \to \mathsf{Prop}) : \mathbb{S} \to \mathsf{Prop} \\ & \mathsf{forever} : (d : D, s : \mathbb{S}) P\,d \to \mathsf{ForEver}_P\,s \to \mathsf{ForEver}_P\,(d :: s). \end{array}$$

The first predicate is satisfied if the stream has one element for which P is true; a proof is a finite sequence of applications of the later constructor followed by now.

The second predicate is satisfied if P holds for all entries of the stream; a proof is an infinite sequence of applications of the forever constructor.

An interesting problem is how to combine the two to define a predicate InfOften which is satisfied if the stream contains an infinite number of entries satisfying P. This is a typical example of a mixed inductive–coinductive definition [23]. We would like to be able to define the predicate by giving it two constructors, one of which works inductively, the other coinductively.

$$\begin{array}{ll} (\textbf{co})\textbf{data} & \mathsf{InfOften}\,(P : D \to \mathsf{Prop}) : \mathbb{S} \to \mathsf{Prop} \\ & \mathsf{infNext} : (d : D, s : \mathbb{S}) P\,d \to \mathsf{InfOften}_P^\infty\,s \to \mathsf{InfOften}_P\,(d :: s) \\ & \mathsf{infLater} : (d : D, s : \mathbb{S}) \mathsf{InfOften}_P\,s \to \mathsf{InfOften}_P\,(d :: s). \end{array}$$

We decorated the recursive argument of infNext with an infinity sign to express that there may be a non-well-founded sequence of constructors under it. On the other hand, the recursive argument of infLater does not have such a decoration: it will behave as a regular inductive constructor. This means that in a proof of InfOften s, sequences of consecutive infLater steps must be well-founded and therefore finite, while there can be an infinite sequence of consecutive infNext steps. There still can be infinitely many infLater steps, as long as they alternate with infNext steps. See the work by Danielsson and Altenkirch to find out how such a definition is realised in Agda.

In the absence of mixed induction/coinduction, we need to use a clunkier construction to obtain the same result. We first need a function evRest: given a stream s and a proof of $\mathsf{Eventually}_P\,s$, it truncates s at the point where the entry satisfying P occurs.

$$\begin{array}{l} \mathsf{evRest} : (s : \mathbb{S}) \mathsf{Eventually}_P\,s \to \mathbb{S} \\ \mathsf{evRest}\,(d :: s)\,(\mathsf{now}\,d\,s\,p) = s \\ \mathsf{evRest}\,(d :: s)\,(\mathsf{later}\,d\,s\,h) = \mathsf{evRest}\,s\,h \end{array}$$

$$\begin{array}{ll} \textbf{codata} & \mathsf{InfOften}\,(P : D \to \mathsf{Prop}) : \mathbb{S} \to \mathsf{Prop} \\ & \mathsf{evInf} : (s : \mathbb{S}, h : \mathsf{Eventually}_P\,s) \mathsf{InfOften}_P\,(\mathsf{evRest}\,s\,h) \to \mathsf{InfOften}_P\,s. \end{array}$$

Coinductive types are not a perfect realisation of final coalgebras. They do not satisfy the coinduction principle. Bisimilar terms are not in general provably equal. This is no surprise, since checking bisimilarity may require the verification of the identity of an infinite number of substructures. For example, for streams, it would require checking that all the infinite entries are the same and would be equivalent to the extensional equality of functions on natural numbers. While Observational Type Theory [6] may eventually provide us with an extensional equality that does not spoil decidability of type checking, for the

moment we define bisimilarity as a binary relation on streams and other coinductive types. The actual final coalgebra would then be a setoid rather than a discrete type [37,39,38,10].

Bisimilarity is itself a coinductive relation. For example, here is its definition for streams:

$$\begin{array}{ll}\textbf{codata} & (\approx) : \mathbb{S} \to \mathbb{S} \to \mathsf{Prop} \\ & \mathsf{bisim} : (d : D; s_1, s_2 : \mathbb{S}) s_1 \approx s_2 \to (d :: s_1) \approx (d :: s_2).\end{array}$$

5. Streams and corecursive equations

We have seen that functions on streams can be defined by equations that satisfy the guardedness condition. For example, here is a definition for the operator interleaving the elements of two streams (the operator $(\ltimes)$ binds more strongly than $(::)$):

$$\begin{array}{l}(\ltimes) : \mathbb{S} \to \mathbb{S} \to \mathbb{S} \\ s_1 \ltimes s_2 = {}^{\mathsf{h}}s_1 :: s_2 \ltimes {}^{\mathsf{t}}s_1.\end{array}$$

The recursive call to $\ltimes$ occurs in the second argument of the top-level constructor $::$, so the equation satisfies the guardedness condition. We can also define several functions simultaneously, provided that each equation satisfies the guardedness condition with respect to all the functions. Although the following definition is not acceptable according to this criterion:

$$\begin{array}{ll}\mathsf{even} : \mathbb{S} \to \mathbb{S} & \mathsf{odd} : \mathbb{S} \to \mathbb{S} \\ \mathsf{even}\, s = {}^{\mathsf{h}}s :: \mathsf{odd}\, {}^{\mathsf{t}}s & \mathsf{odd}\, s = \mathsf{even}\, {}^{\mathsf{t}}s\end{array}$$

it can easily be fixed by changing the second equation to $\mathsf{odd}\, s = {}^{\mathsf{h}1}s :: \mathsf{odd}({}^{2\mathsf{t}}s)$.

However, in some cases we may have equations that are not guarded, but that nevertheless have a unique solution. Partial algorithms to determine whether a corecursive equation has solutions were given in [27,76,64]. The problem in general is undecidable. Rosu [63] proved this for equations on bit streams. The undecidability of pure stream equations (polymorphic on the type of their components) was recently proved by Florent Balestrieri [7].

Let us illustrate the problem of solving corecursive equations with an example that does not satisfy the guardedness condition or any of its obvious extensions.

Example 1.

$$\begin{array}{l}\psi : \mathbb{S} \to \mathbb{S} \\ \psi\ s = {}^{\mathsf{h}}s :: \mathsf{even}(\psi(\mathsf{odd}\, {}^{\mathsf{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\, {}^{\mathsf{t}}s)).\end{array}$$

When we apply this function to the stream $\mathsf{nat} = \mathsf{from}\, 0$ containing all natural numbers, we obtain: $\psi\ \mathsf{nat} = 0 :: 2 :: 5 :: 12 :: 25 :: 52 :: 105 :: 212 :: 425 :: 852 :: 1705 :: 3412 :: 6825 :: 13652 :: 27305 :: 54612 :: 109225 :: 218452 :: 436905 :: 873812 :: \ldots$.

The recursive calls to ψ occur under the application of the operators even and odd. These operators do not preserve productivity, that is, they do not generate an element of the output stream for every element read from the input stream. Therefore, we cannot be immediately sure that this equation correctly defines a infinite stream of values for every input. However, we will show that it does in the next proposition. Note that it would be enough to swap the roles of even and odd and the equation would not have a unique solution anymore.

Proposition 1. *The equation in Example 1 has at most one solution.*

Proof. Suppose ψ_1 and ψ_2 are two solutions to the equation. We want to prove that they are extensionally equal. Towards that goal, we define an inductive relation on $\mathbb{S}$, we prove that it is a bisimulation and we show that equality of the two functions follows by the coinduction principle.

The relation $\sim$ on $\mathbb{S}$ is inductively generated by the following rules:

$$\frac{s : \mathbb{S}}{\psi_1\, s \sim \psi_2\, s}(R0) \qquad \frac{x_1, x_2, y_1, y_2 : \mathbb{S} \quad x_1 \sim x_2 \quad y_1 \sim y_2}{\mathsf{even}\, x_1 \ltimes \mathsf{odd}\, y_1 \sim \mathsf{even}\, x_2 \ltimes \mathsf{odd}\, y_2}(R1).$$

Let us show that $\sim$ is a bisimulation. Assume that $s_1 \sim s_2$, we need to prove that ${}^{\mathsf{h}}s_1 = {}^{\mathsf{h}}s_2$ and ${}^{\mathsf{t}}s_1 \sim {}^{\mathsf{t}}s_2$. We proceed by induction on the generation of $\sim$. We have two cases according to the last rule used in the derivation of $s_1 \sim s_2$:

- If the last rule used was R0, then it must be $s_1 = \psi_1\, s$ and $s_2 = \psi_2\, s$ for some s. Then, because ψ_1 and ψ_2 satisfy the equation in Example 1, we have

  $$\begin{array}{l}s_1 = \psi_1\, s = {}^{\mathsf{h}}s :: \mathsf{even}(\psi_1(\mathsf{odd}\, {}^{\mathsf{t}}s)) \ltimes \mathsf{odd}(\psi_1(\mathsf{even}\, {}^{\mathsf{t}}s)) \\ s_2 = \psi_2\, s = {}^{\mathsf{h}}s :: \mathsf{even}(\psi_2(\mathsf{odd}\, {}^{\mathsf{t}}s)) \ltimes \mathsf{odd}(\psi_2(\mathsf{even}\, {}^{\mathsf{t}}s)).\end{array}$$

  So ${}^{\mathsf{h}}s_1 = {}^{\mathsf{h}}s = {}^{\mathsf{h}}s_2$ and

  $$\begin{array}{l}{}^{\mathsf{t}}s_1 = \mathsf{even}(\psi_1(\mathsf{odd}\, {}^{\mathsf{t}}s)) \ltimes \mathsf{odd}(\psi_1(\mathsf{even}\, {}^{\mathsf{t}}s)), \\ {}^{\mathsf{t}}s_2 = \mathsf{even}(\psi_2(\mathsf{odd}\, {}^{\mathsf{t}}s)) \ltimes \mathsf{odd}(\psi_2(\mathsf{even}\, {}^{\mathsf{t}}s)).\end{array}$$

By R0 we have that

$$\psi_1(\mathsf{odd}\,{}^{\mathrm{t}}s) \sim \psi_2(\mathsf{odd}\,{}^{\mathrm{t}}s) \quad \text{and} \quad \psi_1(\mathsf{even}\,{}^{\mathrm{t}}s) \sim \psi_2(\mathsf{even}\,{}^{\mathrm{t}}s),$$

therefore ${}^{\mathrm{t}}s_1 \sim {}^{\mathrm{t}}s_2$ by R1 (with instantiations $x_1 := \psi_1(\mathsf{odd}\,{}^{\mathrm{t}}s), x_2 := \psi_2(\mathsf{odd}\,{}^{\mathrm{t}}s), y_1 := \psi_1(\mathsf{even}\,{}^{\mathrm{t}}s), y_2 := \psi_2(\mathsf{even}\,{}^{\mathrm{t}}s)$).

- If the last rule used was R1, then there must be streams x_1, x_2, y_1, y_2 such that $s_1 = \mathsf{even}\,x_1 \ltimes \mathsf{odd}\,y_1$ and $s_2 = \mathsf{even}\,x_2 \ltimes \mathsf{odd}\,y_2$ with $x_1 \sim x_2$ and $y_1 \sim y_2$. By induction hypothesis we have that

$$^{\natural}x_1 = {}^{\natural}x_2, \quad {}^{\mathrm{t}}x_1 \sim {}^{\mathrm{t}}x_2; \qquad {}^{\natural}y_1 = {}^{\natural}y_2, \quad {}^{\mathrm{t}}y_1 \sim {}^{\mathrm{t}}y_2.$$

Then we have, by definition of $\ltimes$, even and odd,

$$^{\natural}s_1 = {}^{\natural}(\mathsf{even}\,x_1 \ltimes \mathsf{odd}\,y_1) = {}^{\natural}x_1 \overset{IH}{=} {}^{\natural}x_2 = {}^{\natural}(\mathsf{even}\,x_2 \ltimes \mathsf{odd}\,y_2) = {}^{\natural}s_2$$

and

$$\begin{aligned} &{}^{\mathrm{t}}s_1 = {}^{\mathrm{t}}(\mathsf{even}\,x_1 \ltimes \mathsf{odd}\,y_1) = \mathsf{odd}\,y_1 \ltimes {}^{\mathrm{t}}(\mathsf{even}\,x_1) = \mathsf{even}\,{}^{\mathrm{t}}y_1 \ltimes \mathsf{odd}\,{}^{\mathrm{t}}x_1 \\ &\text{and} \quad {}^{\mathrm{t}}s_2 = {}^{\mathrm{t}}(\mathsf{even}\,x_2 \ltimes \mathsf{odd}\,y_2) = \mathsf{even}\,{}^{\mathrm{t}}y_2 \ltimes \mathsf{odd}\,{}^{\mathrm{t}}x_2. \end{aligned}$$

But we know that ${}^{\mathrm{t}}y_1 \sim {}^{\mathrm{t}}y_2$ and ${}^{\mathrm{t}}x_1 \sim {}^{\mathrm{t}}x_2$ by induction hypothesis, so ${}^{\mathrm{t}}s_1 \sim {}^{\mathrm{t}}s_2$ by R1.

We conclude that $\sim$ is a bisimulation and, by the coinduction principle, every time $s_1 \sim s_2$ we can deduce that $s_1 = s_2$.

Since, for every s, $\psi_1\,s \sim \psi_2\,s$ by R0, we conclude that $\psi_1\,s = \psi_2\,s$ and therefore ψ_1 and ψ_2 are extensionally equal. □

We have shown that the solution of the equation, if it exists, must be unique. Existence needs to be proved with a different method. This technique can be generalised to an algorithm that takes the formal encoding of a corecursive equation as input and generates the rules of the bisimulation as output [19]. The algorithm can terminate with a positive answer: it generates a finite number of rules, it guarantees that they define a bisimulation and therefore unicity obtains. If the algorithm terminates with a negative answer, we cannot draw any conclusion about the unicity of the solution. The algorithm may also run forever. In this case it generates an infinite list of rules which in fact define an inductive bisimulation, but we will never know it because we cannot decide whether the algorithm diverges.

Let us look at how we can prove both existence and unicity of solutions by associating a coalgebra to Example 1. The general idea is that we need an *ad hoc* data type to represent all the possible unfoldings of the right-hand side of the equation. It is a set of trees where the leaves represent applications of ψ and the nodes represent the interleaving of the even entries of the first child with the odd entries of the second.

$$\begin{aligned} &\textbf{data}\ \mathbb{T}_\psi : \mathsf{Set} \\ &\quad \mathsf{leaf}_\psi : \mathbb{S} \to \mathbb{T}_\psi \\ &\quad \mathsf{node}_\psi : \mathbb{T}_\psi \to \mathbb{T}_\psi \to \mathbb{T}_\psi. \end{aligned}$$

We then define a coalgebra on this set, guided by the equation:

$$\begin{aligned} &\tau = \langle \tau_1, \tau_2 \rangle : \mathbb{T}_\psi \to D \times \mathbb{T}_\psi \\ &\tau\ (\mathsf{leaf}_\psi\ s) = \langle {}^{\natural}s, \mathsf{node}_\psi\ (\mathsf{leaf}_\psi\ (\mathsf{odd}\,{}^{\mathrm{t}}s))\ (\mathsf{leaf}_\psi\ (\mathsf{even}\,{}^{\mathrm{t}}s)) \rangle \\ &\tau\ (\mathsf{node}_\psi\ t_1\ t_2) = \langle \tau_1\,t_1, \mathsf{node}_\psi\ (\tau_2\,t_2)\ (\tau_2\,t_1) \rangle. \end{aligned}$$

There exists a unique coalgebra morphism from $\langle \mathbb{T}_\psi, \tau \rangle$ to $\langle \mathbb{S}, \langle {}^{\natural}-, {}^{\mathrm{t}}- \rangle \rangle$, let us call it $[\![\tau]\!]$. Now we define $\psi\ s = [\![\tau]\!](\mathsf{leaf}_\psi\ s)$ and we are going to show that it satisfies the recursive equation in Example 1. This is done by exploiting the unicity property of $[\![\tau]\!]$ as a coalgebra morphism. Let us define another function from trees to streams:

$$\begin{aligned} &f : \mathbb{T}_\psi \to \mathbb{S} \\ &f\ (\mathsf{leaf}_\psi\ s) = {}^{\natural}s :: \mathsf{even}(\psi(\mathsf{odd}\,{}^{\mathrm{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\,{}^{\mathrm{t}}s)) \\ &f\ (\mathsf{node}_\psi\ t_1\ t_2) = \mathsf{even}([\![\tau]\!]t_1) \ltimes \mathsf{odd}([\![\tau]\!]t_2). \end{aligned}$$

We prove that f is also a coalgebra morphism, using the definition of ψ and the fact that ${}^{\mathrm{t}}[\![\tau]\!]t = [\![\tau]\!](\tau_2\,t)$ because $[\![\tau]\!]$ is a coalgebra morphism:

$$\begin{aligned} f\ (\mathsf{leaf}_\psi\ s) &= {}^{\natural}s :: \mathsf{even}(\psi(\mathsf{odd}\,{}^{\mathrm{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\,{}^{\mathrm{t}}s)) \\ &= {}^{\natural}s :: \mathsf{even}([\![\tau]\!](\mathsf{leaf}_\psi\ (\mathsf{odd}\,{}^{\mathrm{t}}s))) \ltimes \mathsf{odd}([\![\tau]\!](\mathsf{leaf}_\psi\ (\mathsf{even}\,{}^{\mathrm{t}}s))) \\ &= {}^{\natural}s :: f(\mathsf{node}_\psi\ (\mathsf{leaf}_\psi\ (\mathsf{odd}\,{}^{\mathrm{t}}s)))\ (\mathsf{leaf}_\psi\ (\mathsf{even}\,{}^{\mathrm{t}}s)) \\ f\ (\mathsf{node}_\psi\ t_1\ t_2) &= \mathsf{even}([\![\tau]\!]t_1) \ltimes \mathsf{odd}([\![\tau]\!]t_2) \\ &= {}^{\natural}[\![\tau]\!]t_1 :: \mathsf{odd}([\![\tau]\!]t_2) \ltimes {}^{\mathrm{t}}\mathsf{even}([\![\tau]\!]t_1) \\ &= \tau_1\,t_1 :: \mathsf{even}({}^{\mathrm{t}}[\![\tau]\!]t_2) \ltimes \mathsf{odd}({}^{\mathrm{t}}[\![\tau]\!]t_1) \\ &= \tau_1\,t_1 :: \mathsf{even}([\![\tau]\!](\tau_2\,t_2)) \ltimes \mathsf{odd}([\![\tau]\!](\tau_2\,t_1)) \\ &= \tau_1\,t_1 :: f\ (\mathsf{node}_\psi\ (\tau_2\,t_2)\ (\tau_2\,t_1)). \end{aligned}$$

So it must be $f = [\![\tau]\!]$. In particular, $\psi\ s = [\![\tau]\!](\mathsf{leaf}_\psi\ s) = f\ (\mathsf{leaf}_\psi\ s) = {}^{\natural}s :: \mathsf{even}(\psi(\mathsf{odd}\,{}^{\mathrm{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\,{}^{\mathrm{t}}s))$, that is, ψ satisfies the corecursive equation.

On the other hand, if ψ' is a solution of the recursive equation, we can use it to define a coalgebra morphism from $\mathbb{T}_\psi$ to $\mathbb{S}$ like this:

$$\begin{array}{l} f_{\psi'} : \mathbb{T}_\psi \to \mathbb{S} \\ f_{\psi'}(\mathsf{leaf}_\psi\, s) = \psi'\, s \\ f_{\psi'}(\mathsf{node}_\psi\, t_1\, t_2) = \mathsf{even}(f_{\psi'}\, t_1) \ltimes \mathsf{odd}(f_{\psi'}\, t_2). \end{array}$$

It must be $f_{\psi'} = [\![\tau]\!]$ and therefore $\psi' = \psi$.

This technique is related to work by Rutten and collaborators on complete sets of co-operations and behavioural differential equations [66,68,43]. It gives a stronger result than the bisimulation construction: it ensures existence besides unicity of the solution.

However, so far no uniform way to construct the *ad hoc* type has been given in a form general enough to cover all the examples for which unicity can be established independently. In some other cases, defining an appropriate data type and a coalgebra on it is less straightforward than above, while the method of proving unicity of solutions using bisimulations still works nicely.

6. Formal power series

There is an engaging application of the coinductive theory of streams to calculus ([60,66]). We see a formal power series as a stream of real numbers. Let us say that f is an analytic function whose Taylor series is

$$f(x) = \sum_{i=0}^{\infty} a_i x^i.$$

We represent f as the stream of coefficients $\langle a_0, a_1, a_2, \ldots\rangle$. Standard operators and theorems on power series can be formulated in terms of coinductive principles on streams. Let us here give a concise summary of the extensive development of this idea presented by Jan Rutten in [66]. First notice that the tail operation on streams has a very similar role to that of the derivative on analytical functions and power series. For this reason Rutten uses the terms *initial value* and *derivative* for the head and tail functions. Although the tail is not exactly the formal derivative of a power series in the standard sense, there is a simple correspondence between them, the Laplace–Carson transform. We will, however, continue to call the tail a tail; instead, we notice that the analytical derivative gives rise to a distinct coalgebra (we use $\cdot$ for real number multiplication):

$$\begin{array}{l} \mathsf{D} : \mathbb{S}_\mathbb{R} \to \mathbb{S}_\mathbb{R} \\ \mathsf{D}\langle s_i\rangle_{i\in\mathbb{N}} = \langle (i+1)\cdot s_{i+1}\rangle_{i\in\mathbb{N}}. \end{array}$$

Theorem 3. *The coalgebra* $\langle \mathsf{head}, \mathsf{D}\rangle$ *is final for the functor* $FX = \mathbb{R}\times X$.

Proof. This result may seem surprising at first: do we not already know that $\langle \mathsf{head}, \mathsf{tail}\rangle$ is the final coalgebra and it has to be unique up to isomorphism? Certainly, but this does not contradict the theorem: it just tells us that $\langle \mathsf{head}, \mathsf{D}\rangle$ is isomorphic to $\langle \mathsf{head}, \mathsf{tail}\rangle$. This isomorphism is known as the *Laplace–Carson transform*:

$$\begin{array}{ll} \mathsf{LC} : \mathbb{S} \to \mathbb{S} & \mathsf{LC}^{-1} : \mathbb{S} \to \mathbb{S} \\ \mathsf{LC}\,\langle s_i\rangle_{i\in\mathbb{N}} = \langle i!\cdot s_i\rangle_{i\in\mathbb{N}} & \mathsf{LC}\,\langle t_i\rangle_{i\in\mathbb{N}} = \langle t_i/i!\rangle_{i\in\mathbb{N}}. \end{array}$$

It is trivial to verify that these functions are coalgebra morphisms between $\langle \mathsf{head}, \mathsf{D}\rangle$ and $\langle \mathsf{head}, \mathsf{tail}\rangle$ and they are inverses of each other. □

It is not often appreciated that we can have isomorphic but quite differently defined final coalgebras. In this specific case, it means that we can use coinductive methods to define and reason about power series in terms of their derivatives.

For example, standard operations on analytic functions can be defined by exploiting the well-known rules of derivatives. Multiplication has the derivation rule: $(f\times g)' = f'\times g + f\times g'$ or, in our notation, $D\,(s_1\times s_2) = (D\,s_1)\times s_2 + s_1\times (D\,s_2)$. This is just a corecursive equation of the kind we have been studying and can easily be turned into a coalgebra on a type of codes for its unfoldings:

$$\begin{array}{l} \textbf{data}\ \mathbb{T}_\times : \mathsf{Set} \\ \quad \mathsf{leaf} : \mathbb{S} \to \mathbb{T}_\times \\ \quad \mathsf{node}_+ : \mathbb{T}_\times \to \mathbb{T}_\times \to \mathbb{T}_\times \\ \quad \mathsf{node}_\times : \mathbb{T}_\times \to \mathbb{T}_\times \to \mathbb{T}_\times \\ \tau : \mathbb{T}_\times \to \mathbb{R}\times\mathbb{T}_\times \\ \tau\ (\mathsf{leaf}\, s) = \langle {}^{\natural}s, \mathsf{leaf}\,(\mathsf{D}s)\rangle \\ \tau\ (\mathsf{node}_+\, t_1\, t_2) = \langle x+y, \mathsf{node}_+\, t_1'\, t_2'\rangle \\ \quad \text{if } \langle x, t_1'\rangle = \tau\, t_1 \text{ and } \langle y, t_2'\rangle = \tau\, t_2 \\ \tau\ (\mathsf{node}_\times\, t_1\, t_2) = \langle x\cdot y, \mathsf{node}_+\,(\mathsf{node}_\times\, t_1'\, t_2)\,(\mathsf{node}_\times\, t_1\, t_2')\rangle \\ \quad \text{if } \langle x, t_1'\rangle = \tau\, t_1 \text{ and } \langle y, t_2'\rangle = \tau\, t_2. \end{array}$$

By the previous finality theorem there exists a unique coalgebra morphism $[\![\tau]\!]$ from τ to $\langle \mathsf{head}, \mathsf{D}\rangle$. It is easy to prove that $[\![\tau]\!]\,(\mathsf{node}_+\, t_1\, t_2)$ is just the pointwise addition $[\![\tau]\!]\, t_1 + [\![\tau]\!]\, t_2$. Define $s_1 \times s_2 = [\![\tau]\!]\,(\mathsf{node}_\times\,(\mathsf{leaf}\, s_1)\,(\mathsf{leaf}\, s_2))$ and the derivation rule is verified.

In a similar style, other operations on power series can be easily defined by giving their constant coefficient (head) and their derivation rule:

$$\begin{array}{lll} \text{inverse:} & {}^{\natural}(s^{-1}) = ({}^{\natural}s)^{-1} & \mathsf{D}(s^{-1}) = -\mathsf{D}s \times s^{-1} \times s^{-1} \\ \text{square root:} & {}^{\natural}\sqrt{s} = \sqrt{{}^{\natural}s} & \mathsf{D}\sqrt{s} = \mathsf{D}s \times (2\sqrt{s})^{-1} \\ \text{composition:} & {}^{\natural}(s \circ t) = {}^{\natural}s & \mathsf{D}(s \circ t) = \mathsf{D}t \times (\mathsf{D}s \circ t) \end{array}$$

with the conditions that ${}^{\natural}s \neq 0$ for the inverse, ${}^{\natural}s \geq 0$ for the square root, and ${}^{\natural}t = 0$ for composition.

The power series expansions of many common analytical functions can be given by coalgebras. For example, the exponential is the unique function which has value 1 at 0 and is its own derivative:

$${}^{\natural}\exp = 1 \qquad \mathsf{D}\exp = \exp$$

and can clearly be defined by a coalgebra on a singleton set.

The sine and cosine functions are related by the derivation rules: $\sin' = \cos$ and $\cos' = -\sin$ and can therefore be defined by the following coalgebra on a four-element set:

$$\begin{array}{ll} \mathsf{sincos} : \mathbf{4} \to \mathbb{R} \times \mathbf{4} & [\![\mathsf{sincos}]\!] : \mathbf{4} \to \mathbb{S}_{\mathbb{R}} \\ \mathsf{sincos}\, 0 = \langle 0, 1\rangle & \\ \mathsf{sincos}\, 1 = \langle 1, 2\rangle & \sin = [\![\mathsf{sincos}]\!]\, 0 \\ \mathsf{sincos}\, 2 = \langle 0, 3\rangle & \cos = [\![\mathsf{sincos}]\!]\, 1. \\ \mathsf{sincos}\, 3 = \langle -1, 0\rangle & \end{array}$$

(It follows that $[\![\mathsf{sincos}]\!]\, 2 = -\sin$ and $[\![\mathsf{sincos}]\!]\, 3 = -\cos$.)

A different view of streams of numbers originates in the study of linear recurrence relations. In this case a useful operator is the *difference stream*:

$$\Delta \langle s_i\rangle_{i\in\mathbb{N}} = \langle s_{i+1} - s_i\rangle_{i\in\mathbb{N}}.$$

Theorem 4. *The coalgebra* $\langle \mathsf{head}, \Delta\rangle$ *is final for the functor* $FX = \mathbb{R} \times X$.

Proof. We define an isomorphism diftrans between the coalgebra $\langle \mathsf{head}, \Delta\rangle$ and the standard coalgebra $\langle \mathsf{head}, \mathsf{tail}\rangle$. It is obtained by repeatedly taking the head and then applying the difference operator, so

$$\mathsf{diftrans}\, s = \langle {}^{\natural}(\Delta^n s)\rangle_{n\in\mathbb{N}}.$$

The inverse transformation is computed similarly by iterating the sum of adjacent elements:

$$\Xi \langle t_i\rangle_{i\in\mathbb{N}} = \langle t_{i+1} + t_i\rangle_{i\in\mathbb{N}}, \qquad \mathsf{diftrans}^{-1} t = \langle {}^{\natural}(\Xi^m t)\rangle_{m\in\mathbb{N}}.$$

We leave it to the reader to verify that these are coalgebra morphisms and that they are inverses of each other. □

One application of the finality property of the difference coalgebra is in defining the binomial coefficients. Consider the following coalgebra:

$$\begin{array}{l} \gamma : \mathbb{N} \to \mathbb{R} \times \mathbb{N} \\ \gamma\; 0 = \langle 0, 0\rangle \\ \gamma\; 1 = \langle 1, 0\rangle \\ \gamma\; (n+2) = \langle 0, n+1\rangle \end{array}$$

and let β be the unique morphism from it to $\langle \mathsf{head}, \Delta\rangle$. Note that the unique morphism from γ to the $\langle \mathsf{head}, \mathsf{tail}\rangle$ final coalgebra maps $n+1$ to the stream that has 1 at position n and 0 everywhere else; and maps 0 to the constant 0 stream. Then we have that

$$\binom{n}{k} = {}^{\natural n}(\beta\; (k+1)).$$

You may check the correctness of this definition with respect to the standard one by verifying the following formulae:

$${}^{\natural k}(\Delta^n s) = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} s_{k+i}, \qquad {}^{\natural n}(\mathsf{diftrans}\, s) = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} s_i;$$

$${}^{\natural h}(\Xi^m t) = \sum_{j=0}^{m} \binom{m}{j} t_{h+j}, \qquad {}^{\natural m}(\mathsf{diftrans}^{-1}\, t) = \sum_{j=0}^{m} \binom{m}{j} t_j.$$

7. Coinductive tabulation

The following astonishing functional program was invented by Ulrich Berger [12]; I learnt about it from a blog post by Martín Escardó[1]:

$$
\begin{array}{l}
\mathsf{allb} : (\mathbb{S}_{\mathbb{B}} \to \mathbb{B}) \to \mathbb{B} \\
\mathsf{allb}\, f = f\ (\mathsf{counterexample}\, f) \\
\\
\mathsf{counterexample} : (\mathbb{S}_{\mathbb{B}} \to \mathbb{B}) \to \mathbb{S}_{\mathbb{B}} \\
\mathsf{counterexample}\, f = \ \ \mathbf{if}\ (\mathsf{allb}\, f_t) \\
\qquad\qquad\qquad\qquad\ \ \mathbf{then}\ (\mathsf{false} :: \mathsf{counterexample}\, f_f) \\
\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ (\mathsf{true} :: \mathsf{counterexample}\, f_t) \\
\quad \text{where } f_t = \lambda s.f\ (\mathsf{true} :: s) \\
\qquad\qquad\ f_f = \lambda s.f\ (\mathsf{false} :: s).
\end{array}
$$

It can be realised in a functional programming language like Haskell, but not in a type-theoretic language like Coq. The program determines whether its input (a Boolean predicate on streams of Booleans) is constantly true or not. It seems at first that the function would have to do a search on the uncountable set of Boolean streams, known as the *Cantor Space*. This would clearly be impossible, but we can prove that allb is a well-defined program using Brouwer's continuity principle: a computable function on Cantor space must be determined, on every stream argument, by a finite initial segment of the stream (see, for example, Section 3.3 of Dummett [25]). This principle allows us to build a potential counterexample: a stream on which f will return false if it does on any stream. Then the result of allb is just the output of f on the potential counterexample.

In fact, this proof exploits compactness of Cantor space, which is stronger than the continuity principle. Compactness states that every infinite covering of the space by open subsets contains a finite subcovering. Specifically, for $f : \mathbb{S}_{\mathbb{B}} \to \mathbb{B}$, consider the following covering ($[\mathbb{B}]$ is the type of lists of booleans, $+\!\!+$ is the concatenation operator):

$$
\{U_l\}_{l\in I} \quad \text{where } U_l = \{l +\!\!+ s \mid s : \mathbb{S}_{\mathbb{B}}\} \quad \text{for } l : [\mathbb{B}]
$$
$$
I = \{l : [\mathbb{B}] \mid \forall s_1, s_2 : U_l . f\ s_1 = f\ s_2\}.
$$

The continuity principle implies that $\{U_l\}_{l\in I}$ is a covering of Cantor space. Therefore, by compactness, a finite subcovering can be extracted and f needs to be checked only on a finite number of finite initial segments.

Other spaces that still satisfy the continuity principle but are not compact do not allow us to perform the same trick. For example, replacing $\mathbb{S}_{\mathbb{B}}$ with $\mathbb{S}_{\mathbb{N}}$ (known as the *Baire Space*), the universal quantifier over $\mathbb{S}_{\mathbb{N}} \to \mathbb{B}$ is not computable. However, a similar line of reasoning can be applied to define *tabulations* of functions on such spaces, as exemplified below.

The type $\mathbb{S}_A$ is isomorphic to the function space $\mathbb{N} \to A$. A function f can be *tabulated* by the stream of all its values: $f\,0 :: f\,1 :: f\,2 :: \cdots$. This observation can be generalised: Altenkirch showed that functions on a finitary inductive type can be tabulated by coinductive objects [5]. A further example tabulates functions on lists by a coinductive type of infinite trees:

$$
\begin{array}{ll}
\mathbf{codata} & \mathsf{lTab}\ (A, B : \mathsf{Set}) : \mathsf{Set} \\
 & \mathsf{lnode} : B \to (A \to \mathsf{lTab}_{A,B}) \to \mathsf{lTab}_{A,B}.
\end{array}
$$

The idea is that a tree defines a function on lists in the following way: given a list, see it as a path in the tree. This makes sense since the branches at every node are labelled by elements of a: follow the branch labelled with the head of the list, then the branch labelled with element with index 1, and so on. Once the list is exhausted, return as output the element of B labelling the node that you reached:

$$
\begin{array}{l}
\mathsf{lapply} : \mathsf{lTab}_{A,B} \to [A] \to B \\
\mathsf{lapply}\ (\mathsf{lnode}\, b\, f)\ [\,] = b \\
\mathsf{lapply}\ (\mathsf{lnode}\, b\, f)\ (a :: l) = \mathsf{lapply}\ (f\ a)\ l.
\end{array}
$$

Vice versa, the tabulation of a function is computed by constructing the tree in which every node is labelled with the value of the function on the list tracing the path leading to the node:

$$
\begin{array}{l}
\mathsf{tabulate} : ([A] \to B) \to \mathsf{lTab}_{A,B} \\
\mathsf{tabulate}\, f = \mathsf{lnode}\ (f\ [\,])\ (\lambda a.\mathsf{tabulate}\ (\lambda l.f\ (a :: l))).
\end{array}
$$

This definition correctly defines a function because the recursive call to tabulate is guarded by the constructor lnode. Categorically, it is justified by the universal property of $\mathsf{lTab}_{A,B}$ as a final coalgebra of the functor $\lambda X.B \times (A \to X)$.

Theorem 5. *Tabulation and application form an isomorphism between* $[A] \to B$ *and* $\mathsf{lTab}_{A,B}$: $\mathsf{tabulate} = \mathsf{lapply}^{-1}$.

[1] http://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/.

This isomorphism works for inductive types that are initial algebras of finitary functors (containers in which every shape as a finite number of positions). It has not been extended to data types with infinitary constructors.

There is a symmetric construction that tabulates functions on coinductive types by an inductive set of trees. A function $f : \mathbb{S}_A \to B$ can, under certain circumstances, be represented by a well-founded tree [29–31]:

data $\mathsf{sTab}_{A,B} : \mathsf{Set}$
$\mathsf{sleaf} : B \to \mathsf{sTab}_{A,B}$
$\mathsf{snode} : (A \to \mathsf{sTab}_{A,B}) \to \mathsf{sTab}_{A,B}$.

Application of a tabulation:

$$\begin{aligned}&\mathsf{sapply} : \mathsf{sTab}_{A,B} \to \mathbb{S}_A \to B\\&\mathsf{sapply}\,(\mathsf{sleaf}\,b)\,s = b\\&\mathsf{sapply}\,(\mathsf{snode}\,g)\,(a :: s) = \mathsf{sapply}\,(g\,a)\,s.\end{aligned}$$

Although the definitions of lapply and sapply look almost identical, there is a fundamental difference: lapply is defined by structural recursion on its list argument while sapply is defined by structural recursion on its tree argument.

Is there an inverse transformation/tabulation,

$$\mathsf{tabulate} : (\mathbb{S}_A \to B) \to \mathsf{sTab}_{A,B}?$$

Certainly not in general. But if B is a discrete/inductive type, an argument invoking a continuity principle can be made. Let $f : \mathbb{S}_A \to B$. By continuity, for every $s : \mathbb{S}_A$ there is an initial segment l such that $s = l +\!\!+ s'$ for some s' and $f\ (l +\!\!+ s'') = f\ s$ for all s''. In addition, assume that we can decide whether f is constant: if f is computed by a program, we just need to check whether it reads its argument. Then we can define:

$$\begin{aligned}&\mathsf{tabulate}\,f = \mathsf{sleaf}\,b && \text{if } f \text{ is constantly } b\\&\mathsf{tabulate}\,f = \mathsf{snode}\,(\lambda a.\mathsf{tabulate}\,(\lambda s.f\ (a :: s))) && \text{otherwise.}\end{aligned}$$

However, Brouwer's continuity principle is not a part of standard type theory and it would have to be added as a non-standard extension to justify such tabulations.

8. Infinite computations

A successful application of coinductive types is to model partial computable functions in type theory. Type theory is an essentially total language: we can only define functions that return an output for every input. It is not possible in general to define functions whose computations may diverge. There are two main reasons for this restriction. First, the Curry–Howard correspondence between types and propositions and between programs and proofs entails that, since allowing divergent proofs would lead to inconsistency, partial terms cannot exist. Second, we need type checking to be decidable: if we have dependent types, a judgement like $a : A[u]$, where A is a type depending on the term u, might be valid or not depending on the normalised value of u; if we cannot guarantee that u has a normal form, we cannot decide the typing judgement.

For this reason, all functions in (intensional) type theory are total. Various systems tried to extend it to an extensional version, abandoning one or other of the structural properties of the theory. A different avenue is to keep the native functions total and find constructions to represent, if not to compute, partial ones.

One approach inspired by set theory is to represent functions as relations with a unicity property. The graph of a partial function from A to B is represented by a relation $R : A \to B \to \mathsf{Prop}$ such that

$$\forall a, b_1, b_2.R\,a\,b_1 \to R\,a\,b_2 \to b_1 = b_2.$$

This representation may be appropriate for mathematical reasoning, but it forgets any computational content of the function. We prefer to explore different formalisations that still implement functions as elements of arrow types.

As a running example, let us choose the function that subsumes all partial programs: the minimisation operator from recursion theory. In our version it takes a stream of natural numbers $s : \mathbb{S}_{\mathbb{N}}$ as input and, if it terminates, returns as output $\mathsf{min}\,s$ the least natural n for which $\natural^n s = 0$. We use the typing notation $\mathsf{min} : \mathbb{S}_{\mathbb{N}} \rightharpoonup \mathbb{N}$ to denote this. The harpoon arrow $\rightharpoonup$ is not a constructor in type theory, however.

$$\begin{aligned}&\mathsf{min} : \mathbb{S}_{\mathbb{N}} \rightharpoonup \mathbb{N}\\&\mathsf{min}\,(0 :: s') = 0\\&\mathsf{min}\,(\mathsf{S}n :: s') = \mathsf{S}(\mathsf{min}\,s').\end{aligned}$$

One way to represent partial functions, pursued by Ana Bove and myself in a series of papers [14,15], is by a pair of an inductive domain predicate and a (total) function on the elements that satisfy it. First of all, characterise those streams on which min terminates, by an inductive predicate:

data $\mathsf{Dom}_{\mathsf{min}} : \mathbb{S}_{\mathbb{N}} \to \mathsf{Prop}$
$\mathsf{min}_0 : (s : \mathbb{S}_{\mathbb{N}})\mathsf{Dom}_{\mathsf{min}}\,(0 :: s)$
$\mathsf{min}_{\mathsf{S}} : (n : \mathbb{N}, s : \mathbb{S}_{\mathbb{N}})\mathsf{Dom}_{\mathsf{min}}\,s \to \mathsf{Dom}_{\mathsf{min}}\,(\mathsf{S}\,n :: s)$.

The domain is an inductive predicate with two constructors. The first, min_0, allows us to prove that streams with head 0 are in the domain. The second, min_S, recursively puts in the domain any stream (with nonzero head) whose tail is already in the domain. The minimisation function itself is realised as a mapping on both a stream and a proof that it satisfies the predicate:

$$\begin{aligned}&\mathsf{min} : (s : \mathbb{S})\mathsf{Dom}_{\mathsf{min}}\, s \to \mathbb{N}\\&\mathsf{min}\,(0 :: s)\,(\mathsf{min}_0\, s) = 0\\&\mathsf{min}\,(\mathsf{S}\, n :: s)\,(\mathsf{min}_\mathsf{S}\, n\, s\, h) = \mathsf{S}(\mathsf{min}\, s\, h).\end{aligned}$$

The function min is now recursively defined on the structure of the proof of $\mathsf{Dom}_{\mathsf{min}}$ and is therefore acceptable in type theory. Observe that for a stream s there can be at most one proof of $\mathsf{Dom}_{\mathsf{min}}\, s$, so the result of min still depends uniquely on s. This formalisation of computable functions is similar to that given by Kleene's normal form theorem. Kleene's T predicate characterises the completed traces of the function's computations, which correspond to the proofs of our domain predicate. (See any introduction to recursion theory, for example [61], Section 1.5.)

This method has proved very useful to represent partial programs and to reason about them. There are, however, some disadvantages. The function is not effectively computable: a proof of the domain predicate is basically a trace of the computation; since it has to be given in input, we must precompute the function ahead to have the proof. Another drawback is that each partial function has a different type because it has a differently defined domain predicate. So we do not get a single type of partial functions. This can be overcome if we work with an impredicative system or with type universes, where quantification over the (small) predicates is allowed. In that case we can define:

$$\begin{aligned}A \rightharpoonup B = \;& \Sigma P : A \to \mathsf{Prop}.\\& \Sigma f : (x : A)P\, x \to B.\\& \forall x : A; h_1, h_2 : P\, x. f\, x\, h_1 = f\, x\, h_2.\end{aligned}$$

A function from A to B is a triple of a domain predicate P on A; a function f on A and P; a proof that f does not depend on the proof of P.

An alternative way, which interests us here, uses a coinductive definition of potentially diverging elements. For every type A, we define a coinductive type whose elements can be thought of as possibly undefined elements of A.

Definition 5. Let A : Set be a data type. Then we define

$$\begin{aligned}\textbf{codata}\;& A^\nu : \mathsf{Set}\\& \mathsf{return} : A \to A^\nu\\& \mathsf{step} : A^\nu \to A^\nu.\end{aligned}$$

We use the notation $\lceil a \rceil$ for (return a) and $\lhd x$ for (step x).

Intuitively, an element of A^ν is either an element of A or a computation step followed by an element of A^ν. Since in coinductive types it is possible to define infinite elements, there is an object $\lhd\lhd\lhd\cdots$ denoting an infinite computation. Formally, it is defined as

$$\lhd^\infty = \lhd\lhd^\infty : A^\nu.$$

We want to identify all terms of the form $\lhd^k\lceil a \rceil$ as equivalent representations of the element a. We do it by defining an equality relation on A^ν that is a strengthening of bisimulation (see, for example, [51,2,11,35]). First we define, inductively, when an element of A^ν converges to a value in A and, coinductively, when it diverges.

Definition 6.

$$\begin{aligned}\textbf{data}\;& \mathsf{Value} : A^\nu \to A \to \mathsf{Prop}\\& \mathsf{value_return} : (a : A)(\mathsf{Value}\,\lceil a \rceil\, a)\\& \mathsf{value_step} : (x : A^\nu; a : A)(\mathsf{Value}\, x\, a) \to (\mathsf{Value}\,\lhd x\, a)\end{aligned}$$

$$\begin{aligned}\textbf{codata}\;& \mathsf{Diverge} : A^\nu \to \mathsf{Prop}\\& \mathsf{diverge} : (x : A^\nu)(\mathsf{Diverge}\, x) \to (\mathsf{Diverge}\,\lhd x).\end{aligned}$$

We use the notation $x \downarrow a$ for (Value x a) and $x^\uparrow$ for (Diverge x).

Since Value is inductively defined, it follows that the recursive constructor value_step can be applied just a finite number of times. So $x \downarrow a$ can be proved only if the x is in the form $\lhd^k\lceil a \rceil$. On the other hand, Diverge is coinductive and has no base case, so $x^\uparrow$ can be proved only if x consists of an infinite sequence of $\lhd$ steps.

Definition 7. Let A be a type. We define equality on A^ν by

$$\begin{aligned}\textbf{codata}\;& \mathsf{Eq}^\nu_A : A^\nu \to A^\nu \to \mathsf{Prop}\\& \mathsf{eq_value} : (x, y : A^\nu; a : A)x \downarrow a \to y \downarrow a \to (\mathsf{Eq}^\nu_A\, x\, y)\\& \mathsf{eq_step} : (x, y : A^\nu)(\mathsf{Eq}^\nu_A\, x\, y) \to (\mathsf{Eq}^\nu_A\,\lhd x\,\lhd y).\end{aligned}$$

We use the notation $x \overset{\nu}{=} y$ for $(\mathsf{Eq}^\nu_A\, x\, y)$.

Equality between partial elements captures the idea that converging elements are equal if they converge to the same value and that all diverging elements are equivalent:

$$x \stackrel{\nu}{=} y \leftrightarrow (\forall a, b. x \downarrow a \rightarrow y \downarrow b \rightarrow a = b);$$
$$x^{\uparrow} \rightarrow (x \stackrel{\nu}{=} y \leftrightarrow y^{\uparrow}).$$

With this definitions, we obtain a type of partial programs:

$$A \rightharpoonup B = A \rightarrow B^{\nu}$$

which allows the representation of all computable functions.

Theorem 6. *Let f be any computable (partial) mapping from natural numbers to natural numbers. There exists a type-theoretic function $\mathsf{f} : \mathbb{N} \rightarrow \mathbb{N}^{\nu}$ such that, for every $n, m : \mathbb{N}$, $(f\ n) = m$ if and only if $(\mathsf{f}\,n) \downarrow m$.*

You can find a proof of this theorem in [18]. Here let us just show how the minimisation function is defined. First, $-^{\nu}$ is a strong monad [52]. We refer to it as the *partiality monad.* Its morphism part lifts a function through computation steps:

$$-^{\nu} : (A \rightarrow B) \rightarrow (A^{\nu} \rightarrow B^{\nu})$$
$$f^{\nu}\ \lceil a \rceil = \lceil f\ a \rceil$$
$$f^{\nu} \lhd x = \lhd (f^{\nu}\ x).$$

The lifting is correctly defined by guarded corecursion, since the only recursive occurrence of f^{ν} is guarded by $\lhd$. Now, still by guarded corecursion, we have the minimisation operator:

$$\mathsf{min} : \mathbb{S}_{\mathbb{N}} \rightarrow \mathbb{N}^{\nu}$$
$$\mathsf{min}\ (0 :: s) = \lceil 0 \rceil$$
$$\mathsf{min}\ (\mathsf{S}\,n :: s) = \lhd\, \mathsf{S}^{\nu}\ (\mathsf{min}\, s).$$

We are being lax in our interpretation of guardedness: the occurrence of min appears as argument of S^{ν}, not of a constructor. But notice that S^{ν} "filters" all the constructors in its input up to its output, preserving productivity. A more rigorous definition is to be found in [18].

We have seen two different realisations of a type $A \rightharpoonup B$ of partial computable functions between types A and B. The first implements them as functions on a domain characterised by an inductive predicate; the second implements them as total functions returning a coinductively defined partial element. A bridge between the two techniques consists in defining a coinductive version of the domain predicate, whose elements are the traces of all computations, including the diverging ones:

codata $\mathsf{Trace}_{\mathsf{min}} : \mathbb{S}_{\mathbb{N}} \rightarrow \mathsf{Set}$
$\quad \mathsf{trace}_0 : (s : \mathbb{S}_{\mathbb{N}})\mathsf{Trace}_{\mathsf{min}}\ (0 :: s)$
$\quad \mathsf{trace}_{\mathsf{S}} : (n : \mathbb{N}, s : \mathbb{S}_{\mathbb{N}})\mathsf{Trace}_{\mathsf{min}}\ s \rightarrow \mathsf{Trace}_{\mathsf{min}}\ (\mathsf{S}\,n :: s).$

The paper [16] gives the theory behind coinductive computation traces and its relation with other methods to represent recursive functions.

9. Non-standard type theory and judgement rewriting

Per Martin–Löf proposed an alternative way of codifying infinite objects in type theory [49], not based on the theory of final coalgebras but inspired by non-standard analysis [62]. He begins by asking if the approach to the formalisation of infinite objects in Peter Aczel's non-well-founded set theory [2,24,11] can suitably extend to type systems. In that approach, the axiom of foundation is dropped in favour of the Anti-Foundation Axiom. Martin-Löf notices that this move would be problematic in a theory different from axiomatic set theory. For example, in arithmetic it would lead to the abandonment of the principle of induction.

In type theory, the introduction of infinite elements is inconsistent in the presence of structural induction. So a distinction has to be made between, on one side, inductive types, in which every element is well-founded and the induction principle holds; on the other side, coinductive types, which contain non-well-founded elements and do not satisfy structural induction.

The simplest infinite object to add to the theory would be the infinite natural number $\infty = \mathsf{SSS}\cdots$. An attempt to implement it in type theory is by introducing a new constant ∞ with an equality rule:

$$\begin{cases} \infty : \mathbb{N} \\ \infty = \mathsf{S}\infty : \mathbb{N}. \end{cases}$$

This is equivalent to adding a general fixed point operation:

$$\frac{a : A \quad x \in A \vdash f(x) : A}{\mathsf{fix}(a, f) : A} \qquad \frac{a : A \quad x \in A \vdash f(x) : A}{\mathsf{fix}(a, f) = f(\mathsf{fix}(a, f)) : A}.$$

But such an addition is inconsistent because we can prove the proposition $\mathsf{Id}_{\mathbb{N}}(\infty, \mathsf{S}\infty)$ (where Id is the *propositional equality*, distinct from the definitional equality $(=)$), but for natural numbers it is also true by induction that $\neg\mathsf{Id}_{\mathbb{N}}(x, \mathsf{S}x)$ for every x [69].

So a different way of modelling infinite objects is proposed, inspired by Brouwer's choice sequences and Robinson's non-standard analysis. An infinite object is not represented as a constant with a circular equality but as an unending sequence of constants, each defined in term of the next: $\infty = \infty_0 = \mathsf{S}\infty_1$, $\infty_1 = \mathsf{S}\infty_2$, $\infty_2 = \mathsf{S}\infty_3$, and so on. Formally:

$$\begin{cases}\infty_i : \mathbb{N} \\ \infty_i = \mathsf{S}\infty_{i+1} : \mathbb{N}\end{cases} \quad \text{for } i = 0, 1, 2, \ldots.$$

Again, the introduction of this sequence of constants is equivalent to introducing a sequence of fixed point operators:

$$\frac{a : A \quad x \in A \vdash f(x) : A}{\mathsf{fix}_i(a, f) : A} \qquad \frac{a : A \quad x \in A \vdash f(x) : A}{\mathsf{fix}_i(a, f) = f(\mathsf{fix}_{i+1}(a, f)) : A}.$$

We can use a single constant ∞ in place of an infinite sequence of them, and just recover the other ones by $\infty_i = \infty - \mathsf{S}^i 0$. We still need an infinite number of axioms:

$$\infty = \mathsf{S}^i(\infty - \mathsf{S}^i 0) : \mathbb{N}.$$

In conclusion, we get a theory with a set of axioms declaring the existence of infinite objects. But, as in non-standard analysis, in every concrete proof only a finite number of those axioms are used and the constants have always a finitary interpretation. This is the role of the element a, which is not involved in the recursive equality for the fixed points: it guarantees that every finite use of the rules can be instantiated. This validates the coexistence of potentially infinite objects and inductive principles on the same type.

These ideas have yet to be developed to their full potential. Not much research exists in the development of mathematics in non-standard type theory. One notable example is the work of Erik Palmgren on constructive non-standard analysis [56,57].

An application of this idea gives us a new approach to the representation and computation of general recursive functions. Remember that in Section 8 we represented a general recursive function by a pair consisting of a domain predicate Dom_f and a function f defined by recursion on a proof that the argument satisfied the predicate.

This meant that the function could be applied only to elements in the domain. The partiality monad allowed us to circumvent this restriction at the cost of not obtaining a value as output but an element of B^ν. A bridge between the two approaches was provided by the method of traces, but at the cost of a rather involved development.

Now consider that a trace is just the infinitary extension of a proof of the domain predicate. If, like in Martin–Löf's non-standard theory proposal, we could have both infinite proofs and the induction/recursion principle on them, we could formalise every partial recursive function directly as a element of an arrow type.

For example, consider the case of the minimisation operator. We could apply it only to streams $s : \mathbb{S}_{\mathbb{N}}$ for which we had a proof of $\mathsf{Dom}_{\mathsf{min}}\, s$. Why don't we add to our type theory a set of constants and axioms stating that this proof always exists?

$$\mathsf{dom}^{\infty}_{s} : \mathsf{Dom}_{\mathsf{min}}\, s \qquad \begin{array}{l}\mathsf{dom}^{\infty}_{(0::s)} = \mathsf{min}_0\, s \\ \mathsf{dom}^{\infty}_{(\mathsf{S}n::s)} = \mathsf{min}_{\mathsf{S}}\, n\, s\, (\mathsf{dom}^{\infty}_{s}).\end{array}$$

Unfortunately this simplistic application of the idea does not work: for some streams, for example the constant 1, it is provable that the domain predicate is false; so our non-standard constants would lead to inconsistency. Adding non-standard constants works only when they inhabit non-empty standard types, so the constants can always be instantiated by well-founded values.

But the idea can bear fruit in a less radical realisation: instead of adding global constants and axioms, we can add local variables and assumptions. That is, for every stream $s : \mathbb{S}_{\mathbb{N}}$ we can formulate the judgement

$$h : \mathsf{Dom}_{\mathsf{min}}\, s \vdash \mathsf{min}\, s\, h : \mathbb{N}.$$

This is a sound judgement, irrespectively of whether s has any 0 entries. Now imagine we unfold s. If we find that $s = 0 :: s'$, then we can immediately instantiate h with $\mathsf{min}_0\, s'$, delete the assumption from the judgement and obtain

$$\vdash \mathsf{min}\,(0 :: s')\,(\mathsf{min}_0\, s') \rightsquigarrow 0 : \mathbb{N}.$$

On the other hand, if we find that, for example, $s = 1 :: s'$, then we cannot fully instantiate h yet, but we can certainly infer that it must be of the form $\mathsf{min}_{\mathsf{S}}\, 0\, s'\, h'$ for some proof $h' : \mathsf{Dom}_{\mathsf{min}}\, s'$ yet to be determined. So let us replace h by h' in the judgement and refine its conclusion:

$$h' : \mathsf{Dom}_{\mathsf{min}}\, s' \vdash \mathsf{min}\,(1 :: s')\,(\mathsf{min}_{\mathsf{S}}\, 0\, s'\, h') \rightsquigarrow \mathsf{S}\,(\mathsf{min}\, s'\, h') : \mathbb{N}.$$

We can define a relation, which we call *judgement rewriting*, relating two judgements when the second is obtained from the first by a step of inversion and term reduction as above:

$$\begin{array}{l}\big(h : \mathsf{Dom}_{\mathsf{min}}\,(0 :: s') \vdash \mathsf{min}\,(0 :: s')\, h : \mathbb{N}\big) \Longrightarrow \big(\vdash 0 : \mathbb{N}\big); \\ \big(h : \mathsf{Dom}_{\mathsf{min}}\,(1 :: s') \vdash \mathsf{min}\,(1 :: s')\, h : \mathbb{N}\big) \Longrightarrow \big(h' : \mathsf{Dom}_{\mathsf{min}}\, s' \vdash \mathsf{S}\,(\mathsf{min}\, s'\, h') : \mathbb{N}\big).\end{array}$$

The standard view of type theory is based on the Curry–Howard correspondence between programs, proofs and terms and between computation and term reduction [40,70]. Here we propose that a better, more faithful, notion of computation can be realised by judgement rewriting.

For example, if $s = 7 :: 2 :: 3 :: 0 :: s$, we can compute the minimisation function on it like this:

$$\begin{aligned}
&\big(h_0 : \mathsf{Dom}_{\mathsf{min}}\, s \vdash \mathsf{min}\, s\, h_0 : \mathbb{N}\big)\\
&= \big(h_0 : \mathsf{Dom}_{\mathsf{min}}\, (7 :: 2 :: 3 :: 0 :: s) \vdash \mathsf{min}\, (7 :: 2 :: 3 :: 0 :: s)\, h_0 : \mathbb{N}\big)\\
&\Longrightarrow \big(h_1 : \mathsf{Dom}_{\mathsf{min}}\, (2 :: 3 :: 0 :: s) \vdash \mathsf{S}\, (\mathsf{min}\, (2 :: 3 :: 0 :: s)\, h_1) : \mathbb{N}\big)\\
&\Longrightarrow \big(h_2 : \mathsf{Dom}_{\mathsf{min}}\, (3 :: 0 :: s) \vdash \mathsf{SS}\, (\mathsf{min}\, (3 :: 0 :: s)\, h_2) : \mathbb{N}\big)\\
&\Longrightarrow \big(h_3 : \mathsf{Dom}_{\mathsf{min}}\, (0 :: s) \vdash \mathsf{SSS}\, (\mathsf{min}\, (0 :: s)\, h_3) : \mathbb{N}\big)\\
&\Longrightarrow \big(\vdash \mathsf{SSS0} : \mathbb{N}\big).
\end{aligned}$$

We obtained a judgement in normal form (where no inversion or reduction steps can be performed), so the result of our computation is 3.

We can apply this computation technique without knowing in advance that it will converge and without causing any problems of consistency or type checking: every judgement in the rewrite sequence is a valid, type-checked, verifiable sequent. For example, let us apply the procedure to $s = 7 :: 2 :: 3 :: s$:

$$\begin{aligned}
&\big(h_0 : \mathsf{Dom}_{\mathsf{min}}\, s \vdash \mathsf{min}\, s\, h_0 : \mathbb{N}\big)\\
&= \big(h_0 : \mathsf{Dom}_{\mathsf{min}}\, (7 :: 2 :: 3 :: s) \vdash \mathsf{min}\, (7 :: 2 :: 3 :: s)\, h_0 : \mathbb{N}\big)\\
&\Longrightarrow \big(h_1 : \mathsf{Dom}_{\mathsf{min}}\, (2 :: 3 :: s) \vdash \mathsf{S}\, (\mathsf{min}\, (2 :: 3 :: s)\, h_1) : \mathbb{N}\big)\\
&\Longrightarrow \big(h_2 : \mathsf{Dom}_{\mathsf{min}}\, (3 :: s) \vdash \mathsf{SS}\, (\mathsf{min}\, (3 :: s)\, h_2) : \mathbb{N}\big)\\
&\Longrightarrow \big(h_3 : \mathsf{Dom}_{\mathsf{min}}\, s \vdash \mathsf{SSS}\, (\mathsf{min}\, s\, h_3) : \mathbb{N}\big)\\
&= \big(h_3 : \mathsf{Dom}_{\mathsf{min}}\, (7 :: 2 :: 3 :: s) \vdash \mathsf{SSS}\, \mathsf{min}\, (7 :: 2 :: 3 :: s)\, h_3 : \mathbb{N}\big)\\
&\Longrightarrow \big(h_4 : \mathsf{Dom}_{\mathsf{min}}\, (2 :: 3 :: s) \vdash \mathsf{SSSS}\, (\mathsf{min}\, (2 :: 3 :: s)\, h_4) : \mathbb{N}\big)\\
&\Longrightarrow \cdots
\end{aligned}$$

The rewriting sequence is infinite, but every step of it is a sound judgement with a strongly normalising term as the conclusion.

Theorem 7. *For every stream $s : \mathbb{S}_{\mathbb{N}}$, s has its first zero entry at position n if and only if*

$$\big(h : \mathsf{Dom}_{\mathsf{min}}\, s \vdash \mathsf{min}\, s\, h : \mathbb{N}\big) \Longrightarrow^{*} (\vdash \mathsf{S}^{n}0 : \mathbb{N}).$$

10. Conclusion

This has been a short, breathless tour of some of the most exciting developments in the use of coalgebraic notions in functional programming and type theory. The selection of topics was admittedly biased towards the author's own interests. Hopefully the reader gained an appreciation of the fertile blossoming of applications in this area. Readers are encouraged to delve deeper into the cited literature for more technical and ample expositions of these topics and others that had to be omitted here.

References

[1] Michael Abott, Thorsten Altenkirch, Neil Ghani, Containers - constructing strictly positive types, Theoretical Computer Science 342 (2005) 3–27. Applied Semantics: Selected Topics.

[2] Peter Aczel, Non-Well-Founded Sets. Number 14 in CSLI Lecture Notes. Stanford University, 1988.

[3] Peter Aczel, Algebras and coalgebras, in: Roland Backhouse, Roy Crole, Jeremy Gibbons (Eds.), Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, in: LNCS, vol. 2297, Springer, 2002, pp. 79–88.

[4] Peter Aczel, Nax Paul Mendler, A final coalgebra theorem, in: D.H. Pitt, D.E. Rydehead, P. Dybjer, A.M. Pitts, A. Poigné (Eds.), Category Theory and Computer Science, in: Lecture Notes in Computer Science, vol. 389, Springer-Verlag, 1989, pp. 357–365.

[5] Thorsten Altenkirch, Representations of first order function types as terminal coalgebras, in: Samson Abramsky (Ed.), TLCA 2001, in: Lecture Notes in Computer Science, vol. 2044, Springer, 2001, pp. 8–21.

[6] Thorsten Altenkirch, Conor McBride, Wouter Swierstra, Observational equality, now!, in: Aaron Stump, Hongwei Xi (Eds.), PLPV '07: Proceedings of the 2007 Workshop on Programming Languages meets Program Verification, ACM, New York, NY, USA, 2007, pp. 57–68.

[7] Florent Balestrieri, The undecidability of pure stream equations. Draft paper, 2011.

[8] H.P. Barendregt, Lambda calculi with types, in: S. Abramsky, Dov M. Gabbay, T.S.E. Maibaum (Eds.), in: Handbook of Logic in Computer Science, vol. 2, Oxford University Press, 1992, pp. 117–309.

[9] Michael Barr, Algebraically compact functors, Journal of Pure and Applied Algebra 82 (1992) 211–231.

[10] Gilles Barthe, Venanzio Capretta, Olivier Pons, Setoids in type theory, Journal of Functional Programming 13 (2) (2003) 261–293.

[11] Jon Barwise, Lawrence Moss, Vicious Circles, Number 60 in CSLI Lecture Notes, CSLI, Stanford, California, 1996.

[12] Ulrich Berger, Totale Objekte und Mengen in der Bereichtheorie. Ph.D. Thesis, Munich LMU, 1990.

[13] Yves Bertot, Pierre Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Springer, 2004.

[14] Ana Bove, Venanzio Capretta, Nested general recursion and partiality in type theory, in: Richard J. Boulton, Paul B. Jackson (Eds.), Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001, in: Lecture Notes in Computer Science, vol. 2152, Springer-Verlag, 2001, pp. 121–135.

[15] Ana Bove, Venanzio Capretta, Modelling general recursion in type theory, Mathematical Structures in Computer Science 15 (4) (2005) 671–708.

[16] Ana Bove, Venanzio Capretta, Computation by prophecy, in: Simona Ronchi Della Rocca (Ed.), Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26–28, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4583, Springer, 2007, pp. 70–83.

[17] Ana Bove, Peter Dybjer, Ulf Norell, A brief overview of Agda - a functional language with dependent types, in: Stefan Berghofer, Tobias Nipkow, Christian Urban, Makarius Wenzel (Eds.), TPHOLs 2009, in: Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 73–78.

[18] Venanzio Capretta, General recursion via coinductive types, Logical Methods in Computer Science 1 (2) (2005) 1–18.

[19] Venanzio Capretta, Bisimulations generated from corecursive equations, in: Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2010, Electronic Notes in Theoretical Computer Science 265 (2010) 245–258.

[20] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, S.F. Smith, Implementing Mathematics with the NuPrl Proof Development System, Prentice-Hall, 1986.

[21] Thierry Coquand, Pattern matching with dependent types, in: Bengt Nordström, Kent Petersson, and Gordon Plotkin, (Eds.), Proceedings of the 1992 Workshop on Types for Proofs and Programs, pp. 71–83, 1992.

[22] Thierry Coquand, Infinite objects in type theory, in: Henk Barendregt, Tobias Nipkow (Eds.), Types for Proofs and Programs. International Workshop TYPES'93, in: Lecture Notes in Computer Science, vol. 806, Springer-Verlag, 1993, pp. 62–78.

[23] Nils Anders Danielsson, Thorsten Altenkirch, Mixing induction and coinduction. Draft paper, 2009.

[24] Keith Devlin, The Joy of Sets, second ed., Springer, 1993.

[25] Michael Dummett, Elements of Intuitionism, second ed., Oxford Science Publications, 2000.

[26] Peter Dybjer, Representing inductively defined sets by Wellorderings in Martin-Löf type theory, Theoretical Computer Science 176 (1997) 329–335.

[27] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, Jan Willem Klop, Productivity of stream definitions, in: Erzsébet Csuhaj-Varjú, Zoltán Ésik (Eds.), FCT 2007, in: Lecture Notes in Computer Science, vol. 4639, Springer, 2007, pp. 274–287.

[28] Peter Freyd, Algebraically complete categories, in: A. Carboni, M.C. Pedicchio, G. Rosolini (Eds.), Category Theory. Proceedings of the International Conference held in Como, Italy, July 22–28, 1990, in: Lecture Notes in Mathematics, vol. 1488, Springer, 1991, pp. 95–104.

[29] Neil Ghani, Peter Hancock, Dirk Pattinson, Continuous functions on final coalgebras, in: Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science, CMCS 2006, Electronic Notes in Theoretical Computer Science 164 (1) (2006) 141–155.

[30] Neil Ghani, Peter Hancock, Dirk Pattinson, Continuous functions on final coalgebras, in: Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics, MFPS 2009, Electronic Notes in Theoretical Computer Science 249 (2009) 3–18.

[31] Neil Ghani, Peter Hancock, Dirk Pattinson, Representations of stream processors using nested fixed points, Logical Methods in Computer Science 5 (3) (2009).

[32] Jeremy Gibbons, Graham Hutton, Proof methods for corecursive programs, Fundamenta Informaticae 66 (4) (2005) 353–366.

[33] Eduardo Giménez, Codifying guarded definitions with recursive schemes, in: Peter Dybjer, Bengt Nordström, Jan Smith (Eds.), Types for Proofs and Programs. International Workshop TYPES'94, in: Lecture Notes in Computer Science, vol. 996, Springer, 1994, pp. 39–59.

[34] Eduardo Giménez, A Tutorial on Recursive Types in Coq. Technical Report 0221, Unité de recherche INRIA Rocquencourt, May 1998.

[35] Claudio Hermida, Bart Jacobs, Structural induction and coinduction in a fibrational setting, Information and Computation 145 (1998) 107–152.

[36] Ralf Hinze, Functional pearl: streams and unique fixed points, in: James Hook, Peter Thiemann (Eds.), ICFP 2008, ACM, 2008, pp. 189–200.

[37] Martin Hofmann, Elimination of extensionality in Martin-Löf type theory, in: Henk Barendregt, Tobias Nipkow (Eds.), Types for Proofs and Programs. International Workshop TYPES'93, in: Lecture Notes in Computer Science, vol. 806, Springer, 1993, pp. 166–190.

[38] Martin Hofmann, Extensional concepts in intensional type theory. Ph.D. Thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1995.

[39] Martin Hofmann, A simple model for quotient types, in: M. Dezani-Ciancaglini, G. Plotkin (Eds.), Proceedings of TLCA'95, in: Lecture Notes in Computer Science, vol. 902, Springer-Verlag, 1995, pp. 216–234.

[40] W.A. Howard, The formulae-as-types notion of construction, in: J.P. Selding, J.R. Hindley (Eds.), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, pp. 479–490.

[41] Graham Hutton, Programming in Haskell, Cambridge University Press, 2007.

[42] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler, Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language, 1999. http://www.haskell.org/onlinereport/.

[43] Clemens Kupke, Jan J.M.M. Rutten, Observational coalgebras and complete sets of co-operations, Electronic Notes in Theoretical Computer Science 203 (5) (2008) 153–174.

[44] Alexander Kurz, Marina Lenisa, Andrzej Tarlecki (Eds.), Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7–10, 2009. Proceedings, in: Lecture Notes in Computer Science, vol. 5728, Springer, 2009.

[45] Joachim Lambek, A fixpoint theorem for complete categories, Math. Zeitschr. 103 (1968) 151–161.

[46] Per Martin-Löf, An intuitionistic theory of types: predicative part, in: H.E. Rose, J.C. Shepherdson (Eds.), Logic colloquium'73, North-Holland, 1975, pp. 153–175.

[47] Per Martin-Löf, Constructive mathematics and computer programming, in: Logic, Methodology and Philosophy of Science, VI, 1979, North-Holland, 1982, pp. 153–175.

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

[49] Per Martin-Löf, Mathematics of infinity, in: Per Martin-Löf, Grigori Mints (Eds.), Conference on Computer Logic, in: Lecture Notes in Computer Science, vol. 417, Springer, 1988, pp. 146–197.

[50] Conor McBride, James McKinna, The view from the left, Journal of Functional Programming 14 (1) (2004) 69–111.

[51] Robin Milner, A Calculus of Communicating Systems, in: Lecture Notes in Computer Science, vol. 92, Springer-Verlag, 1980.

[52] Eugenio Moggi, Notions of computation and monads, Information and Computation 93 (1991) 55–92.

[53] Bengt Nordström, Kent Petersson, Jan M. Smith, Programming in Martin-Löf's Type Theory. An Introduction, in: International Series of Monographs on Computer Scence, vol. 7, Oxford University Press, 1990.

[54] Ulf Norell, Dependently typed programming in Agda, in: Andrew Kennedy, Amal Ahmed (Eds.), TLDI 2009, ACM, 2009, pp. 1–2.

[55] Bryan O'Sullivan, John Goerzen, Don Stewart, Real World Haskell, O'Reilly, 2009.

[56] Erik Palmgren, A note on mathematics of infinity, Journal of Symbolic Logic 58 (4) (1993) 1195–1200.

[57] Erik Palmgren, A constructive approach to nonstandard analysis, Annals of Pure and Applied 73 (3) (1995) 297–325.

[58] Erik Palmgren, On universes in type theory, in: Giovanni Sambin, Jan Smith (Eds.), Twenty-Five Years of Constructive Type Theory, in: Oxford Logic Guides, vol. 36, Oxford Science Publications, 1998, pp. 191–204. Proceedings of a Congress Held in Venice, October 1995.

[59] David Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), Theoretical Computer Science, in: LNCS, vol. 104, Springer-Verlag, 1981, pp. 167–183. Proceedings of the 5th GI-Conference Karlsruhe.

[60] Dusko Pavlović, Martín Escardó, Calculus in coinductive form, in: Vaughan Pratt (Ed.), Proceedings of the Thirteenth Annual IEEE Symp. on Logic in Computer Science, LICS 1998, IEEE Computer Society Press, 1998, pp. 408–417.

[61] J.C.C. Phillips, Recursion theory, in: Handbook of Logic in Computer Science. Volume 1. Background: Mathematical Structures, Oxford University Press, 1992, pp. 79–187.

[62] Abraham Robinson, Non-standard Analysis, North Holland, 1966.
[63] Grigore Rosu, Equality of streams is a Π_2^0-complete problem, in: John H. Reppy, Julia L. Lawall (Eds.), ICFP 2006, ACM, 2006, pp. 184–191.
[64] Grigore Rosu, Dorel Lucanu, Circular coinduction: A proof theoretical foundation, in: Kurz et al. [44], pp. 127–144.
[65] J. Rutten, D. Turi, On the foundation of final semantics: non-standard sets, metric spaces and partial orders, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), Semantics: Foundations and Applications, in: LNCS, vol. 666, Springer-Verlag, Berlin, 1993, pp. 477–530.
[66] J.J.M.M. Rutten, A coinductive calculus of streams, Mathematical Structures in Computer Science 15 (2005) 93–147.
[67] Davide Sangiorgi, On the origins of bisimulation and coinduction, ACM Transactions on Programming Languages and Systems 31 (4) (2009).
[68] Alexandra Silva, Jan J.M.M. Rutten, Behavioural differential equations and coinduction for binary trees, in: Daniel Leivant, Ruy J.G.B. de Queiroz (Eds.), WoLLIC, in: Lecture Notes in Computer Science, vol. 4576, Springer, 2007, pp. 322–336.
[69] Jan M. Smith, The independence of Peano's fourth axiom from Martin-Lof's type theory without universes, Journal of Symbolic Logic 53 (3) (1988) 840–845.
[70] Morten Heine B. Sørensen, P. Urzyczyn, Lectures on the Curry-Howard Isomorphism, Elsevier Science, 2006.
[71] Sam Staton, Relating coalgebraic notions of bisimulation, in: Kurz et al. [44], pp. 191–205.
[72] The Coq Development Team. LogiCal Project, The Coq Proof Assistant. Reference Manual. Version 8.1, INRIA, 2003. Available at the web page http://pauillac.inria.fr/coq/coq-eng.html.
[73] Daniele Turi, Jan Rutten, On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces, Mathematical Structures in Computer Science 8 (5) (1998) 481–540.
[74] Tarmo Uustalu, Varmo Vene, Mendler-style inductive types, categorically, Nordic Journal of Computing 6 (3) (1999) 343–361.
[75] Tarmo Uustalu, Varmo Vene, Comonadic notions of computation, Electronic Notes in Theoretical Computer Science 203 (5) (2008) 263–284.
[76] Hans Zantema, Well-definedness of streams by termination, in: Ralf Treinen (Ed.), RTA, in: Lecture Notes in Computer Science, vol. 5595, Springer, 2009, pp. 164–178.