Undecidability of Equality for Codata Types

Ulrich Berger and Anton Setzer

Dept. of Computer Science, Swansea University, Swansea SA2 8PP, UK,
{u.berger,a.g.setzer}@swan.ac.uk,
WWW home page: http://www.cs.swan.ac.uk/{~csulrich/,~csetzer/}

Abstract. Decidability of type checking for dependently typed languages usually requires a decidable equality on types. Since bisimilarity on (weakly final) coalgebras such as streams is undecidable, one cannot use it as the equality in type checking (which is definitional or judgemental equality). Instead languages based on dependent types with decidable type checking such as Coq or Agda use intensional equality for type checking. Two streams are definitionally equal if the underlying terms reduce to the same normal form, i.e. if the underlying programs are syntactically equivalent. For reasoning about equality of streams one introduces bisimilarity as a propositional rather than judgemental equality.

In this paper we show that it is not possible to strengthen intensional equality in a decidable way while having the property that the equality respects one step expansion, which means that a stream with head n and tail s is equal to $\mathrm{cons}(n, s)$. This property, which would be very useful in type checking, would not necessarily imply that bisimilar streams are equal, and we prove that there exist equalities with this properties which are not bisimilarity. Whereas a proof that bisimilarity on streams is undecidable is straightforward, proving that respecting one step expansion makes equality undecidable is much more involved and relies on an inseparability result for sets of codes for Turing machines. We prove this theorem both for streams with primitive corecursion and with coiteration as introduction rule.

Therefore, pattern matching on streams is, understood literally, not a valid principle, since it assumes that every stream is equal to a stream of the form $\mathrm{cons}(n, s)$. We relate this problem to the subject reduction problem found when adding pattern matching on coalgebras to Coq and Agda. We discuss how this was solved in Agda by defining coalgebras by their elimination rule and replacing pattern matching on coalgebras by copattern matching, and how this relates to the approach in Agda which uses the type of delayed computations.

Keywords: Coalgebra, weakly final coalgebras, codata, decidable type checking, Martin-Löf type theory, intensional equality, intensional type theory, dependent type theory, undecidability results, inseparability, pattern matching, copattern matching

1 Introduction

Many programs in computing are interactive in nature. We use user interfaces, text editors, data bases, interact with sensors and actuators, and communicate with other devices such as mobile phones or servers. Such programs potentially run forever – a text editor will never terminate, unless we terminate it explicitly or by accident, communication with a data base will never stop unless the server is shut down, etc. In a series of articles [HS05,HS00,HS04] Peter Hancock and the second author have shown how to represent interactive programs as non-well-founded trees. Such a connection has been observed in other contexts as well such as in order to give semantics to process algebras or describe interactive programs in functional programs using streams or monads. Because of this, non-well-founded data types play an important rôle in computer science. The usual approach to such non-well-founded structures is to represent them as coalgebraic data types.

In this paper we investigate weakly final strictly positive coalgebras in the context of dependent type theory with decidable type checking. *By coalgebras we will mean*, unless stated differently, *weakly final strictly positive coalgebras.* Decidable type checking requires that definitional equality, i.e. the equality used during type checking, is decidable. Theorem provers with decidable type checking such as Agda are very easy to use and allow to write proofs in the same way as programs in many programming languages are written. The requirement for decidable definitional equality doesn't prevent to reason about bisimilar coalgebras: one can define bisimilarity of coalgebras as a proposition, and prove that certain elements of coalgebras are bisimilar.

Coalgebras can be encoded using inductive types. However, in dependent type theory, it seems to be difficult or might even be impossible to get an encoding which gives the desired equalities w.r.t. decidable definitional equality. Therefore, it is of interest to add coalgebras explicitly to type theory. Coalgebras have been added in the form of codata types to both Coq [INR17] (see [Ber06,BC04] for their approach to coalgebras) and Agda [Nor07,Agd14]. The approach regarding coalgebras in Agda is described in [DA10]. Recently, the approach defining coalgebras by their elimination rules has been added as well to Agda, and used for implementing concepts from object based programming and graphical user interfaces in Agda [AAS17,AAS16].

In this article we answer the often asked question, whether rules for intensional equality can be strengthened so that they allow at least one step expansion: if a stream s has head a and tail s', then it should be equal to (cons a s'). Such an equality does not necessarily imply that bisimilar streams are equal – only streams, which have the same first n elements and then are equal need to be equated. We show that indeed there are equalities which are not bisimilarity, but admit one step expansion. We give a negative answer to the initial question and show that there exists no decidable equality which allows for one step expansion. While a proof that bisimilarity on streams is undecidable is straightforward, since extensional equality on functions of type $\mathbb{N} \to \mathbb{N}$ is undecidable,

this proof is much more evolved and relies on an inseparability result for sets of codes for Turing machines.

A consequence is that, if we want to stay in an intensional type theory with decidable type checking, we cannot assume that every stream is equal to (cons n s) for some n, s. Therefore, pattern matching on streams, understood literally, is not a valid principle: A definition

$$f : (s : \mathsf{Stream}) \to A[s]$$
$$f\ (\text{cons}\ n\ s) = t[n, s]$$

assumes that every stream is equal to a stream of the form (cons n a).

This explains why defining coalgebras by their introduction rules led to a subject reduction problem in both Coq and Agda [Gim96,Our08,McB09,APTS13]. This problem was solved in Agda initially by disallowing the dependency of A on s. This however restricted quite severely its usefulness. Later it was solved together with the pattern matching problem by changing the type of s to be a new type (∞ Stream) of delayed computations. We will discuss this approach in the conclusion. The latest approach taken in Agda is that coalgebras are defined by their elimination rules, and pattern matching is replaced by copattern matching. This approach has good properties: there are no restrictions on when to apply reductions, subject reduction holds, and we have complete duality between algebraic and coalgebraic data types.

Content of the Article In Sect. 2 we review the notion of codata types. We discuss, why decidable type checking and therefore a decidable definitional equality is useful. We review the problems of the codata approach (especially subject reduction) and review the approach of defining coalgebras by their elimination rules, which fixes this problem. We discuss as well the principle of primitive corecursion. In Sect. 3 we introduce encodings of streams which consist of a set of streams, functions head and tail, and an equality. Such encodings are universal if they admit the principle of primitive corecursion. Then we show in Theorem 9 that there is no decidable equality in such a universal encoding, which fulfils the condition that $\langle \text{head}, \text{tail} \rangle$ is injective, i.e., that if the heads and tails of streams are equal, then the streams are equal. It follows (Corollary 11) that it is not possible to have a universal encoding of strings such that every stream is equal to a stream of the form (cons n s). We show as well (Examples 13) that there exist universal encodings for streams such that $\langle \text{head}, \text{tail} \rangle$ is not injective, and that injectivity of $\langle \text{head}, \text{tail} \rangle$ doesn't imply that the equality is bisimilarity. The proof of the main theorem makes essential use of the principle of primitive corecursion, and the question is whether it holds if we have coiteration instead. In Sect. 4 we show (Theorem 17) that this is the case. The paper ends with a conclusion, a discussion of related work, and a discussion of the use of codata types in theorem proving and programming. In particular we will discuss how codata types can be reduced to coalgebras, and how notations such as the so called "musical notation" in Agda can be understood as syntactic sugar, which allows to keep most of the benefits of the codata approach when working with coalgebras.

2 Codata Types and Coalgebras

Codata Types. In the codata approach, pioneered by Turner [Tur04,Tur95][1] one creates non-well-founded versions of algebraic data types. An example of a (well-founded) algebraic type is the type of natural numbers, which is defined as follows:[2]

$$\begin{array}{l}\text{data } \mathbb{N} : \text{Set where}\\ \quad 0 \quad : \mathbb{N}\\ \quad \mathsf{suc} : \mathbb{N} \to \mathbb{N}\end{array}$$

The elements of $\mathbb{N}$ are obtained by finitely many applications of the constructors. One can define a function from $\mathbb{N}$ to another type by pattern matching, i.e. by making a case distinction on whether the argument is 0 or $(\mathsf{suc}\ n)$.

An example of a codata type is the set of streams of natural numbers

$$\begin{array}{l}\text{codata } \mathsf{Stream} : \text{Set where}\\ \quad \text{cons} : \mathbb{N} \to \mathsf{Stream} \to \mathsf{Stream}\end{array}$$

The keyword codata indicates that we are allowed to have infinitely many applications of cons, and therefore form infinitary terms $(\text{cons}\ n_0\ (\text{cons}\ n_1\ \cdots))$. As for data types one would expect pattern matching to work for codata types. We won't make this assumption in this article, and actually show that whether an element of a codata type matches a pattern is in general undecidable. In this article by codata types we mean types which are like data types, but we allow infinite (more generally non-wellfounded) applications of the constructor.

The need for decidable equality. Problems of the codata approach arise when one requires decidable type checking, as it holds in most typed programming languages.

Most theorem provers use a goal-directed approach to derive proofs. One states a goal and then uses inference rules to derive that goal. If one had to write programs in normal programming languages this way one would need to derive a program by, for instance in case of Java, using a rule that it consists of a class with some name and some methods. Then we could use another rule to derive how a method is defined, etc. Using such an approach for deriving programs would be more tedious and would be much more difficult to learn programming than the approach used of writing the program text and then type checking it by the compiler.

Agda is an example of a theorem prover with decidable type checking. Proving is very close to programming: instead of deriving an element of a type using rules, the user types in a program text with some help from the system, which is then type checked automatically. Certain parts of the program text can be left open

[1] The earliest occurrence of codata types we could find is [Gim95], who called it "Coinductive". Hagino uses the notion of "codatatype" in [Hag89], but that notion refers to coalgebras defined by their elimination rules.

[2] We use in this section a notation similar to that of Agda. In particular, as common in Martin-Löf type theory, Set denotes the set of small types, and we write application in functional style, i.e. $(f\ a)$ for f applied to a.

(called "goals"). The code is type checked automatically by the system and the user gets some help for filling in the goals. This allows the programmer to type in known parts directly, and allows to combine in a very flexible way both forward and backward reasoning.

Decidable type checking in dependent types implies decidability of equality. This can be most easily seen when using Leibniz equality: If $a, b : A$, we have

$$(\lambda X.\lambda x.x : \Pi_{X:A\to \mathrm{Set}}(X\ a \to X\ b)) \Leftrightarrow a \text{ and } b \text{ are equal elements of } A\ .$$

Here, $\Pi_{X:A\to Set}(X\ a \to X\ b)$ is the polymorphic type of functions mapping any predicate on A (i.e. of type $X \to \mathrm{Set}$) to an element of $(X\ a \to X\ b)$. Therefore in a type theory, which allows to define the polymorphic type of Leibniz equality, and which has decidable type checking, we can decide using the type statement on the left hand side whether a and b are equal elements of type A. Therefore decidability of type checking implies decidability of equality.

Problems of the codata approach. The natural equality on Streams is bisimilarity, which means that two streams $(\mathrm{cons}\ n_0\ (\mathrm{cons}\ n_1\ \cdots))$ and $(\mathrm{cons}\ m_0\ (\mathrm{cons}\ m_1\ \cdots))$ are equal if $n_i = m_i$ for all $i : \mathbb{N}$, that is, the functions $\lambda i.n_i$ and $\lambda i.m_i$ are extensionally equal. Since extensional equality on $\mathbb{N} \to \mathbb{N}$ is undecidable, bisimilarity is undecidable as well.

In order to deal with the problem of undecidability of extensional equality for function spaces, in Martin-Löf type theory ($MLTT$) one defines for type checking purposes two functions $f, g : A \to B$ as definitionally (or judgementally) equal, if f, g as λ-terms reduce to the same normal form.

One can say that two functions are definitionally equal, if the underlying programs are syntactically equivalent. In order to state that two functions are extensionally equal, one introduces a type (or proposition) expressing extensional equality, and then can prove extensional equality of functions in type theory.

In the same way a decidable equality on codata types can be based on the principle that two elements of a codata type are equal, if the underlying terms have the same normal form, i.e. the elements are generated by the same program up to definitional equality. Such an equality requires usually normalisation. Bisimilarity can then be introduced as a proposition which is given as a coinductive relation.

We cannot permit full expansion of codata types, since we would get infinite and therefore non-normalising terms. The solution taken in Coq and earlier versions of Agda is to impose restrictions on when an element of a codata type can be expanded (see also the approach in [ADLO10] using lifting and boxing operators). These solutions led to a problem of subject reduction in Coq and earlier versions of Agda (see [APTS13] for a discussion on the history of this problem). As a consequence, in Agda elimination rules for codata types have been initially restricted to such extent that they are difficult to use. Later the "musical approach" was taken, which will be discussed in the conclusion. The latest approach taken in Agda uses coalgebras.

Coalgebras. A solution to this problem goes back to Hagino [Hag87,Hag89], namely to use the categorical dual of initial algebras (which correspond to algebraic data types), namely coalgebras. This approach has been further developed

by Geuvers [Geu92], Howard [How96], Greiner [Gre92], Mendler [Men91]. It has been promoted for the use in MLTT by the second author in several talks and in [Set12,Set16], and by Granström [Gra08], and McBride [McB09]. See as well the work by Abbott, Altenkirch, and Ghani on containers [AAG03], and by Basold and Geuvers [BG16]. This approach has now been implemented in Agda (e.g. [AAS17,AAS16]).

Instead of defining Stream by its introduction rule, it is defined by its elimination rules

$$\begin{array}{l}\text{coalg } \mathsf{Stream} : \text{Set where}\\ \quad \text{head} : \mathsf{Stream} \to \mathbb{N}\\ \quad \text{tail} \ \ : \mathsf{Stream} \to \mathsf{Stream}\end{array}$$

The notation used in Agda is

$$\begin{array}{l}\text{record } \mathsf{Stream} : \text{Set where}\\ \quad \text{coinductive}\\ \quad \text{field}\\ \quad\quad \text{head} : \mathbb{N}\\ \quad\quad \text{tail} \ \ : \mathsf{Stream}\end{array}$$

Elements of Stream are terms such that head and tail applied to them return elements of $\mathbb{N}$ and Stream, respectively. A model of coalgebras as sets of natural numbers can for instance be found in [Set16].

The dual of primitive recursion is primitive corecursion (the earliest occurrence of this notion is probably Vene and Uustalu [VU98], see as well [Set12]). It corresponds to guarded recursion (see [Coq94]). Primitive corecursion means for the type Stream that if we have $A : \text{Set}$, $h : A \to \mathbb{N}$, $t : A \to (\mathsf{Stream} + A)$, then there exists

$$\begin{array}{l} f : A \to \mathsf{Stream}\\ \text{head } (f\ a) = h\ a\ \ ,\\ \text{tail} \quad (f\ a) = \begin{cases} s & \text{if } t\ a = \text{inl } s,\\ f\ a' & \text{if } t\ a = \text{inr } a'. \end{cases}\end{array}$$

In the codata approach this principle translates as follows: Assuming h and t as before, we can define

$$\begin{array}{l} f : A \to \mathsf{Stream}\\ f\ a = \begin{cases} \text{cons } (h\ a)\ s & \text{if } t\ a = \text{inl } s,\\ \text{cons } (h\ a)\ (f\ a') & \text{if } t\ a = \text{inr } a'. \end{cases}\end{array}$$

Essentially we can define $f\ a = \text{cons}\, n\, s$, where n and s depend on a, and s can be a stream which was defined before or $s = f\ a'$ for some $a' : A$.

Guarded recursion is widely accepted as a natural rule for coalgebras and codata types. In the POPL article [APTS13], coauthored by the second author, a simply typed recursive calculus was introduced, in which the principle of primitive corecursion is represented by copattern matching, the dual of pattern matching. There it was shown that this calculus fulfils subject reduction.

Guarded Recursion for Codata Types. The principle of copattern matching and primitive corecursion for coalgebras corresponds to the principle of guarded recursion as introduced originally for codata types by Thierry Coquand [Coq94]. If we take the codata definition of Stream, guarded recursion allows to define a function $f : A \to \mathsf{Stream}$ by defining $f\ a = s$ for a stream s (depending on a) defined before, or by defining $f\ a = \text{cons}\ s\ (f\ t)$ for some s, t which depend on a. Nesting of constructors on the right hand side are allowed, but no other functions can be used.

An equation $f\ a = \text{cons}\ s\ (f\ t)$ corresponds in the coalgebra approach to copattern matching equations

$$\begin{aligned} \text{head}\ (f\ a) &= s \\ \text{tail}\ \ (f\ a) &= f\ t \end{aligned}$$

An equation $f\ a = s$ for a stream s defined before corresponds to the copattern equations

$$\begin{aligned} \text{head}\ (f\ a) &= \text{head}\ s \\ \text{tail}\ \ (f\ a) &= \text{tail}\ s \end{aligned}$$

So guarded recursion translates directly into copattern matching and primitive corecursion, and vice versa. Nested applications of constructors in a guarded recursion equation correspond to nested copattern matching: an equation $f\ a = \text{cons}\ s\ (\text{cons}\ t\ (f\ r))$ for guarded recursion corresponds to the equations

$$\begin{aligned} \text{head}\ (f\ a) &= s \\ \text{head}\ (\text{tail}\ (f\ a)) &= t \\ \text{tail}\ \ (\text{tail}\ (f\ a)) &= f\ r \end{aligned}$$

Weakly Final Coalgebras. In final coalgebras one requires uniqueness of the function f introduced by primitive corecursion.[3] This principle is equivalent to bisimilarity as equality on coalgebras, which for streams means componentwise equality, and is therefore undecidable. We note that for final coalgebras the constructor is an isomorphism, so every element of a final coalgebra is introduced by a constructor. In order to obtain decidability of type checking, one replaces final coalgebras by weakly final coalgebras. In *weakly* final coalgebras, only the existence of functions defined by primitive corecursion is required, not their uniqueness. Elements of the coalgebra are introduced by the primitive corecursion operator $\mathrm{P}_{\mathrm{corec},A}$

$$\begin{aligned} &\mathrm{P}_{\mathrm{corec},A} : (A \to \mathbb{N}) \to (A \to (\mathsf{Stream} + A)) \to A \to \mathsf{Stream} \\ &\text{head}\ \ (\mathrm{P}_{\mathrm{corec},A}\ h\ t\ a) = h\ a \\ &\text{tail}\ \ \ (\mathrm{P}_{\mathrm{corec},A}\ h\ t\ a) = \begin{cases} s & \text{if } t\ a = \text{inl}\ s, \\ \mathrm{P}_{\mathrm{corec},A}\ h\ t\ a' & \text{if } t\ a = \text{inr}\ a'. \end{cases} \end{aligned}$$

[3] Actually it is only required for the principle of coiteration, where tail needs always to be of the form $f\ a'$. If one has uniqueness, one can derive the existence and uniqueness of functions defined by primitive corecursion. See [Set16] for a proof that for strictly positive coalgebras uniqueness of the functions defined by coiteration and by primitive corecursion are both equivalent to having a final coalgebra.

Elements of the coalgebra are equal if they reduce to the same normal form. MLTT style rules for coalgebras were worked out in [Set12]. Mendler [Men87] and Geuvers [Geu92] (Prop. 5.7) have shown that the polymorphic lambda calculus extended by weakly initial algebras and weakly final coalgebras for positive type schemes and higher type primitive recursion and primitive corecursion is strongly normalising. Therefore we obtain a decidable equality on coalgebras.

3 Undecidability of Weak Forms of Equality on Streams

We are going to show that, under minimal desirable conditions for streams, there is no decidable equality such that two streams with the same head and the same tail are equal. As usual when defining an undecidability result, we assume some encoding of computable streams as subsets of natural numbers. Any implementation of type theory would need some form of representing the terms inside the systems, which amounts to encoding them in the computer. Since a representation on the computer is binary, and binary numbers are just natural numbers, we can encode them as natural numbers. So we will work now in a standard recursion theoretic setting. As is tradition there, we will use here mathematical notation for application, i.e. we write $f(x)$ instead of $(f\ x)$.

Convention 1. *(a) By a decidable relation on $A \subseteq \mathbb{N}$ we mean a subset $B \subseteq A$ such that there is a partial recursive function f such that for all $x \in A$, $f(x)$ is defined with $f(x) \in \{0,1\}$, and $x \in B$ iff $f(x) = 1$.*

(b) When writing $f : A \to B$ where $A, B \subseteq \mathbb{N}$ we man that f is a function from $\mathbb{N}$ to $\mathbb{N}$ such that $f(x) \in B$ for all $x \in A$.

Assumption 2. *(a) We assume some standard primitive recursive pairing function $\pi : \mathbb{N}^2 \to \mathbb{N}$ together with projections $\pi_0, \pi_1 : \mathbb{N} \to \mathbb{N}$, s.t. for $x, y \in \mathbb{N}$ we have $\pi_0(\pi(x,y)) = x$, $\pi_1(\pi(x,y)) = y$, $x = \pi(\pi_0(x), \pi_1(x))$.*

(b) Let $\mathrm{inl}, \mathrm{inr} : \mathbb{N} \to \mathbb{N}$, $\mathrm{inl}(n) = 2n$, $\mathrm{inr}(n) = 2n + 1$.

(c) For $A, B \subseteq \mathbb{N}$ we set
- *$A \times_{\mathbb{N}} B := \{\pi(a,b)\} \mid a \in A, b \in B\}$.*
- *$A +_{\mathbb{N}} B := \{\mathrm{inl}(a) \mid a \in A\} \cup \{\mathrm{inr}(b) \mid b \in B\}$.*

(Note that $\mathbb{N} +_{\mathbb{N}} \mathbb{N} = \mathbb{N} \times_{\mathbb{N}} \mathbb{N} = \mathbb{N}$).

(d) We assume encodings of Turing machines (TM) and configurations for TMs as natural numbers. A configuration represent the finite portion of the blank tape currently used, the head position and the state of the TM. We assume that the working of TMs is modelled by primitive recursive functions
- *$\mathrm{init} : \mathbb{N} \to \mathbb{N}$, which computes for a TM e its initial configuration;*
- *$\mathrm{next} : \mathbb{N}^2 \to \mathbb{N}$, which computes for a TM e and configuration c the configuration obtained after executing the next step of the TM;*
- *$\mathrm{checkHalt} : \mathbb{N}^2 \to \mathbb{N}$, which for TM e and configuration c determines whether e has halted (then it returns $\mathrm{true} := 1$, otherwise it returns $\mathrm{false} := 0$);*
- *$\mathrm{result} : \mathbb{N}^2 \to \mathbb{N}$, such that $\mathrm{result}(e,c)$ returns, if TM e in configuration c has halted, the result of this TM by reading it off the tape.*

For $e \in \mathbb{N}$, $\{e\}$ denotes the partial recursive function (without input) corresponding to TM e, that is $\{e\} \simeq \mathrm{run}(e, \mathrm{init}(e))$ where

$$\mathrm{run}(e,c) \simeq \begin{cases} \mathrm{result}(e,c) & \text{if } \mathrm{checkHalt}(e,c) = \mathrm{true}, \\ \mathrm{run}(e, \mathrm{next}(e,c)) & \text{otherwise.} \end{cases}$$

Definition 3. *From* $\mathrm{init}, \mathrm{next}, \mathrm{checkHalt}, \mathrm{result}$ *we derive primitive recursive functions which operate on pairs $\pi(e,c)$ for TMs e and configurations c. We also define a bounded variant of the function* run *that models termination after a given number n of computation steps:*

- $\mathrm{init}' : \mathbb{N} \to \mathbb{N}$, $\mathrm{init}'(e) = \pi(e, \mathrm{init}(e))$.
- $\mathrm{next}' : \mathbb{N} \to \mathbb{N}$, $\mathrm{next}'(\pi(e,c)) = \pi(e, \mathrm{next}(e,c))$.
- $\mathrm{checkHalt}' : \mathbb{N} \to \mathbb{N}$, $\mathrm{checkHalt}'(\pi(e,c)) = \mathrm{checkHalt}(e,c)$.
- $\mathrm{result}' : \mathbb{N} \to \mathbb{N}$, $\mathrm{result}'(\pi(e,c)) = \mathrm{result}(e,c)$.
- $\mathrm{run}'_n(d) = \begin{cases} \mathrm{result}'(d) + 1 & \text{if } n = 0 \text{ and } \mathrm{checkHalt}'(d) = \mathrm{true}, \\ \mathrm{run}'_{n-1}(\mathrm{next}'(d)) & \text{if } n > 0 \text{ and } \mathrm{checkHalt}'(d) = \mathrm{false}, \\ 0 & \text{otherwise.} \end{cases}$

$\mathrm{run}'_n(d)$ is a primitive recursive function of n and d such that $\mathrm{run}'_n(\mathrm{init}'(e)) > 0$ if and only if the TM encoded by e halts after exactly n steps and in that case $\{e\} \simeq \mathrm{run}'_n(\mathrm{init}'(e)) - 1$.

Definition 4. *An* encoding of streams $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ *is given by:*

(a) A subset $\mathsf{Stream} \subseteq \mathbb{N}$.
(b) An equivalence relation $== \subseteq \mathsf{Stream} \times \mathsf{Stream}$*, called the* equality *of the stream encoding. We write $s == s'$ for $(s,s') \in ==$, and $s \neq= s'$ for $(s,s') \notin ==$.*
(c) Functions $\mathrm{head} : \mathsf{Stream} \to \mathbb{N}$, $\mathrm{tail} : \mathsf{Stream} \to \mathsf{Stream}$ *that respect $==$, i.e.*

$$\forall s, s' : \mathsf{Stream} \,.\, s == s' \to \mathrm{head}(s) = \mathrm{head}(s') \wedge \mathrm{tail}(s) == \mathrm{tail}(s')$$

Note that we do not impose any effectivity conditions on the set Stream or the functions head and tail.

Definition 5. *Let* $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ *be an encoding of streams. For $s, s' \in \mathsf{Stream}$ and a vector of natural numbers $\boldsymbol{n}$ we define*

$$s \xrightarrow{\boldsymbol{n}} s' \Leftrightarrow \forall i < |\boldsymbol{n}| \mathsf{head}_i(s) = n_i \wedge \mathsf{tail}^{|\boldsymbol{n}|}(s) == s'$$

where tail^k *is the k-fold iteration of* tail*, and* $\mathrm{head}_k(s) := \mathrm{head}(\mathrm{tail}^k(s))$.

Definition 6. *An encoding of streams* $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ *is* injective *if the function*

$$\langle \mathrm{head}, \mathrm{tail} \rangle : \mathsf{Stream} \to \mathbb{N} \times \mathsf{Stream}, \quad \langle \mathrm{head}, \mathrm{tail} \rangle(s) = (\mathsf{head}(s), \mathsf{tail}(s))$$

is injective w.r.t. $==$, that is

$$\forall s, s' : \mathsf{Stream} \,.\, \mathrm{head}(s) = \mathrm{head}(s') \wedge \mathrm{tail}(s) == \mathrm{tail}(s') \to s == s'$$

The following easy lemma shows that every encoding of streams can be naturally turned into two injective ones that differ from the original one only in their equality.

Lemma and Definition 7. Let $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ be an encoding of streams. Define

$$s ==_{<\omega} s' \Leftrightarrow \exists \boldsymbol{n}, t\,(s \xrightarrow{\boldsymbol{n}} t \wedge s' \xrightarrow{\boldsymbol{n}} t)$$
$$s \sim s' \Leftrightarrow \forall i \in \mathbb{N}\,(\mathsf{head}_i(s) = \mathsf{head}_i(s'))$$

Then $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==_{<\omega})$ and $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, \sim)$ are injective encodings of streams with $== \subseteq ==_{<\omega} \subseteq \sim$.

$==_{<\omega}$ can also be inductively defined as the least relation containing $==$ and making $\langle \mathrm{head}, \mathrm{tail}\rangle$ injective. $\sim$ is the usual bisimilarity of stream which can also be defined coinductively. If $==$ is an intensional notion of equality on streams, then the three equalities $==$, $==_{<\omega}$, $\sim$ are usually all different. We will give concrete examples where these equalities differ after the proof of our main result, Theorem 9.

Definition 8. *An encoding of streams* $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ *is* universal *if for any primitive recursive functions* $h : \mathbb{N} \to \mathbb{N}$ *and* $t : \mathbb{N} \to (\mathsf{Stream} +_{\mathbb{N}} \mathbb{N})$ *there exists a primitive recursive function* $g : \mathbb{N} \to \mathsf{Stream}$ *such that*

- $\mathrm{head}(g(n)) = h(n)$
- $\mathrm{tail}(g(n)) == \begin{cases} s & \text{if } t(n) = \mathrm{inl}(s), \\ g(k) & \text{if } t(n) = \mathrm{inr}(k). \end{cases}$

We say g is defined by primitive corecursion (from h and t) *if* g *is primitive recursive and satisfies the equations above.*

Every constructive type theory equipped with coalgebras (or codata) and a primitive corecursion operator gives rise to a universal encoding of streams where the function $g : \mathbb{N} \to \mathsf{Stream}$ defined by primitive corecursion is defined from h and t by a primitive corecursion operator P, that is, $g = \mathrm{P}(g, h)$.

Theorem 9. Every injective universal encoding of streams has an undecidable equality.

Proof: Let $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ be a universal encoding of streams. By universality, let $\mathrm{const} : \mathbb{N} \to \mathsf{Stream}$ be defined from the identity function and inr by primitive corecursion, that is, $\mathrm{head}(\mathrm{const}(i)) = i$ and $\mathrm{tail}(\mathrm{const}(i)) == \mathrm{const}(i)$.
Claim. Assume $s \xrightarrow{0^n} \mathrm{const}(k)$.

(a) $s == \mathrm{const}(0)$ implies $k = 0$.
(b) If $\langle \mathrm{head}, \mathrm{tail}\rangle$ is injective, then $k = 0$ implies $s == \mathrm{const}(0)$.

Proof of the Claim by induction on n: If $n = 0$, then the assumption is $s == \mathrm{const}(k)$. For (a) assume $s == \mathrm{const}(0)$. Then $\mathrm{const}(k) == \mathrm{const}(0)$ and

therefore $k = \text{head}(\text{const}(k)) = \text{head}(\text{const}(0)) = 0$. Part (b) follows trivially from the assumption.

Now assume $n > 0$. The assumption now means $\text{head}(s) = 0$ and $\text{tail}(s) \overset{0^{n-1}}{\to} \text{const}(k)$. For (a) assume $s == \text{const}(0)$. Then $\text{tail}(s) == \text{tail}(\text{const}(0)) == \text{const}(0)$. Hence $k = 0$, by induction hypothesis. For(b) assume $k = 0$. By induction hypothesis, $\text{tail}(s) == \text{const}(0)$. Hence $s == \text{const}(0)$, by injectivity and since $\text{head}(s) = 0 = \text{head}(\text{const}(0))$. This completes the proof of the Claim.

By universality, there exists a primitive recursive function $f : \mathbb{N} \to \mathsf{Stream}$ s.t. if TM e terminates with result k after n steps, that is, $\text{run}'_n(\text{init}'(e)) = k+1$, then

$$f(e) \overset{0^{n+1}}{\to} \text{const}(k)$$

We will give a detailed argument why f exists at the end of the proof.

Now assume that $\langle \text{head}, \text{tail} \rangle$ is injective. Then we have by the Claim, applied to $s = f(e)$ where e is a TM that halts with result k, that $f(e) == \text{const}(0)$ iff $k = 0$. Therefore, if $==$ were decidable, then the function $\lambda e \,.\, f(e) == \text{const}(0)$ would be recursive and it would separate the TMs which terminate with result 0 from the TMs terminating with result > 0. But there is no recursive function separating these two sets, by the following theorem (part of the proof of Theorem II.2.5 on p. 148 in Odifreddi [Odi92]; references to originators are due to Odifreddi; the result can be found as well in Gasarch 1998 [Gas98], p. 1047, Note 2.8.):

Theorem 10. (Rosser [Ros36], Kleene [Kle50], Novikov, Trakhtenbrot [Tra53]) Let $A := \{e \mid \{e\} \simeq 0\}$ and $B := \{e \mid \{e\} \simeq 1\}$. Then A and B are recursively inseparable, that is, there is no (total) recursive function $f : \mathbb{N} \to \{0, 1\}$ such that $f(0) = 0$ for all $e \in A$, and $f(e) = 1$ for all $e \in B$.

We complete the proof of Theorem 9 showing that a function f with the property specified above exists. Define primitive recursive functions $h : \mathbb{N} \to \mathbb{N}$ and $t : \mathbb{N} \to (\mathsf{Stream} +_{\mathbb{N}} \mathbb{N})$ by

$$h(d) = 0$$
$$t(d) = \begin{cases} \text{inl}(\text{const}(\text{result}'(d))) & \text{if checkHalt}'(d) = \text{true}, \\ \text{inr}(\text{next}'(d)) & \text{otherwise.} \end{cases}$$

Let g be defined by primitive corecursion from h and t. We have

$$\begin{aligned} \text{head}(g(d)) &= 0 \\ \text{tail}(g(d)) &== \begin{cases} \text{const}(\text{result}'(d)) & \text{if checkHalt}'(d) = \text{true}, \\ g(\text{next}'(d)) & \text{otherwise.} \end{cases} \end{aligned}$$

Let $f : \mathbb{N} \to \mathsf{Stream}$, $f(e) = g(\text{init}'(e))$. We show that f is as required, that is, if $\text{run}'_n(\text{init}'(e)) = k + 1$, then $f(e) \overset{0^{n+1}}{\to} \text{const}(k)$. We show more generally if $\text{run}'_n(d) = k + 1$, then $g(d) \overset{0^{n+1}}{\to} \text{const}(k)$, by induction on n.

If $n = 0$, then $\mathrm{checkHalt}'(d) = \mathrm{true}$ (since $\mathrm{run}'_n(d) > 0$). Therefore $\mathrm{run}'_n(d) = \mathrm{result}'(d) + 1$ and $k = \mathrm{result}'(d)$. It follows $\mathrm{head}(g(d)) = 0$, $\mathrm{tail}(g(d)) == \mathrm{const}(\mathrm{result}'(d)) = \mathrm{const}(k)$, and therefore $g(d) \overset{0^{n+1}}{\to} \mathrm{const}(k)$.

If $n > 0$, then $\mathrm{checkHalt}'(d) = \mathrm{false}$ (since $\mathrm{run}'_n(d) > 0$). Therefore $\mathrm{run}'_n(d) = \mathrm{run}'_{n-1}(\mathrm{next}'(d)) = k+1$. By induction hypothesis $g(\mathrm{next}'(d)) \overset{0^n}{\to} \mathrm{const}(k)$. Since $\mathrm{head}(g(d)) = 0$ and $\mathrm{tail}(g(d)) == g(\mathrm{next}'(d))$ it follows $g(d) \overset{0^{n+1}}{\to} \mathrm{const}(k)$.

Corollary 11. Assume a universal encoding of streams $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$. Assume a function $\mathrm{cons} : \mathbb{N} \times \mathsf{Stream} \to \mathsf{Stream}$ that respects $==$, that is,

$$\forall n, s, s' \,.\, s == s' \to \mathrm{cons}(n, s) == \mathrm{cons}(n, s')$$

(a) Assume

$$\forall s : \mathsf{Stream} \,.\, s == \mathrm{cons}(\mathrm{head}(s), \mathrm{tail}(s))$$

that is, cons is a left-inverse of $\langle \mathrm{head}, \mathrm{tail} \rangle$ w.r.t. $==$. Then $==$ is undecidable.

(b) Assume

$$\forall s : \mathsf{Stream} \,.\, \mathrm{head}(\mathrm{cons}(n, s)) = n \wedge \mathrm{tail}(\mathrm{cons}(n, s)) == s$$

that is, cons is a right-inverse of $\langle \mathrm{head}, \mathrm{tail} \rangle$ w.r.t. $==$. Assume further

$$\forall s : \mathsf{Stream} \,.\, \exists n . \exists s' : \mathsf{Stream} \,.\, s == \mathrm{cons}(n, s')$$

that is, cons is surjective w.r.t. $==$. Then $==$ is undecidable.

Proof of Corollary 11: (a) If $\langle \mathrm{head}, \mathrm{tail} \rangle$ has a left-inverse, it is injective, hence Theorem 9 applies. (b) A surjective right-inverse is also a left-inverse.

Corollary 12. For every universal encoding of streams the equalities $==_{<\omega}$ and $\sim$ defined in Lemma 7 are undecidable.

Examples 13. Let $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ be a universal encoding of streams that is derived from some intensional constructive type theory with primitive corecursion (like for example the theory underlying Agda) such that $==$ corresponds to definitional equality.

First, we argue that $==_{<\omega}$ is not the same as bisimilarity by constructing bisimilar streams that are not related by $==_{<\omega}$: Let $f : \mathbb{N} \to \mathsf{Stream}$ be defined by primitive corecursion such that $\mathrm{head}(f(x)) = 0$ and $\mathrm{tail}(f(x)) == f(x)$ for all $x \in \mathbb{N}$. Since $f(0)$ and $f(1)$ come from different terms in normal form we have $f(0) \neq== f(1)$. Since for all $n \in \mathbb{N}$

$$\mathrm{tail}^n(f(0)) == f(0) \neq== f(1) == \mathrm{tail}^n(f(1))$$

it follows that $f(0) \neq=_{<\omega} f(1)$. However, clearly $f(0)$ and $f(1)$ are bisimilar.

Next, we construct streams witnessing the fact that $==$ and $==_{<\omega}$ are different. From Theorem 9 we know that these two relation cannot coincide since $==$ is

decidable but $==_{<\omega}$ isn't, however, it is interesting to see the difference by an example. We simply modify the above example slightly. Let $f' : \mathbb{N} \to \mathsf{Stream}$ be defined by primitive corecursion such that $\mathrm{head}(f'(x)) = 0$ and $\mathrm{tail}(f'(x)) == f(0)$ for all $x \in \mathbb{N}$. With the same argument as before, $f'(0) \neq= f'(1)$. However $f'(0) \overset{0}{\to} f(0)$ and $f'(1) \overset{0}{\to} f(0)$, therefore $f'(0) ==_{<\omega} f'(1)$.

Remark 14. In the definitions and proofs above one may replace the class of primitive recursive functions by any other class of recursive functions satisfying some minimal closure conditions, for example all recursive functions, elementary functions, or polynomial time computable functions. Then Theorem 9 is still valid with the same proof.

4 Extension of Theorem 9 to Coiteration

For coalgebras we have the principles of primitive corecursion and coiteration which are the dual of primitive recursion and iteration for algebraic data types. A detailed discussion of these concepts and why they are dual can for instance be found in [Set16]. When we define a function $f : A \to \mathsf{Stream}$ by primitive corecursion, we have the choice of defining $\mathsf{tail}(f(a)) = f(a')$ or $\mathsf{tail}(f(a)) = s$ for some given stream s. Coiteration restricts this choice by demanding that $\mathsf{tail}(f(a))$ always needs to be equal to $f(a')$ for some a'. An encoding of streams is coiteratively universal if it is closed under the coiteration operator:

Definition 15. *An encoding of streams* $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ *is* coiteratively universal *if for any primitive recursive functions* $h : \mathbb{N} \to \mathbb{N}$ *and* $t : \mathbb{N} \to \mathbb{N}$ *there exists a primitive recursive function* $g : \mathbb{N} \to \mathsf{Stream}$ *such that*

- $\mathrm{head}(g(n)) = h(n)$
- $\mathrm{tail}(g(n)) == g(t(n))$.

We say g is defined by coiteration (from h and t)*, if* g *is primitive recursive and satisfies the equations above.*

Note that the functions f and f' in Example 13 are in fact defined by coiteration. However, our main theorem 9 above relied essentially on the fact that we have primitive corecursion. This allowed us to escape once the TM has terminated into the streams $\mathrm{const}(i)$, and it is important that it was the same stream and not only a stream bisimilar to $\mathrm{const}(i)$. We will show that the main theorem applies as well to coiteratively universal encodings of streams, and that we can overcome the problem of not being able to escape into $\mathrm{const}(i)$ directly. But let us first repeat the standard argument that coiteration can simulate primitive corecursion up to bisimilarity:

Lemma 16. Let $(\mathsf{Stream}, \mathrm{head}, \mathrm{tail}, ==)$ be a coiteratively universal encoding of streams. Assume head and tail are primitive recursive (which are therefore defined on $\mathbb{N}$). Let h, t as in the definition of "universal encoding of streams", that is, $h : \mathbb{N} \to \mathbb{N}$ and $t : \mathbb{N} \to (\mathsf{Stream} +_{\mathbb{N}} \mathbb{N})$. Then there exist a primitive recursive function $g : \mathbb{N} \to \mathsf{Stream}$ such that g behaves up to $\sim$ like a function defined by primitive corecursion from h, t, more precisely,

(a) $\mathrm{head}(g(n)) = h(n)$,
(b) if $t(n) = \mathrm{inl}(s)$, then $\mathrm{tail}(g(n)) \sim s$,
(c) if $t(n) = \mathrm{inr}(m)$, then $\mathrm{tail}(g(n)) == g(m)$.

Proof: Define

$$\begin{aligned} &h' : \mathbb{N} \to \mathbb{N} \quad \text{(recall that } \mathbb{N} = \mathbb{N} +_{\mathbb{N}} \mathbb{N}) \\ &h'(\mathrm{inl}(n)) = \mathrm{head}(n) \\ &h'(\mathrm{inr}(n)) = h(n) \\ \\ &t' : \mathbb{N} \to \mathbb{N} \\ &t'(\mathrm{inl}(n)) = \mathrm{inl}(\mathrm{tail}(n)) \\ &t'(\mathrm{inr}(n)) = t(n) \end{aligned}$$

Let g' be defined by coiteration from h' and t', that is, for all $n \in \mathbb{N}$

$\mathrm{head}(g'(n)) = h'(n)$,
$\mathrm{tail}(g'(n)) == g'(t'(n))$.

Let $g(n) := g'(\mathrm{inr}(n))$. Then g is primitive recursive and satisfies the conditions (a), (b), (c) as we show now. Conditions (a) and (c) are easy:

$$\mathrm{head}(g(n)) = \mathrm{head}(g'(\mathrm{inr}(n)) = h'(\mathrm{inr}(n)) = h(n),$$

and if $t(n) = \mathrm{inr}(m)$, then

$$\mathsf{tail}(g(n)) == \mathsf{tail}(g'(\mathrm{inr}(n))) == g'(t'(\mathrm{inr}(n'))) == g'(t(n)) = g(m).$$

For condition (b) we show first that $g'(\mathrm{inl}(s)) \sim s$ for all $s \in \mathsf{Stream}$. In fact, for all $n \in \mathbb{N}$ $\mathrm{tail}^n(g'(\mathrm{inl}(s))) == g'(t'^n(\mathrm{inl}(s))) = g'(\mathrm{inl}(\mathrm{tail}^n(s)))$ and therefore

$$\begin{aligned} \mathrm{head}(\mathrm{tail}^n(g'(\mathrm{inl}(s)))) &= \mathrm{head}(g'(\mathrm{inl}(\mathrm{tail}^n(s)))) \\ &= h'(\mathrm{inl}(\mathrm{tail}^n(s))) \\ &= \mathrm{head}(\mathrm{tail}^n(s)). \end{aligned}$$

Now, if $t(n) = \mathrm{inl}(s)$, then

$$\mathsf{tail}(g(n)) == \mathsf{tail}(g'(\mathrm{inr}(n))) == g'(t'(\mathrm{inr}(n))) == g'(t(n)) = g'(\mathrm{inl}(s)) \sim s.$$

Theorem 17. Every injective coiteratively universal encoding of streams has an undecidable equality.

Proof: First note that although Lemma 16 reduces primitive corecursion to coiteration it cannot be used to reduce Theorem 17 to Theorem 9 since the reduction (b) in Lemma 16 is only with respect to bisimilarity. Therefore, we need a new proof, which however can be obtained by suitably modifying the proof of Theorem 9.

We replace the function g used in Theorem 9 by a function g': on arguments $\mathrm{inl}(n)$ it behaves like the function g before, and on arguments $\mathrm{inr}(n)$ it behaves

like the constant stream with elements in n. Now we can replace escaping into $\mathrm{const}(k)$ by a recursive call to $g'(\mathrm{inr}(n))$: More precisely, we define by coiteration

$$\begin{array}{ll} g'' : \mathbb{N} \to \mathsf{Stream} & \\ \mathsf{head}(g''(\mathrm{inl}(d)) & = 0 \\ \mathsf{tail}(g''(\mathrm{inl}(d)) & = \begin{cases} g''(\mathrm{inr}(\mathrm{result}'(d))) & \text{if } \mathrm{checkHalt}'(d) = \mathrm{true}, \\ g''(\mathrm{inl}(\mathrm{next}'(d))) & \text{otherwise.} \end{cases} \\ \mathsf{head}(g''(\mathrm{inr}(k)) & = k \\ \mathsf{tail}(g''(\mathrm{inr}(k)) & = g''(\mathrm{inr}(k)) \end{array}$$

We define now

$$\begin{array}{ll} g' : \mathbb{N} \to \mathsf{Stream} & \\ g'(k) & = g''(\mathrm{inl}(k)) \end{array}$$

$$\begin{array}{l} \mathrm{const}' : \mathbb{N} \to \mathsf{Stream} \\ \mathrm{const}'(k) = g''(\mathrm{inr}(k)) \end{array}$$

We obtain

$$\begin{array}{ll} \mathsf{head}(g'(d)) & = 0 \\ \mathsf{tail}(g'(d)) & = \begin{cases} \mathrm{const}'(\mathrm{result}'(d)) & \text{if } \mathrm{checkHalt}'(d) = \mathrm{true}, \\ g'(\mathrm{next}'(d)) & \text{otherwise.} \end{cases} \\ \mathsf{head}(\mathrm{const}'(k)) & = k \\ \mathsf{tail}(\mathrm{const}'(k)) & = \mathrm{const}'(k) \end{array}$$

Now by replacing const by const′ and g by g' in the proof of Theorem 9, and using the equations above, we obtain a proof of Theorem 17. ‘

Corollary 18. Corollaries 11 and 12 hold for iteratively universal encodings of streams as well.

5 Conclusion and Related Work

Codata types and Coalgebras in programming and theorem proving. This papers shows that codata types are problematic in dependent type theory if one requires decidability of type checking. Codata types can still be used in a simply typed settings in functional programming since type checking there does not require checking of equalities . They can be used as well in systems such as Nuprl where type statements are derived by the user and therefore decidability of type checking is not required. Otherwise, the best approach known at the moment is to define coalgebraic types as defined by their elimination rules.

Programming with coalgebras is very natural in a situation where a corresponding codata type would only have one constructor. The main example is the type of streams defined by having observations head and tail as defined before. An example demonstrating that copattern matching is very natural is the function enum : $\mathbb{N} \to \mathsf{Stream}$, where (enum n) enumerates the natural numbers

starting with n. It can be defined by the copattern equations head (enum n) $= n$ and tail (enum n) $=$ enum $(n+1)$.

When we define a coalgebra where the corresponding codata type has more than one constructor we face the problem that several constructors in a codata type correspond to a disjoint union whereas several observations in a coalgebra correspond to a product. For instance, the observations head and tail of Stream can be replaced by one observation elim : Stream $\to \mathbb{N} \times$ Stream. Several observations in a coalgebra therefore do not allow to simulate several constructors of a codata type directly. Consider the example of colists (i.e. potentially infinite lists) which are defined as codata as

$$\begin{aligned}&\text{codata coList} : \text{Set where}\\ &\quad \text{nil} \quad : \text{coList}\\ &\quad \text{cons} : \mathbb{N} \to \text{coList} \to \text{coList}\end{aligned}$$

The eliminator for a corresponding coalgebra needs to determine for a colist whether it is nil or (cons n s). It can be done by defining

$$\begin{aligned}&\text{coalg coList} : \text{Set where}\\ &\quad \text{elim} : \text{coList} \to \top + \mathbb{N} \times \text{coList}\end{aligned}$$

Here $\top$ is the one element type with element tt, $+$ the disjoint union. elim $l =$ inl tt means that l is of the form nil, and elim $l =$ inr $(n\ ,\ l')$ means that l is of the form (cons n l').

For programming it is more convenient to replace $\top + \mathbb{N} \times$ Stream by an extra type. A good notation is to replace the name coList by ∞coList and use coList for the extra type. We obtain the simultaneous definition of two types coList and ∞coList (using notations inspired by the "musical approach" in Agda see below):

$$\begin{aligned}&\text{coalg } \infty\text{coList} : \text{Set where}\\ &\quad \flat : \infty\text{coList} \to \text{coList}\\ &\\ &\text{data coList} : \text{Set where}\\ &\quad \text{cons} : \mathbb{N} \to \infty\text{coList} \to \text{coList}\\ &\quad \text{nil} \quad : \text{coList}\end{aligned}$$

Every element of coList is of the form (cons n s) or nil, and one can make case distinction on elements of coList. But one cannot pattern match on ∞coList and therefore not pattern match on the second argument of cons – in order to unfold it further one needs to apply $\flat$ to it.

Decidability of equalities for coalgebras mentioned at the end of Sect. 2 holds in this situation as well. Our proof regarding undecidability of equality (Theorem 9) wouldn't go through in this situation, since it required that if we unfold l, l' : coList finitely many times and get the same heads and tail, then $l == l'$. For example, the case of unfolding the elements l and l' twice doesn't mean that they are both equal to (cons n (cons n' s)). It only means that $l =$ cons n l_0, where $\flat$ $l_0 =$ cons n' s, and $l' =$ cons n l'_0, where $\flat$ $l'_0 =$ cons n' s. But these equations do not imply $l_0 == l'_0$ and therefore neither $l == l'$.

Using the "musical approach" in Agda to simulate codata types by coalgebras (and related work). In Agda there exists apart from the coalgebra approach as well an approach which can be considered as introducing syntactic sugar for the above way of simulating codata types by coalgebras, see [Agd11,Dan09]. In that approach Agda generates for every name A for a type automatically a builtin type $(\infty\, A)$, which is a type defined simultaneously with A.[4] Note that we *should not* have $\infty : \mathrm{Set} \to \mathrm{Set}$.[5] The type $(\infty\, A)$ can be considered as a coalgebra defined simultaneously with A by

$$\begin{array}{l}\mathrm{coalg}\ (\infty\, A) : \mathrm{Set}\ \mathrm{where}\\ \quad \flat : \infty\, A \to A\end{array}$$

Agda provides as well a builtin function $\sharp$ which is defined by copattern matching as

$$\begin{array}{l}\sharp : A \to \infty\, A\\ \flat\ (\sharp\ a) = a\end{array}$$

With this approach we can replace ∞coList by $(\infty$ coList), omit its definition (since it is builtin) and get a definition which is close to that of a codata type:

$$\begin{array}{l}\mathrm{data\ coList} : \mathrm{Set}\ \mathrm{where}\\ \quad \mathrm{cons} : \mathbb{N} \to \infty\ \mathrm{coList} \to \mathrm{coList}\\ \quad \mathrm{nil}\ \ \ : \mathrm{coList}\end{array}$$

We can now define $\mathrm{enum} : \mathbb{N} \to \infty$ coList by copattern matching in a way which is very close to the definition for codata types:[6]

$$\flat\ (\mathrm{enum}\ n) = \mathrm{cons}\ n\ (\mathrm{enum}\ (n+1))$$

In [Agd11,Dan09], the type $(\infty\, A)$ is considered as the type of delayed combinations, and $\sharp : A \to \infty\, A$ allows to form a delayed computation from an element of A. One could say that $\flat$ allows to trigger a delayed computation.

This approach works well in situations where one needs to simulate pure codata types, which occur quite often. However, especially the work of the second author with Bashar Igried on CSP-Agda [IS18,IS17,IS16] has shown that it can be useful to have coalgebras with several observations. Even in a situation where one has a type which has a main observation similar to $\flat$ above, one often needs additional observations (in CSP-Agda there was the need to add an additional string component to the type of processes).

[4] There are various options of how to deal with types depending on parameters – this is left as future work.

[5] Actually, a constant of this type exists in Agda – the reason is that the musical approach is introduced via a library rather than a direct syntactic extension of Agda.

[6] That's how we believe Agda should behave. In fact, in Agda one defines instead $\mathrm{enum} : \mathbb{N} \to \mathrm{coList}$ by $\mathrm{enum}\ n = \mathrm{cons}\ n\ (\sharp\ (\mathrm{enum}\ (n+1)))$, an equation which, considered verbally, is not normalising and brings back the problems avoided by the coalgebra approach.

The musical approach in Agda is as well a way of interpreting the approach by Altenkirch et.al. [ADLO10] who introduce the language $\Sigma\Pi$, which has the type $[A]$ of delayed computations, which require ! (similar to $\flat$) in order to unfold them further. Note that the language $\Pi\Sigma$ is, as stated in [ADLO10], designed as a partial language which allows general recursion.

Further Related Work. Conor McBride has stated a similar result in his articles [McB09]. However, the result is not stated as a theorem, and we were not able to work out from his ideas a complete proof.

Conclusion. We have reviewed the two approaches for introducing non-well-founded data types, namely codata types given by introduction rules, and coalgebras given by elimination rules. We have shown that under weak assumptions, which are very natural for both approaches, there exists no decidable equality on Stream such that every element of Stream is introduced by a constructor. This causes at least conceptual problems for the codata approach. The theory of coalgebras seems to be much simpler, avoids this problem and seems to be conceptually superior to the approach to codata types. Reduction rules are easier in coalgebras since there are no special restrictions on when to apply reductions. Elements of coalgebras are finite objects, which unfold to infinite objects only when applying destructors to them iteratively.

Overall, our results suggest that the future of codata types in dependent type theory with decidable type checking lies in its role as a useful derived concept based on coalgebras defined by observations. The musical notation in Agda can be seen as a realisation of this idea which makes it easy to work with the very commonly occurring situation of coalgebras which originate from codata types.

Acknowledgements. We would like to thank the three anonymous referees for many useful suggestions. These suggestions led us to a more thorough discussion of the codata vs. coalgebra question regarding its origin, current literature and future development.

This research was supported by the projects CORCON (Correctness by Construction, FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638), COMPUTAL (Computable Analysis, FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962), CID (Computing with Infinite Data, Marie Curie RISE project, H2020-MSCA-RISE-2016-731143), and by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES).

References

[AAG03] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003. `https://link.springer.com/chapter/10.1007/3-540-36576-1_2`.

[AAS16] Andreas Abel, Stephan Adelsberger, and Anton Setzer. ooAgda. Agda Library. Available from `https://github.com/agda/ooAgda`, 2016.

[AAS17] Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming*, 27, Jan 2017. https://doi.org/10.1017/S0956796816000319.

[ADLO10] Thorsten Altenkirch, Nils Danielsson, Andres Löh, and Nicolas Oury. $\Pi\Sigma$: Dependent types without the sugar. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, volume 6009, pages 40–55. LNCS, 2010. https://doi.org/10.1007/978-3-642-12251-4_5.

[Agd11] Agda Wiki. Coinductive data types, 1 January 2011. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Codatatypes.

[Agd14] Agda team. The Agda Wiki, 2014. Available from http://wiki.portal.chalmers.se/agda/pmwiki.php.

[Alt04] Thorsten 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.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *Proceedings of POPL '13*, pages 27–38, New York, NY, USA, 2013. ACM. https://doi.org/10.1145/2429069.2429075.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 3-540-20854-2.

[Ber06] Yves Bertot. CoInduction in Coq. Available from http://arxiv.org/abs/cs/0603119, March 2006.

[BG16] Henning Basold and Herman Geuvers. Type theory based on dependent inductive and coinductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 327–336, New York, NY, USA, 2016. ACM. https://doi.org/10.1145/2933575.2934514.

[CF92] Robin Cockett and Tom Fukushima. About charity. Technical report, Department of Computer Science, The University of Calgary, June 1992. Yellow Series Report No. 92/480/18. Available from ftp://ftp.cpsc.ucalgary.ca/pub/projects/charity/literature/papers_and_reports/about_charity.ps.gz.

[Coq94] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806, pages 62–78. LNCS, 1994. https://doi.org/10.1007/3-540-58085-9_72.

[DA10] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction*, pages 100–118, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007%2F978-3-642-13321-3_8?LI=true.

[Dan09] Nils Anders Danielsson. Changes to coinduction, 17 March 2009. Message posted on gmane.comp.lang.agda, available from http://article.gmane.org/gmane.comp.lang.agda/763/.

[Gas98] W. Gasarch. Chapter 16: A survey of recursive combinatorics. In Yu. L. Ershov, S.S. Goncharov, A. Nerode, J.B. Remmel, and V.W. Marek, editors,

Handbook of Recursive Mathematics - Volume 2: Recursive Algebra, Analysis and Combinatorics, volume 139, pages 1041 – 1176. Elsevier, 1998. https://doi.org/10.1016/S0049-237X(98)80049-9.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183 – 207, 1992. Available from http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

[Gim95] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-60579-7_3.

[Gim96] Carlos Eduardo 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. Available from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.9849\&rank=1.

[Gra08] J.G. Granström. *Reference and Computation in Intuitionistic Type Theory*. PhD thesis, Department of Mathematics, Uppsala University, Sweden, 2008. Available from http://intuitionistic.files.wordpress.com/2010/07/theses_published_uppsala.pdf.

[Gre92] John Greiner. Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, ADA249562, Dept. of Computer Science, Carnegie-Mellon University Pittburgh, Pittburgh, PA, USA, January 27 1992. 37 pages. http://www.dtic.mil/docs/citations/ADA249562.

[Hag87] Tatsuya 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.

[Hag89] Tatsuya Hagino. Codatatypes in ML. *J. Symb. Comput.*, 8(6):629–650, 1989. https://doi.org/10.1016/S0747-7171(89)80065-3.

[How96] Brian T. Howard. Inductive, coinductive, and pointed types. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 102–109, New York, NY, USA, 1996. ACM. https://doi.org/10.1145/232627.232640.

[HS00] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *CSL 2000*, LNCS, Vol. 1862, pages 317 – 331, 2000. https://link.springer.com/chapter/10.1007/3-540-44622-2_21.

[HS04] Peter Hancock and Anton 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 from http://drops.dagstuhl.de/opus/.

[HS05] Peter Hancock and Anton Setzer. Interactive programs 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. https://doi.org/10.1093/acprof:oso/9780198566519.003.0007.

[INR17] INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.7.1 edition, 2017. https://coq.inria.fr/refman/.

[IS16] Bashar Igried and Anton Setzer. Programming with monadic CSP-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 28–38, New York, NY, USA, 2016. ACM. https://doi.org/10.1145/2976022.2976032.

[IS17] Bashar Igried and Anton Setzer. Trace and stable failures semantics for CSP-Agda. In Ekaterina Komendantskaya and John Power, editors, *Proceedings of the First Workshop on Coalgebra, Horn Clause Logic Programming and Types, Edinburgh, UK, 28-29 November 2016*, volume 258 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–51. Open Publishing Association, 2017. https://doi.org/10.4204/EPTCS.258.3.

[IS18] Bashar Igried and Anton Setzer. Defining trace semantics for CSP-Agda, 30 Jan 2018. Accepted for publication in Postproceedings TYPES 2016, 23 pp, avilable from http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997. Available from http://www.cs.ru.nl/B.Jacobs/PAPERS/JR.pdf.

[Kle50] S C Kleene. A symmetric form of Gödel's theorem. *Ind. Math.*, 12:244 – 246, 1950. https://doi.org/10.2307/2266709.

[McB09] Conor McBride. Let's see how things unfold: Reconciling the infinite with the intensional. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Proceedings of the 3rd international Conference on Algebra and Coalgebra in Computer Science, CALCO'09*, pages 113 – 126. LNCS, 2009. https://doi.org/10.1007/978-3-642-03741-2_9.

[Men87] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In David Gries, editor, *Proceedings of the Second Annual IEEE Symp. on Logic in Computer Science, LICS 1987*, pages 30–36. IEEE Computer Society Press, June 1987.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1 – 2):159 – 172, 1991. https://doi.org/10.1016/0168-0072(91)90069-X.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2007. http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[Odi92] Piergiorgio Odifreddi. *Classical Recursion Theory*, volume 125. Elsevier, 1992. https://doi.org/10.1016/S0049-237X(08)70011-9.

[Our08] Nicolas 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.

[Ros36] Barkley Rosser. Extensions of some theorems of Gödel and Church. *The Journal of Symbolic Logic*, 1(3):pp. 87–91, 1936. http://www.jstor.org/stable/2269028.

[Rut00] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3 – 80, 2000. https://doi.org/10.1016/S0304-3975(00)00056-6.

[Set12] Anton Setzer. Coalgebras as types determined by their elimination rules. In P. Dybjer, Sten Lindström, Erik Palmgren, and G. Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 351–369. Springer, 2012. https://doi.org/10.1007/978-94-007-4435-6_16.

[Set16] Anton Setzer. How to reason coinductively informally. In Reinhard Kahle, Thomas Strahm, and Thomas Studer, editors, *Advances in Proof Theory*, pages 377–408. Springer, 2016. https://doi.org/10.1007/978-3-319-29198-7_12.

[Tra53] B.A. Trakhtenbrot. On recursive separability. *Dokl. Acad. Nauk*, 88:953 – 956, 1953.

[Tur95] D. A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Funtional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-60675-0_35.

[Tur04] D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004. http://www.jucs.org/jucs_10_7/total_functional_programming.

[VU98] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. http://cs.ioc.ee/~tarmo/papers/nwpt97-peas.pdf.