Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 265 (2010) 245–258
www.elsevier.com/locate/entcs

Bisimulations Generated from Corecursive Equations

Venanzio Capretta[1,2]

School of Computer Science
University of Nottingham
Nottingham, UK

Abstract

We consider the problem of determining the unicity of solutions for corecursive equations. This can be done by transforming the equations into a guarded form, that is, representing them as a coalgebra. However, in some examples this transformation is very hard to achieve. On the other hand, mere unicity of solutions can be determined independently by constructing a bisimulation relation. The relation is defined inductively by successive steps of reduction of the body of the equation and abstraction of the recursive calls. The algorithm is not complete: it may terminate successfully, in which case unicity is proved; it may terminate with a negative answer, in which case no bisimulation could be constructed; or it may run forever. If it diverges, the inductively defined relation is in fact a bisimulation and unicity obtains. However, we cannot decide whether the algorithm will run forever or not.

Keywords: corecursion, bisimulation, streams, coalgebra.

1 Introduction

We investigate the solutions of *corecursive equations* in functional programming languages and type theories. Coinductive types were introduced in functional programming by Hagino [10]. Their categorical properties were first clarified by Aczel and Mendler [2]. Functions to coinductive types are defined by exploiting their universal property as final coalgebras. This has been encoded syntactically as a principle of definition by *guarded recursion* [6].

Type-theoretic realizations of these ideas were implemented starting from Nuprl [14] and then in Coq [9] and Agda [16,5,3].

[1] Thanks to Hans Zantema for challenging me with some engaging corecursive equations and to Thorsten Altenkirch, Nils Anders Danielsson and Graham Hutton for stimulating discussions. Some typos and minor errors occurring in the published version of the article have been corrected; I thank Florent Balestrieri for signaling them.

[2] Email: vxc@cs.nott.ac.uk

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2010.08.015

Reasoning on coinductive types exploits the coinduction principle, stating that bisimilarity implies equality. The notion of bisimulation was first introduced by Park [17] and Milner [15] as a way of reasoning about processes. Aczel [1] adopted it as the appropriate notion of equality for non-well-founded sets. In type theory it is realized by a coinductive relation (see Chapter 13 of [4]).

As productive functions toward coinductive types become more common in (dependent) functional programming [8,11], the question arises of how to deal with equations that do not satisfy the guardedness condition.

Two recent advancements in this direction use, respectively, a notion of pebble-flow network [7] and term rewriting [22]. This paper tackles some cases of equations that are not treated by known methods. We know of no uniform way to prove existence of a solution for them. However, the coinduction principle provides a way to prove unicity of solutions by exploiting *ad hoc* bisimulations.

Section 2 introduces the basic terminology about streams and bisimulations on them, the notion of guardedness and some basic operations on streams. Section 3 illustrates the topic by studying two examples of non-guarded equations and proving the unicity of solutions by defining bisimulation relations for both. Then, in Section 4 the general algorithm to inductively define such bisimulations is given and a proof of its correctness is sketched. Finally, in Section 5 we outline the questions that are still open and lay down a plan for future research.

2 Streams and Bisimulations

By *coinductive data types* we mean sets of structured objects with a non-well-founded structure. To be more precise, such a type is, in categorical terminology, the *final coalgebra* of a functor. We will not work in full generality, but limit our discourse to the category of sets and concentrate on the type of *streams* over a given type of elements D. Streams are infinite sequences of elements.

Definition 2.1 Given a type D, the type $\mathbb{S}_D$ of *streams* over D is the set of infinite sequences of elements of D. We usually drop the subscript D if the element type is understood. The basic operations on streams are $\mathsf{head} : \mathbb{S} \to D$ that returns the first element of the sequence, $\mathsf{tail} : \mathbb{S} \to \mathbb{S}$ that returns the sequence of elements after the first, and $\mathsf{cons} : D \to \mathbb{S} \to \mathbb{S}$ that prepends a new head to a given stream. We use the notation ${}^{\mathfrak{h}}s$ for $(\mathsf{head}\, s)$, ${}^{\mathfrak{t}}s$ for $(\mathsf{tail}\, s)$, and $d :: s$ for $(\mathsf{cons}\, d\, s)$.

In categorical terms, $\langle \mathsf{head}, \mathsf{tail} \rangle : \mathbb{S} \to D \times \mathbb{S}$ is the final coalgebra for the functor $FX = D \times X$ and cons is its inverse.

We use the notation ${}^{n\mathfrak{t}}s$ for successive applications of tail and ${}^{\mathfrak{h}n}s$ for the nth element of s: ${}^{0\mathfrak{t}}s = s$, ${}^{(n+1)\mathfrak{t}}s = {}^{\mathfrak{t}}({}^{n\mathfrak{t}}s)$, ${}^{\mathfrak{h}n}s = {}^{\mathfrak{h}}({}^{n\mathfrak{t}}s)$.

Determining that two streams are equal requires verifying that all the elements in corresponding positions are the same. A powerful logical characterization of equality of streams is bisimilarity: Two streams are bisimilar if they have the same head and their tails are also bisimilar.

Definition 2.2 A *bisimulation* on $\mathbb{S}$ is a binary relation $\sim$ on streams such that

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

Theorem 2.3 (Coinduction Principle) *Let $\sim$ be a bisimulation; for every $s_1, s_2 : \mathbb{S}$, if $s_1 \sim s_2$, then $s_1 = s_2$.*

The notion of bisimulation and the coinduction principle can be generalized and applied to final coalgebras in any category. However, there are subtle differences between several notions of bisimulation that are not equivalent in full generality: recent work by Staton [21] investigated their correlations.

In Type Theory, final coalgebras are implemented by coinductive types. The syntactic notion of *guardedness* is used to determine whether recursive equations correctly define coinductive objects. An equation is called *guarded* if the recursive calls occur only as direct arguments of top-level constructors. In the case of streams, a subterm x of a term e is guarded if it recursively satisfies these requirements: x is guarded in $a :: x$; if x is guarded in e then x is guarded in $a :: e$; if x is guarded in $e_1, \ldots, e_n$ then x is guarded in $(\mathsf{case}\ t\ \mathsf{of}\ c_1 \mapsto e_1 \mid \cdots \mid c_n \mapsto e_n)$. There is some allowance for the application of functions that *filter* constructors: for example, if x is guarded in e then it is guarded in $(\mathsf{map}\, f\, e)$ because the definition of the map function guarantees that it will generate a $(::)$ in for each $(::)$ of its input. The full definition of the guardedness condition can be found in its Coq implementation [9]. For example, the following definition of the *interleave* function (the $(::)$ operator has the lowest precedence):

$$(\ltimes) : \mathbb{S} \to \mathbb{S} \to \mathbb{S}$$

$$s_1 \ltimes s_2 = {}^{\mathfrak{h}}s_1 :: s_2 \ltimes {}^{\mathfrak{t}}s_1$$

is accepted because the recursive call to $\ltimes$ on the right-hand side occurs immediately under the application of the constructor $(::)$.

On the other hand, definitions like these:

$$\mathsf{bad} : \mathbb{N} \to \mathbb{S} \qquad \mathsf{wrong} : \mathbb{S} \to \mathbb{S}$$

$$\mathsf{bad}\, n = \mathsf{bad}\, (n+1) \qquad \mathsf{wrong}\, s = {}^{\mathfrak{h}}s :: {}^{\mathfrak{t}}(\mathsf{wrong}\, s)$$

are incorrect: the first because the recursive call occurs at top level and not under a constructor; the second because, even if the recursive call is under a constructor, it does not occur directly as an argument, but it has an application of the tail function on top of it.

Mutually recursive equations defining several functions are accepted if each satisfies the guardedness condition with respect to itself and the others. The following mutual recursion is not accepted as it is presented:

$$\mathsf{even} : \mathbb{S} \to \mathbb{S} \qquad \mathsf{odd} : \mathbb{S} \to \mathbb{S}$$

$$\mathsf{even}\, s = {}^{\mathfrak{h}}s :: \mathsf{odd}\, {}^{\mathfrak{t}}s \qquad \mathsf{odd}\, s = \mathsf{even}\, {}^{\mathfrak{t}}s$$

because the occurrence of even in the definition of odd is not guarded by a constructor. However, the equation can be easily modified to satisfy the condition:

$$\mathsf{odd}\, s = {}^{\mathfrak{h}1}s :: \mathsf{odd}\, {}^{2\mathfrak{t}}s.$$

It is not always so easy to put a definition into guarded form; and yet, it may still be *productive*, that is, it may guarantee that its unfolding will always produce new elements of the result stream. The main topic of this article is in fact how to derive logical principles for those equations for which a transformation into guarded form is too difficult.

Guardedness guarantees that an equation has a unique solution. In fact, finality of $\langle \mathsf{head}, \mathsf{tail} \rangle$ means that for every coalgebra $\gamma : C \to D \times C$ there exists a unique function $f : C \to \mathbb{S}$ such that $f\, c = \pi_1\, (\gamma\, c) :: f\, (\pi_2\, (\gamma\, c))$. It is a simple task to associate a coalgebra to any guarded equation or system of equations, so that existence and unicity of solutions is ensured. In some cases, we may be able to prove the unicity of solutions independently of their existence for non-guarded equations. This is the topic of this article.

3 Corecursive Equations

In this section we look at two examples of corecursive equations on streams that are not guarded and not easily reducible to guarded form. However, there is a relatively simple technique to prove that, if they have any solution, that solution must be unique. The method works by assuming that there are two solutions to the equation, constructing an appropriate bisimulation relation, and showing that the two solutions must be related by it. We can then conclude, by the coinduction principle, that the two solutions must be equal. The technique is closely related to that developed by Rosu and Lucanu [18] and implemented in the prover CIRC.

The first example was suggested to me by Hans Zantema.

Example 3.1

$$\phi : \mathbb{S} \to \mathbb{S}$$

$$\phi\ s = {}^{\mathfrak{h}}s :: (\phi(\mathsf{even}\, {}^{\mathfrak{t}}s)) \ltimes (\phi(\mathsf{odd}\, {}^{\mathfrak{t}}s))$$

It is easy to check that the identity function is a solution to the equation in Example 3.1. But how do we prove that it is the only one? The equation is not guarded by constructors: The recursive calls to ϕ occur as arguments of $\ltimes$, not directly under a top-level constructor.

Proposition 3.2 *The equation in Example 3.1 has at most one solution.*

Proof. Suppose ϕ_1 and ϕ_2 are two solutions to the equation. We want to prove that they are extensionally equal. Towards that goal, we define an inductive relation on $\mathbb{S}$, we prove that it is a bisimulation and we show that equality of the two functions follows by the coinduction principle.

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

$$\frac{s : \mathbb{S}}{\phi_1\, s \sim \phi_2\, s}(R0) \qquad \frac{x_1, x_2, y_1, y_2 : \mathbb{S} \quad x_1 \sim x_2 \quad y_1 \sim y_2}{x_1 \ltimes y_1 \sim x_2 \ltimes y_2}(R1).$$

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

- If the last rule used was R0, then it must be $s_1 = \phi_1\, s$ and $s_2 = \phi_2\, s$ for some s. Then, because ϕ_1 and ϕ_2 satisfy the equation in Example 3.1, we have

$$s_1 = \phi_1\, s = {}^{\mathfrak{h}}s :: (\phi_1(\mathsf{even}\, {}^{\mathfrak{t}}s)) \ltimes (\phi_1(\mathsf{odd}\, {}^{\mathfrak{t}}s)),$$
$$s_2 = \phi_2\, s = {}^{\mathfrak{h}}s :: (\phi_2(\mathsf{even}\, {}^{\mathfrak{t}}s)) \ltimes (\phi_2(\mathsf{odd}\, {}^{\mathfrak{t}}s)).$$

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

$${}^{\mathfrak{t}}s_1 = (\phi_1\, (\mathsf{even}\, {}^{\mathfrak{t}}s)) \ltimes (\phi_1(\mathsf{odd}\, {}^{\mathfrak{t}}s)), \quad {}^{\mathfrak{t}}s_2 = (\phi_2(\mathsf{even}\, {}^{\mathfrak{t}}s)) \ltimes (\phi_2(\mathsf{odd}\, {}^{\mathfrak{t}}s)).$$

  By R0 we have that

$$\phi_1(\mathsf{even}\, {}^{\mathfrak{t}}s) \sim \phi_2(\mathsf{even}\, {}^{\mathfrak{t}}s) \quad \text{and} \quad \phi_1(\mathsf{odd}\, {}^{\mathfrak{t}}s) \sim \phi_2(\mathsf{odd}\, {}^{\mathfrak{t}}s);$$

  therefore ${}^{\mathfrak{t}}s_1 \sim {}^{\mathfrak{t}}s_2$ by R1 (with instantiations $x_1 := \phi_1(\mathsf{even}\, {}^{\mathfrak{t}}s)$, $x_2 := \phi_2(\mathsf{even}\, {}^{\mathfrak{t}}s)$, $y_1 := \phi_1(\mathsf{odd}\, {}^{\mathfrak{t}}s)$, $y_2 := \phi_2(\mathsf{odd}\, {}^{\mathfrak{t}}s)$).
- If the last rule used was R1, then there must be streams x_1, x_2, y_1, y_2 such that $s_1 = x_1 \ltimes y_1$ and $s_2 = x_2 \ltimes y_2$ with $x_1 \sim x_2$ and $y_1 \sim y_2$. By induction hypothesis we have that

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

  Then we have, by definition of $\ltimes$,

$${}^{\mathfrak{h}}s_1 = {}^{\mathfrak{h}}(x_1 \ltimes y_1) = {}^{\mathfrak{h}}x_1 \overset{IH}{=} {}^{\mathfrak{h}}x_2 = {}^{\mathfrak{h}}(x_2 \ltimes y_2) = {}^{\mathfrak{h}}s_2$$

  and

$${}^{\mathfrak{t}}s_1 = {}^{\mathfrak{t}}(x_1 \ltimes y_1) = y_1 \ltimes {}^{\mathfrak{t}}x_1 \quad \text{and} \quad {}^{\mathfrak{t}}s_2 = {}^{\mathfrak{t}}(x_2 \ltimes y_2) = y_2 \ltimes {}^{\mathfrak{t}}x_2.$$

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

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

Since, for every s, $\phi_1\, s \sim \phi_2\, s$ by R0, we conclude that $\phi_1\, s = \phi_2\, s$ and therefore ϕ_1 and ϕ_2 are extensionally equal. □

Although Example 3.1 does not satisfy the guardedness condition, it is easy to see why it has a unique solution: the equation guarantees that at least one element will

be produced by ϕ, and the $(\ltimes)$ operator outputs the elements generated by the two recursive calls, ensuring productivity. We could extend the notion of guardedness to take such obvious cases into account, but we would still miss some of the tricky examples shown next.

It is possible to associate a coalgebra to the equation in Example 3.1. We leave this to the reader: The construction is similar to what we do for the next example.

Example 3.3 In the following example, the recursive call occurs under an application of the tail function, making even an extended guardedness check fail:

$$\chi : \mathbb{S} \to \mathbb{S}$$
$$\chi\ s = {}^{\mathfrak{h}}s :: {}^{\mathfrak{t}}s \ltimes {}^{\mathfrak{t}}(\chi\ {}^{\mathfrak{t}}s)$$

To illustrate it, here is its output when it is applied to the stream of natural numbers $\mathsf{nat} = 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: \ldots$:

$$\chi\ \mathsf{nat} = 0 :: 1 :: 2 :: 2 :: 3 :: 3 :: 3 :: 4 :: 4 :: 5 :: 4 :: 6 :: 4 :: 7 :: 5 :: 8 :: 5 :: 9 :: 6 :: 10 :: \ldots.$$

Proposition 3.4 *The equation in Example 3.3 has at most one solution.*

Proof. Suppose χ_1 and χ_2 are two solutions to the equation. We want to prove that they are extensionally equal. Towards that goal, we define an inductive relation on $\mathbb{S}$, we prove that it is a bisimulation and we show that equality of the two functions follows by the coinduction principle. (The next section gives the algorithm to automatically generate the rules of the inductive relation from the recursive equation.)

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

$$\frac{s : \mathbb{S}}{\chi_1\ s \sim \chi_2\ s}(R0) \qquad \frac{s, x_1, x_2 : \mathbb{S} \quad x_1 \sim x_2}{s \ltimes {}^{\mathfrak{t}}x_1 \sim s \ltimes {}^{\mathfrak{t}}x_2}(R1) \qquad \frac{s, x_1, x_2 : \mathbb{S} \quad x_1 \sim x_2}{x_1 \ltimes s \sim x_2 \ltimes s}(R2).$$

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

- If the last rule used was R0, then it must be $s_1 = \chi_1\ s$ and $s_2 = \chi_2\ s$ for some s. Then, because χ_1 and χ_2 satisfy the equation in Example 3.1, we have

$$s_1 = \chi_1\ s = {}^{\mathfrak{h}}s :: {}^{\mathfrak{t}}s \ltimes {}^{\mathfrak{t}}(\chi_1\ {}^{\mathfrak{t}}s), \qquad s_2 = \chi_2\ s = {}^{\mathfrak{h}}s :: {}^{\mathfrak{t}}s \ltimes {}^{\mathfrak{t}}(\chi_2\ {}^{\mathfrak{t}}s).$$

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

$${}^{\mathfrak{t}}s_1 = {}^{\mathfrak{t}}s \ltimes {}^{\mathfrak{t}}(\chi_1\ {}^{\mathfrak{t}}s), \qquad {}^{\mathfrak{t}}s_2 = {}^{\mathfrak{t}}s \ltimes {}^{\mathfrak{t}}(\chi_2\ {}^{\mathfrak{t}}s).$$

  By R0 we have that $\chi_1\ {}^{\mathfrak{t}}s \sim \chi_2\ {}^{\mathfrak{t}}s$, therefore ${}^{\mathfrak{t}}s_1 \sim {}^{\mathfrak{t}}s_2$ by R1 (with the instantiations $s := {}^{\mathfrak{t}}s$, $x_1 := \chi_1\ {}^{\mathfrak{t}}s$ and $x_2 := \chi_2\ {}^{\mathfrak{t}}s$.)

- If the last rule used was R1, then there must be streams s, x_1, x_2 such that $s_1 = s \ltimes {}^{\mathfrak{t}}x_1$ and $s_2 = s \ltimes {}^{\mathfrak{t}}x_2$ with $x_1 \sim x_2$. By induction hypothesis we have that ${}^{\mathfrak{h}}x_1 = {}^{\mathfrak{h}}x_2$ and ${}^{\mathfrak{t}}x_1 \sim {}^{\mathfrak{t}}x_2$. Then we have, by definition of $\ltimes$,

$$ {}^{\mathfrak{h}}s_1 = {}^{\mathfrak{h}}(s \ltimes {}^{\mathfrak{t}}x_1) = {}^{\mathfrak{h}}s = {}^{\mathfrak{h}}(s \ltimes {}^{\mathfrak{t}}x_2) = {}^{\mathfrak{h}}s_2 $$

  and

$$ {}^{\mathfrak{t}}s_1 = {}^{\mathfrak{t}}(s \ltimes {}^{\mathfrak{t}}x_1) = {}^{\mathfrak{t}}x_1 \ltimes {}^{\mathfrak{t}}s \quad \text{and} \quad {}^{\mathfrak{t}}s_2 = {}^{\mathfrak{t}}(s \ltimes {}^{\mathfrak{t}}x_2) = {}^{\mathfrak{t}}x_2 \ltimes {}^{\mathfrak{t}}s. $$

  But we know that ${}^{\mathfrak{t}}x_1 \sim {}^{\mathfrak{t}}x_2$ by induction hypothesis, so ${}^{\mathfrak{t}}s_1 \sim {}^{\mathfrak{t}}s_2$ by R2 (with the instantiations $s := {}^{\mathfrak{t}}s$, $x_1 := {}^{\mathfrak{t}}x_1$ and $x_2 := {}^{\mathfrak{t}}x_2$).
- If the last rule used was R2, then there must be streams s, x_1, x_2 such that $s_1 = x_1 \ltimes s$ and $s_2 = x_2 \ltimes s$ with $x_1 \sim x_2$. By induction hypothesis we have that ${}^{\mathfrak{h}}x_1 = {}^{\mathfrak{h}}x_2$ and ${}^{\mathfrak{t}}x_1 \sim {}^{\mathfrak{t}}x_2$. Then we have, by definition of $\ltimes$,

$$ {}^{\mathfrak{h}}s_1 = {}^{\mathfrak{h}}(x_1 \ltimes s) = {}^{\mathfrak{h}}x_1 \overset{IH}{=} {}^{\mathfrak{h}}x_2 = {}^{\mathfrak{h}}(x_2 \ltimes s) = {}^{\mathfrak{h}}s_2 $$

  and

$$ {}^{\mathfrak{t}}s_1 = {}^{\mathfrak{t}}(x_1 \ltimes s) = s \ltimes {}^{\mathfrak{t}}x_1 \quad \text{and} \quad {}^{\mathfrak{t}}s_2 = {}^{\mathfrak{t}}(x_2 \ltimes s) = s \ltimes {}^{\mathfrak{t}}x_2. $$

  Since $x_1 \sim x_2$ by hypothesis, then ${}^{\mathfrak{t}}s_1 \sim {}^{\mathfrak{t}}s_2$ by R1.

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

Since, for every s, $\chi_1\, s \sim \chi_2\, s$ by R0, we conclude that $\chi_1\, s = \chi_2\, s$ and therefore χ_1 and χ_2 are extensionally equal. □

Let us show, for comparison, how we can prove both existence and unicity of solutions by associating a coalgebra to Example 3.3. The general idea is that we need an *ad hoc* data type to represent all the possible unfoldings of the right-hand side of the equation. It's a set of trees where the nodes represent applications of $\ltimes$ and the leaves represent either a constant stream or the tail of a recursive application of χ.

$$
\begin{array}{ll}
\mathsf{data}\ \mathbb{T}_\chi : \mathsf{Set} & \tau : \mathbb{T}_\chi \to D \times \mathbb{T}_\chi \\
\quad \mathsf{leaf}_\triangle : \mathbb{S} \to \mathbb{T}_\chi & \tau\,(\mathsf{leaf}_\triangle\, s) = \langle {}^{\mathfrak{h}}s, \mathsf{leaf}_\triangle\, {}^{\mathfrak{t}}s \rangle \\
\quad \mathsf{leaf}_\square : \mathbb{S} \to \mathbb{T}_\chi & \tau\,(\mathsf{leaf}_\square\, s) = \langle {}^{\mathfrak{h}1}s, \mathsf{node}\,(\mathsf{leaf}_\square\, {}^{\mathfrak{t}}s)\,(\mathsf{leaf}_\triangle\, {}^{2\mathfrak{t}}s) \rangle \\
\quad \mathsf{node} : \mathbb{T}_\chi \to \mathbb{T}_\chi \to \mathbb{T}_\chi & \tau\,(\mathsf{node}\, t_1\, t_2) = \langle d, \mathsf{node}\, t_2\, t_1' \rangle \ \mathsf{if}\ \langle d, t_1' \rangle = \tau\, t_1.
\end{array}
$$

There exists a unique coalgebra morphism from $\langle \mathbb{T}_\chi, \tau \rangle$ to $\langle \mathbb{S}, \langle {}^{\mathfrak{h}}-, {}^{\mathfrak{t}}- \rangle \rangle$, let's call it χ_τ. We can define $\chi\ s = {}^{\mathfrak{h}}s :: \chi_\tau(\mathsf{node}\,(\mathsf{leaf}_\triangle\, {}^{\mathfrak{t}}s)\,(\mathsf{leaf}_\square\, {}^{\mathfrak{t}}s))$ to obtain a solution of the recursive equation in Example 3.3. On the other hand, if χ is a solution of the recursive equation, we can use it to define a coalgebra morphism from $\mathbb{T}_\chi$ to $\mathbb{S}$ like

this:

$$\chi_\star : \mathbb{T}_\chi \to \mathbb{S}$$
$$\chi_\star(\mathsf{leaf}_\triangle\, s) = s$$
$$\chi_\star(\mathsf{leaf}_\square\, s) = {}^{\mathsf{t}}(\chi\, s)$$
$$\chi_\star(\mathsf{node}\, t_1\, t_2) = (\chi_\star\, t_1) \ltimes (\chi_\star\, t_2).$$

This technique is related to work by Rutten and collaborators on complete sets of co-operations and behavioral differential equations [19,20,12]. It gives a stronger result than the bisimulation construction: it ensures existence besides unicity of the solution.

However, so far no uniform way to construct the *ad hoc* type has been given in a form general enough to cover all the examples in this paper. In some other cases, defining an appropriate data type and a coalgebra on it is less straightforward than above, while the method of proving unicity of solutions using bisimulations still works nicely. Example 4.1 in the next section illustrates this point.

4 General Method

We now show how to apply the method in general. The starting point is a recursive equation defining a function ϕ with several arguments from the domain D and from $\mathbb{S}_D$; so $\phi : D^m \to \mathbb{S}^n \to \mathbb{S}$.

The general form of the equation is

$$\phi\, \vec{a}\, \vec{s} = t[\vec{a}, \vec{s}, \phi]$$

where $\vec{a}$ is a vector of element variables and $\vec{s}$ a vector of stream variables. The right-hand side term t may contain occurrences of the function ϕ that we are trying to define. We allow only fully applied occurrences, that is, ϕ is always applied to m subterms of type D and n subterms of type $\mathbb{S}$.

The idea is to try to "straighten" the equation by forcing it to be in constructor form, that is, the right-hand side is the application of $(::)$ to two subterms. To that end we can just take the head and tail of the right-hand side and put the equation in the equivalent form

$$\phi\, \vec{a}\, \vec{s} = {}^{\mathsf{h}}(t[\vec{a}, \vec{s}, \phi]) :: {}^{\mathsf{t}}(t[\vec{a}, \vec{s}, \phi]).$$

After reduction and simplification we require that the recursive calls disappear from the head:

$$\phi\, \vec{s} = d[\vec{a}, \vec{s}] :: t_1[\vec{a}, \vec{s}, \phi].$$

We are not going to be any more precise about what is meant by *reduction and simplification*: the details will depend on the nature and strength of the formal theory we are working with. For our purposes, we can just assume that we are given a relation $\rightsquigarrow$ on terms; it has to preserve the denotation of the term and, for the algorithm to be effective, it has to be decidable.

So we will have that $^{\mathfrak{h}}(t[\vec{a},\vec{s},\phi]) \rightsquigarrow d[\vec{a},\vec{s}]$ and $^{\mathfrak{t}}(t[\vec{a},\vec{s},\phi]) \rightsquigarrow t_1[\vec{a},\vec{s},\phi]$. In set theory, we will mean that d and t_1 are terms denoting the same object (element of the set D or infinite sequence of elements) as $^{\mathfrak{h}}t$ and $^{\mathfrak{t}}t$ for every value of the variables. In type theory, we will mean that d and t_1 are reducts or normal forms of $^{\mathfrak{h}}t$ and $^{\mathfrak{t}}t$. The algorithm is modular with respect to the adopted notion of simplification, and in what follows we will not be any more specific about it.

If the equation cannot be simplified into this form, the algorithm fails immediately. If we are very lucky, the term t_1 will satisfy the guardedness condition with respect to all occurrences of ϕ and then the algorithm may terminate with success. But in general, the occurrences of ϕ may not be guarded.

The idea is to repeat this kind of *splitting and simplification* on the term t_1 and continue in this vein until the guardedness condition is satisfied. This, however, may never happen.

To increase our chances of success, we also apply a *generalization* operation that replaces some subterms with parameters and variables. The generalized term cannot be used to prove guardedness directly anymore, but it can be used to generate a derivation rule for the bisimulation.

Algorithm

The input of the algorithm is a recursive equation for a function ϕ of the form

$$\phi : D^m \to \mathbb{S}^n \to \mathbb{S}$$
$$\phi\,\vec{a}\,\vec{s} = t[\vec{a},\vec{s},\phi]$$

We assume that there are no nested applications of ϕ: the problem of extending the algorithm to nested recursion is still open. We introduce two variables ϕ_1 and ϕ_2 denoting two solutions of the equation. The output of the algorithm is a bisimulation relation $\sim$ given by a sequence of inductive rules. The very first rule states that the results of ϕ_1 and ϕ_2 are related and can be used together with the coinduction principle to prove that the two solutions are extensionally equal.

The algorithm generates an infinite sequence of terms $t_0, t_1, t_2, \ldots$. Each term contains occurrences of three kinds of variables: *object parameters* are variables of type D denoted by a, b; *stream parameters* are variables of type $\mathbb{S}$ denoted by r, s; *recursive variables* are variables of type $\mathbb{S}$ denoted by x, y, z. The difference between parameters and recursive variables lies in their use in the generation of the rules for the bisimulation relation. Some of the terms may also depend on the function variable ϕ itself.

To each of these terms, $t[\vec{a},\vec{s},\vec{x},\phi]$ we associate the following rule for the bisimulation relation:

$$\frac{\overrightarrow{a : D} \quad \overrightarrow{s : \mathbb{S}} \quad \overrightarrow{x_1, x_2 : \mathbb{S}} \quad \overrightarrow{x_1 \sim x_2}}{t[\vec{a},\vec{s},\vec{x_1},\phi_1] \sim t[\vec{a},\vec{s},\vec{x_2},\phi_2]}$$

where we use a vector notation to summarize sequences of assertions: for example $\overrightarrow{s : \mathbb{S}}$ is short for the sequence of assumptions $s_1 : \mathbb{S}, s_2 : \mathbb{S}, \ldots$ for all the variables

in $\vec{s}$; similarly $\overrightarrow{x_1 \sim x_2}$ is short for $x_{1,1} \sim x_{2,1}, x_{1,2} \sim x_{2,2}$ and so on for all variables in $\vec{x}$.

Base case: $t_0[\vec{a}, \vec{s}, \phi] = (\phi\, \vec{a}\, \vec{s})$.

Recursive step: t_{k+1} is generated from t_k in the following way. We take the head and tail of t_k and simplify them as described in the two steps below. During simplification, we are allowed to replace the head of any recursive variable x with a fresh object parameter d_x and its tail with a fresh recursive variable x'. In other words, we postulate $x = d_x :: x'$.

- First take $^{\mathfrak{h}}t_k$ and reduce/simplify it as much as possible. If the reduced form does not contain any occurrence of ϕ or of recursive variables, then proceed to the next point. Otherwise the algorithm terminates with a negative response: *no bisimulation could be constructed.*
- Next take $^{\mathfrak{t}}t_k$ and reduce/simplify it as much as possible. Let's say $^{\mathfrak{t}}t_k \rightsquigarrow t'_k$. Then apply the following *generalization* operations.

  Replace every maximal subterm not containing any occurrence of recursive variables or ϕ with a new parameter: if $t'_k = C[u]$ where u is a maximal subterm of type D not containing ϕ or any recursive variable, then generalize it to $C[b]$ where b is a fresh object parameter; if $t'_k = C[v]$ where v is a maximal subterm of type $\mathbb{S}$ not containing ϕ or any recursive variable, then generalize it to $C[s]$ where s is a fresh stream parameter.

  Replace every recursive application of ϕ with a new recursive variable: if $t'_k = C[(\phi\, \cdots)]$, then generalize it to $C[y]$ where y is a fresh recursive variable.

  Also replace every instance of a previous term in the sequence with a new recursive variable: if $t'_k = C[t_i[\vec{d}, \vec{s}, \vec{y}, \phi]]$, then generalize it to $C[z]$ where z is a fresh recursive variable.

  After performing all the possible generalizations and simplifications, if the term we obtained is a single parameter or recursive variable, then the algorithm terminates with a positive response: *we constructed a bisimulation.*

  Otherwise we take this term as t_{k+1}, we generate the bisimulation rule associated to it and repeat the recursive step to generate the next term.

Note 1 *By an* instance of a previous term *we mean the result of applying a substitution to an earlier term in the sequence. We can substitute a parameter with any term not depending on recursive variables or ϕ, and we can substitute a recursive variable with either x' or $d_x :: x'$.*

Note 2 *In the reduction/simplification of $^{\mathfrak{t}}t_k$ we must at first not split $d_x :: x'$, that is, we should treat this term as the single recursive variable x. We perform all the simplification and generalization steps and check if the term is an instance of a previous term in the sequence. If it isn't, then we lazily reduce the term, splitting d_x away from x' as necessary, to obtain further simplification.*

Note 3 *It will be observed that, in the case that there are no nested application of it, ϕ will occur only in t_0 and will be substituted with a recursive variable everywhere in t_1. However, we maintained the generalization case for it in the recursive step because it will not be so anymore when we extend the algorithm to nested recursion.*

We illustrate the algorithm on a new example.

Example 4.1

$$\psi : \mathbb{S} \to \mathbb{S}$$
$$\psi\ s = {}^{\mathfrak{h}}s :: \mathsf{even}(\psi(\mathsf{odd}\ {}^{\mathfrak{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\ {}^{\mathfrak{t}}s))$$

$\psi\ \mathsf{nat} = 0 :: 2 :: 5 :: 12 :: 25 :: 52 :: 105 :: 212 :: 425 :: 852 :: 1705 :: 3412 :: 6825 :: 13652 :: 27305 :: 54612 :: 109225 :: 218452 :: 436905 :: 873812 :: \ldots$.

The application of the algorithm to this equation proceeds as follows.

- We start with the base term and its associated derivation rule:

$$t_0 = \psi\, s, \qquad \frac{s : \mathbb{S}}{\psi_1\, s \sim \psi_2\, s}(R0).$$

- ${}^{\mathfrak{h}}t_0 = {}^{\mathfrak{h}}(\psi\, s) = {}^{\mathfrak{h}}s$ by the equation. This expression does not depend on ψ or any recursive variable, so it's OK.

  ${}^{\mathfrak{t}}t_0 = {}^{\mathfrak{t}}(\psi\, s) = \mathsf{even}(\psi(\mathsf{odd}\ {}^{\mathfrak{t}}s)) \ltimes \mathsf{odd}(\psi(\mathsf{even}\ {}^{\mathfrak{t}}s))$. We apply the generalization operation to this term. The two calls to ψ are replaced with recursive variables, $\psi(\mathsf{odd}\ {}^{\mathfrak{t}}s)$ by x and $\psi(\mathsf{even}\ {}^{\mathfrak{t}}s)$ by y. Thus we obtain the next term in the sequence and the next rule:

$$t_1 = \mathsf{even}\, x \ltimes \mathsf{odd}\, y, \qquad \frac{x_1, x_2, y_1, y_2 : \mathbb{S} \quad x_1 \sim x_2 \quad y_1 \sim y_2}{\mathsf{even}\, x_1 \ltimes \mathsf{odd}\, y_1 \sim \mathsf{even}\, x_2 \ltimes \mathsf{odd}\, y_2}(R1).$$

- ${}^{\mathfrak{h}}t_1 = {}^{\mathfrak{h}}(\mathsf{even}\, x) = {}^{\mathfrak{h}}x = d_x$. Remember that we are allowed to replace the head of a recursive variable with a fresh parameter, so the resulting expression does not depend on the recursive variable anymore and it's OK.

  ${}^{\mathfrak{t}}t_1 = \mathsf{odd}\, y \ltimes {}^{\mathfrak{t}}(\mathsf{even}\, x) = \mathsf{even}\ {}^{\mathfrak{t}}y \ltimes \mathsf{odd}\ {}^{\mathfrak{t}}x = \mathsf{even}\, y' \ltimes \mathsf{odd}\, x'$, where we were allowed to replace the tails of recursive variables with fresh variables x' and y'. This expression is actually an instance of t_1 (with $x := y'$ and $y := x'$) so we can replace it with a single recursive variable z. Since the resulting term is a single variable, the algorithm stops with success.

The algorithm constructed an inductive relation $\sim$ with two introduction rules, R0 and R1. The reader can verify, in the way we did for the examples in the previous section, that it is a bisimulation relation and therefore, by the coinduction principle, we can conclude unicity of solution for the equation.

The reader is also invited to check that the bisimulation relations used in the previous section are exactly the ones produced by the algorithm on those examples.

Theorem 4.2 *For every corecursive equation on streams not containing nested recursive calls, if the algorithm terminates with a positive answer or diverges, then the equation has at most one solution.*

Proof. (Sketch) The algorithm generates a sequence of derivation rules for the relation $\sim$. If we can prove that it is a bisimulation, then the coinduction principle,

together with rule R0, implies that any two solutions of the equation must be extensionally equal.

The proof that $\sim$ is a bisimulation proceeds by induction on its generation. For every rule, we must prove that the related terms in the conclusion must have equal heads and similar tails.

In fact, in the recursive case of the algorithm we check first of all if the head of t_n (of which the heads of the related terms are an instantiation) does not depend on ϕ or on recursive variables. If it is in fact so, then the two heads are syntactically identical and we have proved the first part of our goal. Note that the allowance for the head of a recursive variable to be replaced by a fresh parameter corresponds to an application of the induction hypothesis for it.

Similarly, when proving that the tails are similar, recursive variables in the term correspond to subterms that are similar either by assumption (the original recursive variables) or inductive hypothesis (the recursive variables x' introduced as tails of the original ones) or by the application of one of the derivation rules (the recursive variables replacing instances of previous terms). We are allowed to apply any of the previously constructed rules: this corresponds to substituting any instance of a previous term with a recursive variable. Also, we are allowed to replace tails of recursive variables with fresh variables because of the corresponding induction hypothesis. If the simplified term is just a variable, then we know that the tails can be proved similar. Otherwise we add a new rule stating exactly that the generalizations of the tails are similar. □

5 Conclusion

We defined a partial algorithm that, when applied to a corecursive equation on streams, investigates the unicity of its solutions by constructing a bisimulation relation. If the algorithm terminates successfully or diverges, the defined relation is indeed a bisimulation and the equation has at most one solution.

Among the related literature, the studies most similar to this work are the ones on pebbleflow networks [7] and on circular corecursion [18].

The first reference tackles the same problem: proving unicity (and existence) of solutions for recursive equations on streams. Their method is quite different, consisting in the generation and analysis of a king of circuit associated to the equation. They characterize the class of equations on which their technique can decide productivity. The examples analyzed in this paper, however, do not belong to that class.

The second method, implemented in the system CIRC, addresses a more general problem: proving the equality of functions satisfying different equations. The technique is very similar to the algorithm presented here: a sequence of unfoldings of the desired equality is generated until an instance of a previous goal is encountered. However, due to the fact that they compare terms with different structures, it is not possible to perform a generalization step as in the present work, which relies on abstracting corresponding subterms. For this reason CIRC cannot solve

the equalities of different solutions to the equations presented in this paper.

The results of this article point to four directions for further research.

Meaning of failure. We have seen that the algorithm is sound in the sense that whenever it gives a positive outcome, unicity of solutions obtains. Moreover, if the algorithm diverges, we proved that we also have unicity. On the other hand, if the algorithm terminates with failure, all we can say is that we couldn't construct a bisimulation. This does not necessarily mean that the equation does not satisfy unicity of solutions. The generalization step in the algorithm obliterates some of the information contained in the term. The lost information might be essential for productivity. Future work will concentrate on the analysis of the failure cases.

Nested recursive equations. We postulated that the equations we consider don't contain any nested recursive call. A simple equation with recursive nesting is the following:

$$\theta : \mathbb{S} \to \mathbb{S}$$

$$\theta\ s = {}^{\mathfrak{h}}s :: \theta(\theta(\mathsf{even}\ {}^{\mathfrak{t}}s)) \ltimes {}^{\mathfrak{t}}s$$

$\theta\ \mathsf{nat} = 0 :: 1 :: 1 :: 3 :: 2 :: 3 :: 3 :: 7 :: 4 :: 3 :: 5 :: 7 :: 6 :: 7 :: 7 :: 15 :: 8 :: 5 :: 9 :: 7 :: \ldots$. This equation appears to be productive, but our algorithm is not refined enough to deal with it. Future work will try to extend the method to nested recursion.

Application to other coinductive types. Finally, we plan to extend the field of research beyond the application to stream functions, and study if similar results can be obtained for other coinductive data types, for example infinite trees. The final goal could be a general algorithm applicable to corecursive equations on any coinductive structure.

References

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

[2] Peter Aczel and Nax Paul Mendler. A final coalgebra theorem. In D. H. Pitt, D. E. Rydehead, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 357–365. Springer-Verlag, 1989.

[3] Thorsten Altenkirch and Nils Anders Danielsson. Mixing induction and coinduction. http://www.cs.nott.ac.uk/~nad/publications/danielsson-altenkirch-mixing.html, 2009.

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

[5] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[6] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[7] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *FCT 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 2007.

[8] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informatica*, 66(4):353–366, 2005.

[9] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[10] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *LNCS*, pages 140–157. Springer, 1987.

[11] Ralf Hinze. Functional pearl: streams and unique fixed points. In James Hook and Peter Thiemann, editors, *ICFP*, pages 189–200. ACM, 2008.

[12] Clemens Kupke and Jan J. M. M. Rutten. Observational coalgebras and complete sets of co-operations. *Electr. Notes Theor. Comput. Sci.*, 203(5):153–174, 2008.

[13] Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors. *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 of *Lecture Notes in Computer Science*. Springer, 2009.

[14] N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite objects in type theory. In *Proceedings, Symposium on Logic in Computer Science*, pages 249–255, Cambridge, Massachussetts, 16–18 June 1986. IEEE Computer Society.

[15] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[16] Ulf Norell. Dependently typed programming in Agda. Lecture notes from the summer school on Advanced Functional Programming, 2008.

[17] David Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981. Proceedings of the 5th GI-Conference Karlsruhe.

[18] Grigore Rosu and Dorel Lucanu. Circular coinduction: A proof theoretical foundation. In Kurz et al. [13], pages 127–144.

[19] Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[20] Alexandra Silva and Jan J. M. M. Rutten. Behavioural differential equations and coinduction for binary trees. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2007.

[21] Sam Staton. Relating coalgebraic notions of bisimulation. In Kurz et al. [13], pages 191–205.

[22] Hans Zantema. Well-definedness of streams by termination. Proceedings of the 20th Conference on Rewriting Techniques and Applications (RTA), 2009, editor R. Treinen, Springer Lecture Notes in Computer Science.