Chapter 1
Coalgebras as Types determined by their Elimination Rules

Anton Setzer

Abstract We develop rules for coalgebras in type theory, and give meaning explanations for them. We show that elements of coalgebras are determined by their elimination rules, whereas the introduction rules can be considered as derived. This is in contrast with algebraic data types, for which the opposite is true: elements are determined by their introduction rules, and the elimination rules can be considered as derived. In this sense, the function type from the logical framework is more like a coalgebraic data type, the elements of which are determined by the elimination rule. We illustrate why the simplest form of guarded recursion is nothing but the introduction rule originating from the formulation of coalgebras in category theory. We discuss restrictions needed in order to preserve decidability of equality.

Dedicated to Per Martin-Löf on the occasion of his retirement.

1.1 Introduction

Most programs in computing are interactive programs. This means that they are not batch programs, which, once started, are guaranteed to terminate after a certain amount of time and deliver their result. They are programs which keep running and interacting with user input, until they are terminated by the user. Such programs correspond to non-well-founded trees: Nodes are labelled by commands and the branching degree of a node labelled by a command is the set of responses to this command. A computation which goes on for ever corresponds to an infinite path in this tree. More details of this can be found in a series of articles by the author and Peter Hancock [22, 23, 24, 25, 26]. Colists are simple trees with branching degrees 0 or 1, and for ease of presentation, we restrict ourselves in this article to colists.

Supported by EPSRC grant EP/G033374/1
Dept. of Computer Science, Swansea University, Singleton Park, Swansea SA2 8PP, UK, e-mail: a.g.setzer@swansea.ac.uk

Martin-Löf type theory is supposed to be a language in which programs can be written and in which we can prove correctness properties of such programs. In order to be able to write interactive programs and reason about them, we need to represent non-well-founded structures. Coalgebras originating from category theory provide a theory of non-well-founded structures. They allow to represent the elements of such structures in a finitary way. Elements are not per se infinitary – in fact we will represent them in type theory as finitary objects. As part of a coalgebra we have a case distinction operation. In case of colists, the result of applying it to a colist is the information whether the element represents the empty list or a list formed from a given head and a given tail. By iteratively applying case distinction, a colist then unfolds to a potentially infinite list.

The goal of this article is to introduce a notion of coalgebras into type theory and provide meaning explanations for them. We want coalgebras to be first class citizens, i.e. they are not encoded in terms of other data types. This seems to be the general way of moving forward in type theory. In most other mathematical theories the goal is to define a minimal closed theory, which allows to encode all structures needed in mathematics. In type theory it is usual practice to continuously extend the theory in such a way that new structures needed are represented directly.

In this article we develop the theory of coalgebras in type theory, while closely following the categorical notions. One main focus is to develop meaning explanations for coalgebras, in order to fully integrate them into the theoretical setting of type theory. Whereas coalgebras only extend the expressiveness, not the proof theoretic strength, of type theory, we hope that this project will help to develop the basis for future proof theoretic strong extensions of type theory.

We start by exploring the notion of inductive data types, which correspond to initial algebras. We will as well review meaning explanations for them. Then we develop the notion of a final coalgebra. We will see that a simple form of guarded recursion is nothing but the introduction rule of final coalgebras, which represent the existence of morphisms in the defining diagrams for coalgebras. We will develop a slight extension of guarded recursion as well. We then explore limitations of coalgebras needed in order to maintain decidable equality. For this reason we will switch to weakly final coalgebras with an extended version of guarded recursion. We will see that in a decidable type theory we cannot assume that every element is introduced by a coconstructor. This is the underlying reason for the failure of subject reduction in implementations of type theory and problems with dependent case distinction. Next, we develop type theoretic rules for coalgebras based on extended guarded recursion.

In the last part, we will develop meaning explanations for coalgebras. We will need to change the setting of meaning explanations in order to be able to explain coalgebras. As in the original meaning explanations by Martin-Löf, inductive data types are given given by explaining how to introduce its elements and when two elements introduced are equal. So the elements are determined by their introduction rules. The elimination rules are justified by verifying that they operate correctly for every element introduced. Meaning explanations of coalgebras are given differently. Elements of coalgebras are given by defining how to compute other elements from them. Elements are equal if the computed results are equal. Therefore elements are given by their elimination rules. The introduction rules are justified by verifying that they introduce elements which allow to apply the elimination principle.

Related Work. The use of coalgebras in non-dependent functional programming was to the author's knowledge first introduced 1987 in the PhD thesis of Hagino [20] (see as well [21]). He used the terminology codatatype for coalgebras defined by their elimination rules. Aczel introduced 1988 in his book [1] non-well-founded set theory. Non-well-founded sets are necessarily infinite objects, which can be introduced by the anti-foundation axiom, a form of guarded recursion. Based on Hagino's work, Cockett, Fukushima and Spencer developed 1992 the non-dependent functional programming language Charity with a very clean categorical syntax. Leclerc and Paulin-Mohring in [32] 1994 used the impredicative types in Coq in order to represent streams and define the sieve of Eratosthenes. Coquand 1994 introduced in [10] the concept of guarded recursion. Giménez [19, 18] developed 1994 an extension of the calculus of constructions by inductive and coinductive types. He showed how to reduce general forms of guarded recursion to coalgebras. Already in his PhD thesis [18], he discovered problems with subject reduction, which will discussed later in this paper. Paulson implemented 1994 axioms for coinduction in Isabelle [43]. Telford and Turner [47, 49, 48] starting 1995 promoted the use of codata as truly infinite data types introduced by their introduction rules, and implemented them in the functional programming language Miranda. The author has together with Hancock since 1999 developed in [22, 23, 24, 25, 26] interactive programs in dependent type theory. This included in [25, 26] a definition of the rules for guarded recursion and weakly final coalgebras in Martin-Löf Type Theory (2004). Coalgebras have been introduced in the interactive theorem prover Coq. The "Coq-book" [6] by Bertot and Castéran contains an extensive chapter 14 on the development of coinductive data types and proofs of their properties. See as well the note [5] by Bertot. Coinductive data types have as well been implemented in Agda [41] by Norell, Danielsson, Abel and other members of the Agda development team – see intense discussions on the Agda email list [2]. The latest version, which is currently implemented in Agda using a notion for coalgebraic arguments, was presented in [4]. McBride has written a short paper [38] on the problem of subject reduction in coalgebras, and how to develop coalgebras in observational type theory. We will discuss this paper later in detail.

General setting and notations. This paper is heavily based on Martin-Löf Type Theory [34], mainly on the version presented in the second part of [40], with the restriction to the small logical framework outlined below. As usual we have the basic judgements $A : \mathrm{Set}$, $A = B : \mathrm{Set}$, $a : A$ and $a = b : A$. Hypothetical judgements will be written as $\Gamma \Rightarrow \theta$, where θ is a basic judgement and Γ a context. Contexts Γ have the form $x : A_1, \ldots, x_n : A_n$, where $x_1 : A_1, \ldots, x_{i-1} : A_{i-1} \Rightarrow A_i : \mathrm{Set}$. If $\emptyset$ is the empty context, we write instead of $\emptyset \Rightarrow \theta$ simply θ.

We will develop type theory based on the small logical framework, see for instance [44]. If $A : \mathrm{Set}$ and $x : A \Rightarrow B : \mathrm{Set}$, we can form the dependent function set $(x : A) \to B : \mathrm{Set}$. (This type is often written as $\Pi x : A.B$. However, in Martin-Löf

Type Theory $\Pi x : A.B$ is reserved for the inductive data type having constructor $\lambda : ((x : A) \to B) \to \Pi x : A.B)$.

The canonical elements of $(x : A) \to B$ are terms $(x)t$ where $x : A \Rightarrow t : B$, which is sometimes written as $\lambda x.t$. Following the conventions in Martin-Löf Type Theory, we reserve λ for the constructor of $\Pi x : A.B$. Application is written in functional style in the form $(s\,t)$. We use usual abbreviations such as writing $(r\,s\,t)$ for $((r\,s)\,t)$ (the outermost brackets are only for better readability). Furthermore $(x : A, y : B, z : C) \to D$ denotes $(x : A) \to ((y : B) \to ((z : C) \to D))$.

Note that large types such as $(x : A) \to \mathrm{Set}$ are only allowed in the full logical framework. The reason for restricting ourselves to the small logical framework is that we have a satisfactory understanding of how to develop meaning explanations for it. One central part of this article is the discussion of meaning explanation for coinductive types.

Because of the restriction to the small logical framework, arguments referring to elements of type Set are presented as premises in rules. For practical applications, the use of the full logical framework, as it is implemented for instance in Agda, is preferred. Then these arguments can easily be abstracted.

Apart from the standard structural rules and the rules for the dependent function sets, we add rules for the intensional equality type $a ==_A b$ (where $A : \mathrm{Set}$, $a : A$ and $b : A$), the one element set $\mathbf{1}$ with only element $* : \mathbf{1}$, the binary product $(A \times B)$ (where $A, B : \mathrm{Set}$), the disjoint union $(A + B)$ (again $A, B : \mathrm{Set}$), and the set of natural numbers $\mathbb{N}$. The use of $\mathbb{N}$ is not crucial for the development of type theory in this article, we just use it as a convenient example set.

We will use expressions such as $C(x)$, $\mathrm{step}_{\mathrm{cons}}(n,l)$ for terms depending on free variables x or n, l. After using $C(x)$, the expression $C(t)$ is the result of substituting the term t for x (where we identify α-equivalent terms and resolve substitution problems as usual). After a premise of a rule $x : A \Rightarrow C(x) : \mathrm{Set}$ we write simply C rather than $(x)C(x)$ for the argument C. The same applies to similar expressions as well.

Acknowledgements. We want to thank the anonymous referee for valuable comments on earlier version of this articles. We want to thank as well our PhD student Fredrik Nordvall Forsberg for diligent proof reading and valuable remarks.

1.2 Initial Algebras defined by their Introduction Rules

The set of lists in Martin-Löf Type Theory. In Martin-Löf type theory, types are usually introduced by their introduction rules. Let us consider the type of lists of natural numbers. It has formation rule

$$\mathrm{List}_{\mathbb{N}} : \mathrm{Set}$$

and introduction rules

$$\mathrm{nil} : \mathrm{List}_{\mathbb{N}} \qquad \mathrm{cons} : \mathbb{N} \to \mathrm{List}_{\mathbb{N}} \to \mathrm{List}_{\mathbb{N}}$$

The elimination rules express that $\mathrm{List}_{\mathbb{N}}$ is the least set closed under these operations, as expressed by the principle of higher type primitive recursion over lists:

$$\frac{x : \mathrm{List}_{\mathbb{N}} \Rightarrow C(x) : \mathrm{Set}}{\begin{array}{ll}\mathrm{Rec}^{\mathrm{List}}_{C} : & (\mathrm{step}_{\mathrm{nil}} : C(\mathrm{nil})) \\ & \to (\mathrm{step}_{\mathrm{cons}} : (n : \mathbb{N}, l : \mathrm{List}_{\mathbb{N}}) \to C(l) \to C(\mathrm{cons}\, n\, l)) \\ & \to (l : \mathrm{List}_{\mathbb{N}}) \\ & \to C(l)\end{array}}$$

The equality rules, where we omit the obvious assumptions on types of the parameters, are as follows:

$$\begin{array}{lcl}\mathrm{Rec}^{\mathrm{List}}_{C}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ \mathrm{nil} & = & \mathrm{step}_{\mathrm{nil}} \\ \mathrm{Rec}^{\mathrm{List}}_{C}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ (\mathrm{cons}\, n\, l) & = & \mathrm{step}_{\mathrm{cons}}\ n\ l\ (\mathrm{Rec}^{\mathrm{List}}_{C}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ l)\end{array}$$

By the type theoretic rules for $\mathrm{List}_{\mathbb{N}}$ we mean the rules above.

Meaning explanations were introduced by Per Martin-Löf [34, 35, 36, 37]. They are part of a program to develop a theory in such a way that we have a direct insight that everything proved in it is correct. By Gödel's incompleteness theorem we know that there is no proof of the consistency of any reasonable mathematical theory by weaker methods. Therefore, there is no mathematical argument which guarantees that the mathematical theories used for proving theorems are actually consistent, and which wouldn't be prone to the danger of using an inconsistency of the theory in question. So any justification for the consistency of a reasonable mathematical theory needs ultimately be based on a philosophical argument. Such an argument can never be fully formal – otherwise we would obtain a mathematical proof of the consistency of the theory in question. What meaning explanations by Martin-Löf provide is the to the author's knowledge at this time best possible way of getting a direct insight into the validity of the judgements derivable in Martin-Löf type theory. They are a way of making as precise as possible the reasons why all judgements derivable in this theory are valid.

In meaning explanations one gives a meaning to each judgement and investigates for each rule that we obtain valid judgements in the conclusion from valid judgements in the premise. The meaning of a set is given by explaining what the elements are and when two elements are equal. Two sets are equal if an element of one set is an element of the other, and if two elements are equal in one set they are so in the other.

One should note that meaning explanations, as the author understand them, justify extensional equality. For colists, as defined later, they will even justify bisimilarity as equality (which will be introduced below). We do not see any inherent problem in it. The reason for having intensional equality is that we want to decide for every proposition whether a term p is a proof of this proposition. Hence we need decidable type checking.

Meaning explanations for $\mathrm{List}_{\mathbb{N}}$. In these explanations, elements are determined by their introduction rules. $\mathrm{List}_{\mathbb{N}}$ is a set. We have nil is a canonical element of $\mathrm{List}_{\mathbb{N}}$, and for n a natural number, and l an element of $\mathrm{List}_{\mathbb{N}}$ we have that $(\mathrm{cons}\ n\ l)$ is a canonical element of $\mathrm{List}_{\mathbb{N}}$. Non-canonical elements of $\mathrm{List}_{\mathbb{N}}$ are programs which evaluate to canonical elements of $\mathrm{List}_{\mathbb{N}}$. The element nil is equal to itself. The elements $(\mathrm{cons}\ n\ l)$ and $(\mathrm{cons}\ n', l')$ are equal, if n and n' are equal elements of $\mathbb{N}$ and l and l' are equal elements of $\mathrm{List}_{\mathbb{N}}$. The elements nil and $(\mathrm{cons}\ n\ l)$ are not equal. Non-canonical elements are equal, if the results of evaluating them to canonical elements are equal.

The elimination and equality rules are explained by showing how to compute from elements of $\mathrm{List}_{\mathbb{N}}$ elements of other sets. Their explanation uses that we have determined what the canonical elements of $\mathrm{List}_{\mathbb{N}}$ are, so it makes use of the introduction rules for $\mathrm{List}_{\mathbb{N}}$. The explanation of $\mathrm{Rec}_C^{\mathrm{List}}$ is as follows: Assume $C(x)$ is a set, depending on an element x of $\mathrm{List}_{\mathbb{N}}$. So for every element l of $\mathrm{List}_{\mathbb{N}}$ we have that $C(l)$ is a set. Assume $\mathrm{step}_{\mathrm{nil}}$ is an element of $C(\mathrm{nil})$ and $\mathrm{step}_{\mathrm{cons}}$ is a function, which maps elements n of $\mathbb{N}$, l of $\mathrm{List}_{\mathbb{N}}$ and p of $C(l)$ to elements of $C(\mathrm{cons}\ n\ l)$. Assume l is an element of $\mathrm{List}_{\mathbb{N}}$. Then $(\mathrm{Rec}_C^{\mathrm{List}}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ l)$ is a program which computes an element of $C(l)$. This element is computed as follows: First l is computed which evaluates to a canonical element of $\mathrm{List}_{\mathbb{N}}$. If this element is nil, then $(\mathrm{Rec}_C^{\mathrm{List}}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ l)$ evaluates to the result of computing $\mathrm{step}_{\mathrm{nil}}$ which is an element of $C(\mathrm{nil})$ and therefore as well of $C(l)$. Otherwise l evaluates to $(\mathrm{cons}\ n\ l')$, where n is an element of $\mathbb{N}$ and l' is an element of $\mathrm{List}_{\mathbb{N}}$. Before we introduce l we have introduced l' and therefore $c' := \mathrm{Rec}_C^{\mathrm{List}}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ l'$ is an element of $C(l')$. Now $(\mathrm{Rec}_C^{\mathrm{List}}\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ l)$ is evaluated by computing $(\mathrm{step}_{\mathrm{cons}}\ n\ l'\ c')$ which has as result an element of $C(\mathrm{cons}\ n\ l')$ and therefore of $C(l)$. The equality rules follow since the left hand side is evaluated by evaluating the right hand side.

$\mathrm{List}_{\mathbb{N}}$ as an initial algebra. Assume a category having finite products (including an initial object $\mathbf{1}$ which is the empty product), and a binary coproduct $(A+B)$ for objects A,B. Assume as well a natural numbers object $\mathbb{N}$ (we will not need any specific properties about it). Elements a of objects A are arrows $a : \mathbf{1} \to A$, and we write $a : A$ for such elements. Let $\mathrm{F}_{\mathrm{List}}$ be the functor with object part $\mathrm{F}_{\mathrm{List}}\ X = \overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, X)$. Here $\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, X)$ is a notation for $\mathbf{1} + (\mathbb{N} \times X)$, where we write $\overline{\mathrm{nil}} := \mathrm{inl} *$ for the element $* : \mathbf{1}$ (corresponding to $\mathrm{id} : \mathbf{1} \to \mathbf{1}$) and $(\overline{\mathrm{cons}}\ n\ x)$ for the element $(\mathrm{inr}\ \langle n, x \rangle)$ where $n : \mathbb{N}$ and $x : X$. The name $\overline{\mathrm{nil}}$ signifies a nil-shape and $\overline{\mathrm{cons}}$ a cons-shape. For $f : A \to B$ we obtain an obvious morphism part $\mathrm{F}_{\mathrm{List}}\ f : \mathrm{F}_{\mathrm{List}}\ A \to \mathrm{F}_{\mathrm{List}}\ B$. An $\mathrm{F}_{\mathrm{List}}$-algebra is a pair (A, f) where A is an object and $f : \mathrm{F}_{\mathrm{List}}\ A \to A$. A morphism between $\mathrm{F}_{\mathrm{List}}$-algebras (A, f) and (B, g) is a function $h : A \to B$ s.t. the following diagram commutes:

$$\begin{tikzcd}
\mathrm{F}_{\mathrm{List}}\ A \arrow[r, "f"] \arrow[d, "\mathrm{F}_{\mathrm{List}}\ h"'] & A \arrow[d, "h"] \\
\mathrm{F}_{\mathrm{List}}\ B \arrow[r, "g"] & B
\end{tikzcd}$$

An initial $\mathrm{F}_{\mathrm{List}}$-algebra $(\mathrm{List}_{\mathbb{N}}, \mathrm{intro})$ is an initial object in the category with objects $\mathrm{F}_{\mathrm{List}}$-algebras and morphism being $\mathrm{F}_{\mathrm{List}}$-morphisms. So we have a morphism $\mathrm{intro} : \mathrm{F}_{\mathrm{List}}(\mathrm{List}_{\mathbb{N}}) \to \mathrm{List}_{\mathbb{N}}$, and if we have any other $\mathrm{F}_{\mathrm{List}}$-algebra (A, f), i.e. if we have $f : \mathrm{F}_{\mathrm{List}}\, A \to A$, then there exists a unique $g : \mathrm{List}_{\mathbb{N}} \to A$ s.t. the following diagram commutes:

$$\begin{tikzcd}
\mathrm{F}_{\mathrm{List}}(\mathrm{List}_{\mathbb{N}}) \arrow[r, "\mathrm{intro}"] \arrow[d, "\mathrm{F}_{\mathrm{List}}\, g"'] & \mathrm{List}_{\mathbb{N}} \arrow[d, "\exists ! g"] \\
\mathrm{F}_{\mathrm{List}}\, A \arrow[r, "f"] & A
\end{tikzcd}$$

Consider now the specific category, in which objects are elements of Set (where definitionally equal sets are identified) derivable in Martin-Löf type theory. Let morphism $f : A \to B$ be functions of this type derivable in type theory. Let $f, f' : A \to B$. Consider f equal to f' as morphisms in category theoretic diagrams, if and only if f, f' are equal extensionally, i.e. $\forall a : A. f\ a ==_B f'\ a$, where $==_B$ is the intensional equality type. Assume the type theoretic rules for $\mathrm{List}_{\mathbb{N}}$. Let $\mathrm{intro} : \mathrm{F}_{\mathrm{List}}(\mathrm{List}_{\mathbb{N}}) \to \mathrm{List}_{\mathbb{N}}$, $\mathrm{intro}\ \overline{\mathrm{nil}} = \mathrm{nil}$ and $\mathrm{intro}\ (\overline{\mathrm{cons}}\ n\ l) = \mathrm{cons}\ n\ l$. Then $(\mathrm{List}_{\mathbb{N}}, \mathrm{intro})$ is an initial $\mathrm{F}_{\mathrm{List}}$-algebra: It is obviously an $\mathrm{F}_{\mathrm{List}}$-algebra. Furthermore, assume (A, f) is another $\mathrm{F}_{\mathrm{List}}$-algebra. Then we can define using the elimination rule for $\mathrm{F}_{\mathrm{List}}$ a function $g : \mathrm{List}_{\mathbb{N}} \to A$ such that $g\ \mathrm{nil} = f\ \overline{\mathrm{nil}}$, $g\ (\mathrm{cons}\ n\ l) = f\ (\overline{\mathrm{cons}}\ n\ (g\ l))$. It follows in type theory that g is the unique $\mathrm{F}_{\mathrm{List}}$-algebra morphism $g : (\mathrm{List}_{\mathbb{N}}, \mathrm{intro}) \to (A, f)$: That it is a $\mathrm{F}_{\mathrm{List}}$-algebra morphism is obvious. Further, if there is any other $\mathrm{F}_{\mathrm{List}}$-algebra morphism $g' : (\mathrm{List}_{\mathbb{N}}, \mathrm{intro}) \to (A, f)$, one can show by induction on $l : \mathrm{List}_{\mathbb{N}}$ (which corresponds to the elimination rule for $\mathrm{List}_{\mathbb{N}}$) $\forall l : \mathrm{List}_{\mathbb{N}}. g(l) ==_{\mathrm{List}_{\mathbb{N}}} g'(l)$, so g and g' are equal morphisms.

Therefore the rules of type theory for $\mathrm{List}_{\mathbb{N}}$ imply the principle that $\mathrm{List}_{\mathbb{N}}$ is an initial algebra. One can show as well that the principle of $(\mathrm{List}_{\mathbb{N}}, \mathrm{intro})$ being an initial algebra implies the type theoretic rules for $\mathrm{List}_{\mathbb{N}}$. However, this direction requires extensional equality. This result is in fact a special case of [16]. The type theoretic rules for $\mathrm{List}_{\mathbb{N}}$ and the principle of $(\mathrm{List}_{\mathbb{N}}, \mathrm{intro})$ being an initial algebra are therefore extensionally equivalent, but are intensionally different (although we have no formal proof for this). In this sense we can regard the type theoretic rules without extensional equality as one possible representation of the rules of an initial algebra.

1.3 Weakly Final Coalgebras

Colist. We will introduce the type of colists, which are elements which can be unfolded to potentially infinite lists of natural numbers. Colists will be defined as weakly final coalgebras. Coalgebras are the dual of algebras, and are obtained by inverting the direction of the arrows in the category theoretic formulation of algebras. An $\mathrm{F}_{\mathrm{List}}$-coalgebra is a pair (A, f) where $f : A \to \mathrm{F}_{\mathrm{List}}\, A$, and as for algebras we sometimes omit f when it is obvious from the context. An $\mathrm{F}_{\mathrm{List}}$-coalgebra morphism between coalgebras (A, f) and (B, g) is a function $h : A \to B$ s.t. the following diagram commutes:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "h"'] & \mathrm{F}_{\mathrm{List}}\, A \arrow[d, "\mathrm{F}_{\mathrm{List}}\, h"] \\ B \arrow[r, "g"'] & \mathrm{F}_{\mathrm{List}}\, B \end{tikzcd}$$

A final $\mathrm{F}_{\mathrm{List}}$-coalgebra $(\mathrm{coList}, \mathrm{case})$ is a terminal object in the category of $\mathrm{F}_{\mathrm{List}}$-coalgebras. Therefore, it is an $\mathrm{F}_{\mathrm{List}}$-coalgebra. Furthermore, for any other coalgebra (A, f), i.e. $f : A \to \mathrm{F}_{\mathrm{List}}\, A$ there exists a unique coalgebra morphism $g : (A, f) \to (\mathrm{coList}, \mathrm{case})$, i.e. a unique $g : A \to \mathrm{coList}$ s.t. the following diagram commutes:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "\exists! g"'] & \mathrm{F}_{\mathrm{List}}\, A \arrow[d, "\mathrm{F}_{\mathrm{List}}\, g"] \\ \mathrm{coList} \arrow[r, "\mathrm{case}"'] & \mathrm{F}_{\mathrm{List}}(\mathrm{coList}) \end{tikzcd}$$

Weakly final $\mathrm{F}_{\mathrm{List}}$-coalgebras are weakly terminal objects in the category of $\mathrm{F}_{\mathrm{List}}$-coalgebras, which means that we omit the condition that g as above is unique. Assume in the following $(\mathrm{coList}, \mathrm{case})$ is a weakly final $\mathrm{F}_{\mathrm{List}}$-coalgebra.

The function case : $\mathrm{coList} \to (\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, \mathrm{coList}))$ determines for an element of coList whether it is of the form $\overline{\mathrm{nil}}$ or $(\overline{\mathrm{cons}}\, n\, l)$. Note that we can apply case to l again. So an element of coList is an element which can, by iteratively applying case to it, be unfolded to a potentially infinite list. For instance an element a : coList s.t. case $a = \overline{\mathrm{cons}}\; 0\; a$ represents what would in a framework of infinite terms be the infinite list (cons 0 (cons 0 (cons 0 $\cdots$))).

Codata types and guarded recursion. In functional programming, codata types ([49]) are often considered as variants of algebraic data types which allow the formation of infinitely many applications of constructors. For instance one could define the codata type of colists, which has constructors nil and cons. Then it is possible to have infinite nesting of cons and define a colist (cons 0 (cons 0 (cons 0 $\cdots$))) directly. One sees immediately that this destroys normalisation. We will see below that decidable type checking is not possible, if we assume that each element of a coalgebra is introduced by a constructor. Coalgebras are a version of codata types, where elements are not per se infinitary, but unfold to infinite objects.

Relationship to guarded recursion. Guarded recursion was introduced by T. Coquand in [10] in a setting of infinitary terms. Bertot and Castéran use in Chapter 13 of the "Coq-book" [6] guarded recursion and codata types extensively for the development of infinite objects and proofs for these objects. Guarded recursion allows to define elements of codata types recursively, by allowing full recursion, as long as recursive calls are guarded by at least one (possibly more) constructors of the codata type in question, and no other functions are applied to the result of a recursive call. A simple form of guarded recursion is where we always have one recursive call guarded by exactly one constructor.

We can see now that in the coalgebraic setting the existence of the $\mathrm{F}_{\mathrm{List}}$-coalgebra morphism $g : A \to \mathrm{coList}$ for any $\mathrm{F}_{\mathrm{List}}$-coalgebra (A, f) corresponds to this simple form of guarded recursion: We have

$$\mathrm{case}\ (g\ a) = \begin{cases} \overline{\mathrm{nil}} & \text{if } f\ a = \overline{\mathrm{nil}}, \\ \overline{\mathrm{cons}}\ n\ (g\ a') & \text{if } f\ a = \overline{\mathrm{cons}}\ n\ a'. \end{cases}$$

By choosing suitable f we can therefore define $g : A \to \mathrm{coList}$ by guarded recursion, s.t. for $a : A$ we have case $(g\ a) = \overline{\mathrm{nil}}$ or case $(g\ a) = \overline{\mathrm{cons}}\ n\ (g\ a')$. Which of the two cases holds and the choice of n and a' can be decided depending on a. Note that there are no conditions on a' to be smaller than a. This principle is the simple form of the principle of guarded recursion. The difference to the setting using codata types is that $(g\ a)$ is not equal to nil or $(\mathrm{cons}\ n\ (g\ a'))$, but unfolds when applying case to it to an element having the shape $\overline{\mathrm{nil}}$ or $(\overline{\mathrm{cons}}\ n\ (g\ a'))$.

An example of guarded recursion is that we can define a function $g : \mathbb{N} \to \mathrm{coList}$ s.t. case $(g\ n) = \overline{\mathrm{cons}}\ n\ (g\ (n+1))$. Then $(g\ 0)$ represents the infinite list $(\mathrm{cons}\ 0\ (\mathrm{cons}\ 1\ (\mathrm{cons}\ 2\ \cdots)))$.

Extended guarded recursion. Let $(\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$ be the set having elements $\overline{\mathrm{nil}'}$, $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a)$ for $n : \mathbb{N}, a : A$ and $(\overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l)$ for $n : \mathbb{N}, l : \mathrm{coList}$. We are going to show that, if $g : A \to (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$, then we can define a function $f : A \to \mathrm{coList}$ s.t.

$$\mathrm{case}\ (f\ a) = \begin{cases} \overline{\mathrm{nil}} & \text{if } g\ a = \overline{\mathrm{nil}'}, \\ \overline{\mathrm{cons}}\ n\ (g\ a') & \text{if } g\ a = \overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a', \\ \overline{\mathrm{cons}}\ n\ l & \text{if } g\ a = \overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l \end{cases}$$

So $(g\ a)$ decides whether $(f\ a)$ is of nil-shape (constructor $\overline{\mathrm{nil}'}$); of cons-shape with a recursive call to $(g\ a')$ (therefore the name $\overline{\mathrm{cons}^{\mathrm{r}}}$); or of non-recursive cons-shape (therefore the name $\overline{\mathrm{cons}^{\mathrm{n}}}$). This principle adds to the principle of guarded recursion the possibility of defining $(\mathrm{case}\ (f\ a))$ by a non-recursive cons shape.

We show the existence of f just given, provided that coList is a final coalgebra.

Here $(\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$ will be a notation for the disjoint union $(\mathbf{1} + ((\mathbb{N} \times A) + (\mathbb{N} \times \mathrm{coList})))$ where $\overline{\mathrm{nil}'} := \mathrm{inl}\ *$, $\overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a := \mathrm{inr}\ (\mathrm{inl}\ \langle n, a \rangle)$ and $\overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l := \mathrm{inr}\ (\mathrm{inr}\ \langle n, l \rangle)$.

Assume g as just given. Define $A' := A + \mathrm{coList}$, and $g' : A' \to (\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, A'))$,

$$g'\ (\mathrm{inl}\ a) = \begin{cases} \overline{\mathrm{nil}} & \text{if } f\ a = \overline{\mathrm{nil}'}\ , \\ \overline{\mathrm{cons}}\, n\ (\mathrm{inl}\ a') & \text{if } f\ a = \overline{\mathrm{cons}^{\mathrm{r}}}\, n\ a'\ , \\ \overline{\mathrm{cons}}\, n\ (\mathrm{inr}\ l) & \text{if } f\ a = \overline{\mathrm{cons}^{\mathrm{n}}}\, n\ l\ . \end{cases}$$

$$g'\ (\mathrm{inr}\ l)\ = \begin{cases} \overline{\mathrm{nil}} & \text{if case } l = \overline{\mathrm{nil}}\ , \\ \overline{\mathrm{cons}}\, n\ (\mathrm{inr}\ l') & \text{if case } l = \overline{\mathrm{cons}}\, n\ l'\ . \end{cases}$$

Let $f' : A' \to \mathrm{coList}$ be the coalgebra morphism such that the following diagram commutes:

$$\begin{tikzcd} A' \arrow[r, "g'"] \arrow[d, "f'"'] & \mathrm{F_{List}}\, A' \arrow[d, "\mathrm{F_{List}}\, f'"] \\ \mathrm{coList} \arrow[r, "\mathrm{case}"'] & \mathrm{F_{List}}(\mathrm{coList}) \end{tikzcd}$$

If coList is a final coalgebra, then one can see that $(f'\ (\mathrm{inr}\ l))$ is equal to l. The reason for defining $(f'\ (\mathrm{inr}\ l))$ was that it allows to replace the non-recursive call to l in f by a recursive call to $(f'\ (\mathrm{inr}\ l))$. Let $f := f' \circ \mathrm{inl} : A \to \mathrm{coList}$. We obtain that f indeed fulfils the desired equations.

We call the principle that, for every $g : A \to (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$ we can define $f : A \to \mathrm{coList}$ such that the equations for (case $(f\ a)$) just given hold the principle of extended guarded recursion. Full details will be found in [45]. Note that we chose in the third case not to escape directly to an element l : coList, but only to an element l such that case $l = \overline{\mathrm{cons}}\ n\ l'$ for given n, l'. The reason for this is that this allows to define cons as given before.

Giménez shows in [19] how to derive more general forms of guarded recursion for coalgebras.

The coconstructors nil, cons. In case of final coalgebras it follows (e.g. [30], Lemma 2.3.3) that case : $\mathrm{coList} \to \mathrm{F_{List}}(\mathrm{coList})$ is an isomorphism. Let case^{-1} be its inverse and define $\mathrm{nil} := \mathrm{case}^{-1}\ \overline{\mathrm{nil}}$, $\mathrm{cons}\ n\ l := \mathrm{case}^{-1}\ (\overline{\mathrm{cons}}\ n\ l)$. Then we have that case nil $= \overline{\mathrm{nil}}$ and case $(\mathrm{cons}\ n\ l) = \overline{\mathrm{cons}}\ n\ l$. cons^{-1} is surjective, so every l : coList is equal to nil or $(\mathrm{cons}\ n\ l')$ for some n, l'. Especially, case $l = \overline{\mathrm{nil}}$ if and only if $l = \mathrm{nil}$, and case $l = \overline{\mathrm{cons}}\ n\ l'$ if and only if $l = \mathrm{cons}\ n\ l'$. By iterating it we obtain that if l : coList, then for every k we have that $l = \mathrm{cons}\ n_1\ (\mathrm{cons}\ n_1\ \cdots\ (\mathrm{cons}\ n_i\ \mathrm{nil}) \cdots)$ for some $i < k$ and $n_1, \ldots, n_i : \mathbb{N}$ or l is equal to $(\mathrm{cons}\ n_1\ (\mathrm{cons}\ n_1\ \cdots\ (\mathrm{cons}\ n_k\ l') \cdots))$ for some $n_1, \ldots, n_k : \mathbb{N}$ and l' : coList. Roughly speaking, an element of coList is a potentially infinite list of natural numbers. Furthermore, the principle of extended guarded recursion can be rewritten as follows: We can define $g : A \to \mathrm{coList}$ s.t. depending on a we can choose $g\ a = \mathrm{nil}$, $g\ a = \mathrm{cons}\ n\ (g\ a'))$ for some n, a' or $g\ a = \mathrm{cons}\ n\ l$ for some n, l.

Bisimilarity as equality. A weakly final $\mathrm{F_{List}}$-coalgebra (coList, case) is final if and only if equality on coList is bisimilarity. Here bisimilarity on colists is the largest relation $\sim$ s.t., if $l \sim l'$, then $(\mathrm{case}\ l) = \overline{\mathrm{nil}} = (\mathrm{case}\ l')$ or $(\mathrm{case}\ l) = (\overline{\mathrm{cons}}\ n\ l_0)$ and $(\mathrm{case}\ l') = (\overline{\mathrm{cons}}\ n\ l'_0)$ for some $l_0 \sim l'_0$. Bisimilarity can be introduced as an indexed coalgebra (as will e.g. be shown in the current context in [45]). Bisimilarity is equality on final $\mathrm{F}_{\mathrm{List}}$-coalgebras (see e.g. [30], Theorem 3.4.1) and one can easily show as well that weakly final coalgebras are final coalgebras, if bisimilar elements are equal. Full details will be presented in [45].

Coconstructors in case of weakly final coalgebras. In case of weakly final coalgebras we can define nil by case $\mathrm{nil} = \overline{\mathrm{nil}}$. We can define $(\mathrm{cons}\, n\, l)$ s.t. case $(\mathrm{cons}\, n\, l) = \overline{\mathrm{cons}}\, n\, l'$ for some l which is bisimilar to l'. This can be done by defining $A = (\mathbb{N} \times \mathrm{coList}) + \mathrm{coList}$, $f : A \to (\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, A))$, $f\ (\mathrm{inl}\ \langle n, l\rangle) = \overline{\mathrm{cons}}\ n\ (\mathrm{inr}\ l)$, $f\ (\mathrm{inr}\ l) = \overline{\mathrm{nil}}$ if case $l = \overline{\mathrm{nil}}$, $f\ (\mathrm{inr}\ l) = \overline{\mathrm{cons}}\ n\ (\mathrm{inr}\ l')$ if case $l = \overline{\mathrm{cons}}\ n\ l'$. Then one can easily see that $f\ (\mathrm{inr}\ l) \sim l$, and define therefore $\mathrm{cons}\ n\ l = f\ (\mathrm{inl}\ \langle n, l\rangle)$. We obtain case $(\mathrm{cons}\ n\ l) = \overline{\mathrm{cons}}\ n\ l'$ for some $l' \sim l$.

Combining the above we obtain a version of case^{-1} as well. The function case^{-1} is not surjective. In case of $\overline{\mathrm{cons}}$, the equality holds only up to bisimilarity. If we add the principle of extended guarded recursion to weakly final coalgebras, we can define case^{-1} in such a way that the equality holds definitionally (however case^{-1} will not be surjective): Define $\mathrm{case}^{-1} : \mathrm{F}_{\mathrm{List}}(\mathrm{coList}) \to \mathrm{coList}$, case $(\mathrm{case}^{-1}\ \overline{\mathrm{nil}}) = \overline{\mathrm{nil}}$, case $(\mathrm{case}^{-1}\ (\overline{\mathrm{cons}}\ n\ l)) = \overline{\mathrm{cons}}\ n\ l$. In order to allow this definition we defined the non-recursive case in case of extended guarded recursion the way we did it.

Undecidability results. Bisimilarity on $\mathrm{F}_{\mathrm{List}}$-coalgebras is undecidable: Define $\mathrm{toColist} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathrm{coList}$, case $(\mathrm{toColist}\ f\ n) = \overline{\mathrm{cons}}\ (f\ n)\ (\mathrm{toColist}\ f\ (n+1))$. Therefore, in case of final coalgebras we have $\mathrm{toColist}\ f\ n = \mathrm{cons}\ (f\ n)\ (\mathrm{cons}\ (f\ (n+1))\ (\mathrm{cons}\ (f\ (n+2))\ \cdots))$. Now it follows immediately that f, g are extensionally equal if and only if $(\mathrm{toColist}\ f\ 0) \sim (\mathrm{toColist}\ g\ 0)$. Since extensional equality on $\mathbb{N} \to \mathbb{N}$ is undecidable, bisimilarity is undecidable as well. Therefore, if we want decidable definitional equality, we cannot define final coalgebras, only weakly final coalgebras.

In [45] we will show that the assumption that case^{-1} is surjective results in an undecidable equality as well. So, if we want decidable equality on a weakly final coalgebra, we cannot assume that every element of it is of the form nil or $(\mathrm{cons}\ n\ l)$ for some n, l. This implies that pattern matching on coalgebras in the setting of decidable type checking is misleading, since it suggests that every element of a coalgebra is introduced by a coconstructor, and therefore contains the hidden assumption that case^{-1} is surjective.

Problem of Subject Reduction. The problems of pattern matching have been discussed intensively on the Agda email list. Giménez [18, Sect. 3.4] discovered that dependent case distinction results in a problem with subject reduction. Later Nicolas Oury found a very short program in a previous version of Agda, which exposes this problem, and which he orally communicated to N. Danielsson, who then posted it in [11]. Oury then converted it to Coq and posted it in [42]. A detailed analysis can be found in [38]. There were as well intensive discussions on the Agda and Coq club mailing lists, to which the author contributed. Some changes have been made to Agda which avoid this problem, see [4]. The author would prefer a more aesthetically clear solution, based on what is presented in his article. The goal would be to have a solution which presents algebras and coalgebras in a symmetric way. In Coq the problem of subject reduction seems to persist.

Type theoretic rules for weakly final coalgebras. Because of the undecidability of equality in final coalgebras, we can only introduce rules for weakly final coalgebras, if we want to preserve decidable type checking. For weakly final coalgebras we can still derive the principle of extended guarded recursion, but the equations we want to satisfy will only hold up to bisimilarity as equality. For initial algebras we observed that the fact that the type theoretic rules for $\mathrm{List}_{\mathbb{N}}$ are extensionally equal but intensionally stronger than the rules for $\mathrm{List}_{\mathbb{N}}$ being an initial algebra. In the same way we are defining rules for coList which are up to bisimilarity equivalent but without bisimilarity as equality stronger than the rules for coList being a weakly final coalgebra. The principle of a weakly final coalgebra plus the principle that bisimilarity is equality is equivalent to the principle of a final coalgebra. If we take the rules for coiteration derived from the diagram, we get type theoretic rules which are up to bisimilarity equivalent to the rules of a weakly final coalgebra. If we extend the principle of guarded recursion to extended guarded recursion, we get a principle which is up to bisimilarity derivable, but without it stronger than the principle of simple guarded recursion. Therefore extended guarded recursion plus the principle of $(\mathrm{coList}, \mathrm{case})$ being a coalgebra is without bisimilarity as equality stronger, with it equivalent to that of a weakly final coalgebra. As in case of $\mathrm{List}_{\mathbb{N}}$ we use the rules of $(\mathrm{coList}, \mathrm{case})$ being a coalgebra augmented by the principle of extended guarded recursion as one possible type theoretic formulation of the rules for $(\mathrm{coList}, \mathrm{case})$ being a weakly final coalgebra. It is not the only possible one. In general one can think of adding rules which imply further definitional equalities, which are provable up to bisimilarity, as long as the rules behave well (we have decidable type checking, subject reduction and other good properties). One reason for including extended guarded recursion is that it allows us to define the coconstructor cons by defining $\mathrm{cons}\, n\, l : \mathrm{coList}$, s.t. $\mathrm{case}\, (\mathrm{cons}\, n\, l) = \overline{\mathrm{cons}}\, n\, l)$.

For completeness, we introduce rules for dealing with $(\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(X,Y))$ and $(\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z'))$. (Note that if as above we treat these definitions as abbreviations, these rules can be derived from the rules for $\mathbf{1}$, $+$ and $\times$). We borrow notations for case distinction from [9]:

Assume in the following rules $X, Y, Z, Z' : \mathrm{Set}$.

Formation rule: $\quad (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z')) : \mathrm{Set}$.

Introduction rules:

$$\begin{array}{lll}
\overline{\mathrm{nil}'} & : & (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z')) \ , \\
\overline{\mathrm{cons}^{\mathrm{r}}} & : X \to Y \ \to & (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z')) \ , \\
\overline{\mathrm{cons}^{\mathrm{n}}} & : Z \to Z' \ \to & (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z')) \ .
\end{array}$$

Elimination rule:

$$\frac{\begin{array}{c} x : (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z')) \Rightarrow C(x) : \mathrm{Set} \\ \mathrm{step}_{\overline{\mathrm{nil}'}} : C(\mathrm{nil}') \\ x : X, y : Y \Rightarrow \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{r}}}}(x,y) : C(\overline{\mathrm{cons}^{\mathrm{r}}}\ x\ y) \\ z : Z, z' : Z' \Rightarrow \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{n}}}}(z,z') : C(\overline{\mathrm{cons}^{\mathrm{n}}}\ z\ z') \end{array}}{\left\{ \begin{array}{ll} \overline{\mathrm{nil}'} & \mapsto \mathrm{step}_{\overline{\mathrm{nil}'}} \\ \overline{\mathrm{cons}^{\mathrm{r}}}\ x\ y & \mapsto \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{r}}}}(x,y) \\ \overline{\mathrm{cons}^{\mathrm{n}}}\ z\ z' & \mapsto \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{n}}}}(z,z') \end{array} \right\} : (x : (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(X,Y) + \overline{\mathrm{cons}^{\mathrm{n}}}(Z,Z'))) \to C(x)}$$

Equality rules:

$$\begin{array}{ll} \{\cdots\}\ \overline{\mathrm{nil}'} & = \mathrm{step}_{\overline{\mathrm{nil}'}} \\ \{\cdots\}\ (\overline{\mathrm{cons}^{\mathrm{r}}}\ x\ y) & = \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{r}}}}(x,y) \\ \{\cdots\}\ (\overline{\mathrm{cons}^{\mathrm{n}}}\ z\ z') & = \mathrm{step}_{\overline{\mathrm{cons}^{\mathrm{n}}}}(z,z') \end{array}$$

where $\{\cdots\}$ is the expression introduced in the elimination rule

Now we can define the rules for colist:

Formation rule: $\mathrm{coList} : \mathrm{Set}$

Elimination rule: $\mathrm{case} : \mathrm{coList} \to (\overline{\mathrm{nil}} + \overline{\mathrm{cons}}(\mathbb{N}, \mathrm{coList}))$

Introduction rule:

$$\frac{A : \mathrm{Set}}{\begin{array}{l} \mathrm{intro}_A : (A \to (\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N},A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N},\mathrm{coList})) \\ \qquad \to A \to \mathrm{coList} \end{array}}$$

Equality rule:

$$\mathrm{case}\ (\mathrm{intro}_A\ f\ a) = \left\{ \begin{array}{ll} \overline{\mathrm{nil}'} & \mapsto \overline{\mathrm{nil}} \\ \overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a' & \mapsto \overline{\mathrm{cons}}\ n\ (\mathrm{intro}_A\ f\ a') \\ \overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l & \mapsto \overline{\mathrm{cons}}\ n\ l \end{array} \right\} (f\ a)$$

Note that the introduction rule is complex because a generic form of guarded recursion in the same way that the elimination rule for algebraic data types is complicated, because it is generic. Specific instances can be described more easily. For instance we can define

$$\begin{array}{l} \mathrm{toColist} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathrm{coList} \\ \mathrm{case}\ (\mathrm{toColist}\ f\ n) = \overline{\mathrm{cons}}\ (f\ n)\ (\mathrm{toColist}\ f\ (n+1)) \end{array}$$

The coconstructors nil and cons can be defined by

$$\begin{array}{ll} \mathrm{nil} : \mathrm{coList} & \mathrm{cons} : \mathbb{N} \to \mathrm{coList} \to \mathrm{coList} \\ \mathrm{case}\ \mathrm{nil} = \overline{\mathrm{nil}} & \mathrm{case}\ (\mathrm{cons}\ n\ l) = \overline{\mathrm{cons}}\ n\ l \end{array}$$

We observe that the elimination rules are simple whereas the introduction rules seem to be complicated and refer to all sets. This is dual to the setting for initial algebras where the introduction rules are simple and the elimination rules refer to all sets. So a weakly final coalgebra is given by its elimination rules, which essentially expresses: elements of coList are programs, to which we can apply case and obtain $\overline{\mathrm{nil}}$ or $(\overline{\mathrm{cons}}\ n\ l)$ for some other colist l.

Problems with dependent case distinction. McBride [38] discussed dependent case distinction, as it occurs in the PhD thesis by Giménez [18] and is implemented in Coq. In our notation it reads

$$\frac{x : \mathrm{coList} \Rightarrow B(x) : \mathrm{Set}}{\begin{array}{l}\mathrm{depcase}_B : (\mathrm{step}_{\mathrm{nil}} : B(\mathrm{nil}))\\ \qquad\qquad \to (\mathrm{step}_{\mathrm{cons}} : (n : \mathbb{N}, l : \mathrm{coList}_{\mathbb{N}}) \to B(\mathrm{cons}\, n\, l))\\ \qquad\qquad \to (l : \mathrm{coList})\\ \qquad\qquad \to B(l)\end{array}}$$

$$\begin{array}{ll}\mathrm{depcase}_B\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ \mathrm{nil} & = \mathrm{step}_{\mathrm{nil}}\\ \mathrm{depcase}_B\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ (\mathrm{cons}\, n\, l) & = \mathrm{step}_{\mathrm{cons}}\ n\ l\end{array}$$

There is an equality rule missing, namely for an element introduced by intro. Such a rule should be (the case for $\overline{\mathrm{cons}^{\mathrm{n}}}$ was added by the author to stay in accordance with the rest of the current article):

$$\begin{array}{l}\mathrm{depcase}_B\ \mathrm{step}_{\mathrm{nil}}\ \mathrm{step}_{\mathrm{cons}}\ (\mathrm{intro}_A\ f\ a)\\ \quad = \left\{\begin{array}{lll}\overline{\mathrm{nil}'} & \mapsto & \mathrm{step}_{\mathrm{nil}}\\ \overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a' & \mapsto & \mathrm{step}_{\mathrm{cons}}\ n\ (\mathrm{intro}_A\ f\ a')\\ \overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l & \mapsto & \mathrm{step}_{\mathrm{cons}}\ n\ l\end{array}\right\}\ (f\ a)\end{array}$$

McBride states that this is the source of the problem discovered/communicated by Giménez/Oury/Danielsson [18, 42, 11]. As McBride observes, it does not even type check: in case of $f\ a = \overline{\mathrm{nil}'}$, the two sides of the equations have types $B(\mathrm{intro}_A\ f\ a)$ and $B(\mathrm{nil})$, but $\mathrm{intro}_A\ f\ a \neq \mathrm{nil}$.

As observed by McBride dependent case distinction results, if we omit the last rule, in non-canonical terms for the intensional equality type. In fact the situation is even worse: We get non-canonical elements of $\mathbb{N}$ in normal form: Let $f = \mathrm{depcase}_{(x)\mathbb{N}}\ 0\ ((n,l)0) : \mathrm{coList} \to \mathbb{N}$. Let $\mathrm{zeroStream} = \mathrm{intro}_{\mathbf{1}}\ ((x)(\overline{\mathrm{cons}^{\mathrm{r}}}\ 0\ *))\ * :$ coList. We have that $(f\ \mathrm{zeroStream})$ is a non-canonical closed element of $\mathbb{N}$ in normal form. The reason is of course that we do not have an equality rule for depcase applied to an element introduced by intro.

The underlying problem is that dependent case distinction expresses that every element of coList is of the form nil or $(\mathrm{cons}\, n\, l)$, i.e. that case^{-1} is surjective. In order to repair this problem, McBride suggests to switch to observational type theory. This means essentially to define for all types a propositional equality together with some additional axioms. In case of coList, this equality would be bisimilarity. Since, if we add to weakly final coalgebras bisimilarity as equality, we obtain final coalgebras, the problem vanishes. However, it does not solve the problem, what the correct rules regarding definitional equalities in intensional type theory are.

1.4 Meaning explanations for coalgebraic types as determined by their elimination rules.

We give now meaning explanations for coList based on the principle that elements of coalgebras are determined by their elimination rules. coList is a set. Elements of coList are programs l, which, if we apply case to them, evaluate to $\overline{\mathrm{nil}}$ or $(\overline{\mathrm{cons}}\ n\ l')$ for some n in $\mathbb{N}$, and some other element l' of coList. Note that we do not demand that l' is defined before l. Several elements of coList might be introduced simultaneously. Two elements l, l' of coList are equal if after applying case to it, both evaluate to $\overline{\mathrm{nil}}$ or they evaluate to $(\overline{\mathrm{cons}}\ n\ l_0)$ and $(\overline{\mathrm{cons}}\ n'\ l_0')$ where n, n' are equal elements of $\mathbb{N}$ and l_0, l_0' are equal elements of coList. Again we do not demand that the equality of l_0, l_0' is established before the equality of l, l' is established.
Assume A is a set and f a function mapping an element of A to an element of $(\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$. Then for every $a : A$, $(\mathrm{intro}_A\ f\ a)$ is an element of coList. For this we determine $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$: Compute $(f\ a)$. If $(f\ a)$ evaluates to $\overline{\mathrm{nil}'}$ then $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ evaluates to $\overline{\mathrm{nil}}$. If $(f\ a)$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a')$, then $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ evaluates to $(\overline{\mathrm{cons}}\ n\ (\mathrm{intro}_A\ f\ a'))$. If $(f\ a)$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{n}}}\ n\ l)$, then $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ evaluates to $(\overline{\mathrm{cons}}\ n\ l)$.
Assume A, A' are equal sets, f, f' map elements of A to equal elements of $(\overline{\mathrm{nil}'} + \overline{\mathrm{cons}^{\mathrm{r}}}(\mathbb{N}, A) + \overline{\mathrm{cons}^{\mathrm{n}}}(\mathbb{N}, \mathrm{coList}))$. For all a, a' equal elements of A we have that $(\mathrm{intro}_A\ f\ a)$ and $(\mathrm{intro}_{A'}\ f'\ a')$ are equal elements of coList: Assume a and a' are equal elements of A.

Assume $(f\ a)$ evaluates to $\overline{\mathrm{nil}'}$. Then, since f is equal to f' and a is equal to a', $f'\ a'$ evaluates to $\overline{\mathrm{nil}'}$ as well. Then $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ and $(\mathrm{case}\ (\mathrm{intro}_{A'}\ f'\ a'))$ both evaluate to the same element $\overline{\mathrm{nil}}$.

Assume $(f\ a)$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n\ a_0)$. Then $(f'\ a')$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n'\ a_0')$ for some n' equal to n and a_0' equal to a_0. Then $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ evaluates to $(\mathrm{cons}\ n\ (\mathrm{intro}_A\ f\ a_0))$ and $(\mathrm{case}\ (\mathrm{intro}_{A'}\ f'\ a'))$ evaluates to $(\overline{\mathrm{cons}}\ n'\ (\mathrm{intro}_{A'}\ f'\ a_0'))$. n and n' are equal elements of $\mathbb{N}$, and $(\mathrm{intro}_A\ f\ a_0)$ and $(\mathrm{intro}_{A'}\ f'\ a_0')$ are equal elements of coList. Therefore $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a))$ and $(\mathrm{case}\ (\mathrm{intro}_{A'}\ f'\ a'))$ evaluate to equal elements.

Assume $(f\ a)$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n\ l)$. Then $(f\ a')$ evaluates to $(\overline{\mathrm{cons}^{\mathrm{r}}}\ n'\ l')$ for some n equal to n' and l equal to l'. Therefore $(\mathrm{case}\ l)$ and $(\mathrm{case}\ l')$ and therefore as well $(\mathrm{case}\ (\mathrm{intro}_A\ f\ a)$ and $(\mathrm{case}\ (\mathrm{intro}_{A'}\ f'\ a')$ evaluate to equal elements. Therefore $(\mathrm{intro}_A\ f\ a)$ and $(\mathrm{intro}_{A'}\ f'\ a')$ are equal.
Function sets as determined by their elimination rules. We can see now that the elements of the function type of the logical framework are as well introduced by their elimination rules: Assume A is a set and $B(x)$ is a set depending on elements x of A. Then $(x : A) \to B(x)$ is a set. An element of $(x : A) \to B(x)$ is a program t which, when applied to an element a of A evaluates to an element of $B(a)$. Two elements t, t' of $(x : A) \to B(x)$ are equal, if, when applied to an element a of A, they evaluate to equal elements of $B(a)$. Assume that for every x of A we have that t is an element of $B(x)$. Then $(x)t$ is the following element of $(x : A) \to B(x)$: If applied to $a : A$ it first substitutes in t the variable x by a. Let the result be s. Then s is evaluated,

which is the result returned. Since for x of A, t is an element of $B(x)$, s is an element of $B(a)$. So $(x)t$ is an element of $(x:A) \to B$. Assume that t,t' are equal elements of $B(x)$, depending on x of type A. Then if $(x)t$ and $(x)t'$ are applied to an element a of type A, we obtain s, s' which are equal elements of $B(a)$. So $(x)t$ and $(x)t'$ are equal elements of $(x:A) \to B(x)$.

More advanced examples of coalgebras. coList is only the simplest example of a coalgebra. More advanced examples are the definition of bisimilarity on colists or on other transition systems. In [22, 23, 24, 25, 26] we discussed how to define state-dependent interactive programs in Martin-Löf type theory, and in [25] we showed how to define them as an indexed coalgebra. More examples can be found for instance in Chapter 13 of [6].

1.5 Conclusion

We have seen that coalgebras can be introduced in Martin-Löf type theory using formation, elimination, introduction and equality rules. Meaning explanations can be given by defining as elements of coalgebras those which allow elimination rules. One can then explain that the introduction rules indeed introduce elements of the coalgebra. So elements of coalgebras are given by their elimination rules, the introduction rules can be considered as being derived. This is similar to algebraic data types, for which the elements are given by their introduction rules, and the elimination rules are derived. We have seen as well that the elements of the function types from the logical framework are as well determined by their elimination rules. One can as well develop models of coalgebras, in which coalgebras are interpreted as the set of those terms which allow to apply the elimination principle.

References

1. P. Aczel. *Non-wellfounded set theory*, volume 14. CSLI Lecture notes, Stanford, CA: Stanford University, Center for the Study of Language and Information, 1988. pp. xx+137.
2. Agda. Email list archive. Available at https://lists.chalmers.se/pipermail/agda/, 2011.
3. T. Altenkirch. Codata. Talk given at the TYPES Workshop in Jouy-en-Josas, December 2004. Available from http://www.cs.nott.ac.uk/ txa/talks/types04.pdf, 2004.
4. T. Altenkirch and N. A. Danielsson. Termination checking nested inductive and coinductive types. In *Proceedings of Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR'10*, 2010.
5. Y. Bertot. CoInduction in Coq. Available from http://arxiv.org/abs/cs/0603119, March 2006.
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
7. J. Cockett and D. Spencer. Strong categorical datatypes I. In *Category theory 1991: proceedings of an International Summer Category Theory Meeting, held June 23-30, 1991*, volume 13, page 141. American Mathematical Society, 1992.
8. J. R. B. Cockett and D. Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theoretical Computer Science*, 139(1-2):69–113, 1995.

9. R. Cockett and T. Fukushima. About charity. Technical report, Department of Computer Science, The University of Calgary, June 1992. Yellow Series Report No. 92/480/18.
10. T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806, pages 62–78. Springer Lecture Notes in Computer Science, 1994.
11. N. A. Danielsson. Codata oddity. Email posted on the Agda email list. Available from http://thread.gmane.org/gmane.comp.lang.agda/226, 2008.
12. G. F. Díez. Five observations concerning the intended meaning of the intuitionistic logical constants. *Journal of Philosophical Logic*, 29(4):409 – 424, August 2000.
13. G. F. Díez. The logic of constructivism. *Disputatio*, 12:37 – 41, May 2002.
14. G. F. Díez. Is the language of intuitionistic mathematics adequate for intuitionistic purposes? *L&PS - Logic and Philosophy of Science*, 1(1), 2003.
15. M. Dummett. *Elements of Intuitionism*. Oxford Logic Guides. Oxford University Press, 2nd edition, 2000.
16. P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1 – 47, 2003.
17. P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66:1 – 49, 2006.
18. C. E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants. (English: A calculus of infinite constructions and its application to the verification of communicating systems)*. PhD thesis, Ecole normale supérieure de Lyon, Lyon, France, 1996.
19. E. Giménez. Codifying guarded definitions with recursive schemes. In *Proceedings of the 1994 Workshop on Types for Proofs and Programs*, pages 39–59. LNCS No. 996, 1994.
20. T. Hagino. *A Categorical Programming Language*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987. Available from http://www.tom.sfc.keio.ac.jp/ hagino/thesis.pdf.
21. T. Hagino. Codatatypes in ml. *J. Symb. Comput.*, 8(6):629–650, 1989.
22. P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. Available via http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html.
23. P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic. 14th international workshop, CSL 2000*, Springer Lecture Notes in Computer Science, Vol. 1862, pages 317 – 331, 2000.
24. P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings, available via http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.
25. P. Hancock and A. Setzer. Interactive programs and weakly final coalgebras (extended version). In T. Altenkirch, M. Hofmann, and J. Hughes, editors, *Dependently typed programming*, number 04381 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2004. Available via http://drops.dagstuhl.de/opus/.
26. P. Hancock and A. Setzer. Guarded induction and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.
27. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1995.
28. B. Jacobs. Coalgebraic reasoning about classes in object-oriented languages. *Electronical Notes in Computer Science*, 11:231 – 242, 1998. Special issue on the workshop Coalgebraic Methods in Computer Science (CMCS 1998).
29. B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

30. B. Jacobs. Introduction to Coalgebra. Towards mathematics of states and oberservations. Available via http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf, 2005.
31. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.
32. F. Leclerc and C. Paulin-Mohring. Programming with streams in Coq. A case study: The sieve of eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 191–212. Springer Berlin / Heidelberg, 1994.
33. P. Martin-Löf. Infinite terms and a system of natural deduction. *Compositio Mathematica*, 24(1):93 – 103, 1972.
34. P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984.
35. P. Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73:407 – 420, 1987.
36. P. Martin-Löf. On the meaning of the logical constants and the justification of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11 – 60, May 1996.
37. P. Martin-Löf. An intuitionistic theory of types. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, pages 127 – 172, Oxford, 1998. Oxford University Press. Reprinted version of an unpublished report from 1972.
38. C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional. In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *Proceedings of the 3rd international Conference on Algebra and Coalgebra in Computer Science, CALCO'09*, Lecture Notes in Computer Science, pages 113 – 126, Berlin, Heidelberg, 2009. Springer-Verlag.
39. A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9:23:1–23:49, June 2008.
40. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory. An introduction.* Oxford University Press, 1990.
41. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
42. N. Oury. Coinductive types and type preservation. Email posted 6 June 2008 at science.mathematics.logic.coq.club. Available from https://sympa-roc.inria.fr/wws/arc/coq-club/2008-06/msg00022.html?checked_cas=2, 2008.
43. L. Paulson. A fixedpoint approach to implementing (Co) inductive definitions. In A. Bundy, editor, *Automated Deduction CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer Berlin / Heidelberg, 1994.
44. A. Setzer. Universes in type theory part I – Inaccessibles and Mahlo. In A. Andretta, K. Kearnes, and D. Zambella, editors, *Logic Colloquium '04*, pages 123 – 156. Association of Symbolic Logic, Lecture Notes in Logic 29, Cambridge University Press, 2008.
45. A. Setzer. Why codata should be replaced by coalgebras. In preparation, 2011.
46. G. Sundholm. Constructions, proofs and the meaning of logical constants. *Journal of Philosophical Logic*, 12:151–172, 1983. 10.1007/BF00247187.
47. A. Telford and D. Turner. Ensuring streams flow. In M. Johnson, editor, *Algebraic Methodology and Software Technology. 6th International Conference, AMAST'96 Sydney, Australia, December 13 – 17, 1997*, pages 509 – 523. Springer Lecture Notes in Computer Science 1349, 1997.
48. D. Turner. Elementary strong functional programming. *Funtional Programming Languages in Education*, pages 1–13, 1995.
49. D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.