Approximation of Nested Fixpoints – A Coalgebraic View of Parametric Dataypes

Alexander Kurz[1], Alberto Pardo[2], Daniela Petrişan[3], Paula Severi[1], and Fer-Jan de Vries[1]

1 **Department of Computer Science, University of Leicester, UK**
2 **Instituto de Computación, Universidad de la República, Uruguay**
3 **Radboud University, The Netherlands**

Abstract

The question addressed in this paper is how to correctly approximate infinite data given by systems of simultaneous corecursive definitions. We devise a categorical framework for reasoning about regular datatypes, that is, datatypes closed under products, coproducts and fixpoints. We argue that the right methodology is on one hand coalgebraic (to deal with possible non-termination and infinite data) and on the other hand 2-categorical (to deal with parameters in a disciplined manner). We prove a coalgebraic version of Bekič lemma that allows us to reduce simultaneous fixpoints to a single fix point. Thus a possibly infinite object of interest is regarded as a final coalgebra of a many-sorted polynomial functor and can be seen as a limit of finite approximants. As an application, we prove correctness of a generic function that calculates the approximants on a large class of data types.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases coalgebra, Bekič lemma, infinite data, functional programming, type theory

Digital Object Identifier 10.4230/LIPIcs.CALCO.2015.205

1 Introduction

As forcefully argued in [4], the initial algebra semantics underpins much of the theory of functional programming languages, allowing for structured recursion over data structures. As long as we restrict our attention to the fragment of Haskell programs that actually terminate, this is indeed enough. But what happens if we want to take into account infinite computations? Recursively defined datatypes in Haskell are inherently *coinductive* and, as we shall see in this paper, further complications arise when several nested fixpoints are involved.

Let us consider the parametrised datatype of streams and two functions on integers `const` and `matrix` defined as follows.

```
data Stream a = Cons a (Stream a)

const :: Int -> Stream Int
const n = Cons n (const n)
matrix :: Int -> Stream (Stream Int)
matrix n = Cons (const n) (matrix (n+1))
```

The function `const` takes an integer `n` and evaluates to an *infinite* normal form, the constant stream `n:n:n...` where we abbreviate `Cons` by a colon to improve readability. The function

© Alexander Kurz, Alberto Pardo, Daniela Petrişan, Paula Severi, and Fer-Jen de Vries;
licensed under Creative Commons License CC-BY
6th International Conference on Algebra and Coalgebra in Computer Science (CALCO'15).
Editors: Lawrence S. Moss and Pawel Sobocinski; pp. 205–220

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

`matrix` evaluated at `0` also yields an infinite computation whose normal form is the stream of streams

```
(0:0:0:...):(1:1:1:...):(2:2:2:...):...
```

This is obtained as the limit of the following reduction sequence:

```
matrix 0 → Cons (const 0) (matrix 1)
         → Cons (Cons 0 (const 0)) (Cons (const 1) (matrix 2))
         → ...
```

With finite resources printing an infinite stream like `(0:0...)` is impossible. One could try to see this stream as the limit of an ω long sequence of growing finite terms

`0, 0:0, 0:0:0, ...`

How would this work for the infinite normal form of `matrix 0`? Continuing in the same manner as before would give a converging sequence of length ω^2 of prefixes:

```
0            0:0            ...  (0:0:...)
(0:0:...):1  (0:0:...):1:1  ...  (0:0:...):(1:1:...)
...
```

Note that there is no clue that after `0:0:...` the sequence will continue and indeed a naive Haskell implementation would evaluate only the head of `matrix 0` and thus would render only a sequence of `0`$'s$. In the above sequence of approximants we are missing any indication where the terms are incomplete. A much better sequence of approximants would be

`•, (0:•):•, (0:0:•):(1:•):•, (0:0:0:•):(1:1:•):(2:•):•, ...`

Each of these truncations (or approximating terms) is finite and can in principle be printed, as we show in Section 4. This raises interesting Haskell questions: given a nested datatype `T`, how to define a generic datatype `B T` for such truncations and how to define a generic function `trunc` of type `Nat -> T -> B T` that allows us to print these approximants. Moreover the above sequence of truncations is ω-long, suggesting that conceptually we transformed the two nested fixpoints involved in the datatype `Stream`(`Stream`(`Int`)) into a single fixpoint. Let us analyse this problem from a category theoretic perspective. Consider a category $\mathcal{C}$, assumed for simplicity complete and cocomplete. The datatype `Stream`(X) depending on the parameter X in $\mathcal{C}$ can be regarded as the final coalgebra, i.e. the greatest fixpoint νF_X of a functor $F_X : \mathcal{C} \to \mathcal{C}$ given by $F_X Z = X \times Z$. Therefore the datatype `Stream`(`Stream`(X)) is isomorphic to a final coalgebra of $F_{\texttt{Stream}(X)}$, that is,

$$\nu Y.(\nu Z.X \times Z) \times Y. \tag{1}$$

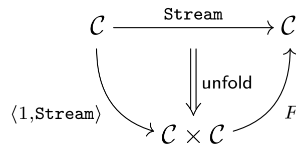
Performing ω reductions of `matrix 0` following the leftmost strategy in the example above is roughly the same as approximating this final coalgebra by iterating only on Z. But the sequence obtained in this way does not converge to the infinite normal form. On the other hand, iterating on Y alone does not work either because $\nu Z.X \times Z$ contains infinite streams which are not printable. The solution is to iterate on Y and Z simultaneously. This can be done by using a version of the Bekič lemma which states that the final coalgebra (1) can be obtained as the first projection of the final coalgebra for the endofunctor on $\mathcal{C}^2$ given by

$$\begin{pmatrix} Y \\ Z \end{pmatrix} \mapsto \begin{pmatrix} Z \times Y \\ X \times Z \end{pmatrix}.$$

Since this is a polynomial functor, a final coalgebra can be obtained as a limit of the final ω-chain in $\mathcal{C}^2$. Thus, the problem of obtaining an ω-long sequence of truncations for an element of a datatype containing several nested fixpoints can be solved by approximating instead elements of a final coalgebra for a polynomial functor on $\mathcal{C}^n$ for a positive integer n.

Another question is how to deal with parameters in a coherent manner. We saw that the parametrised datatype $\mathtt{Stream}(X)$ can be obtained as the fixpoint νF_X of a $\mathcal{C}$-endofunctor. But `Stream` is itself a functor, and this can be *proved* using the universal property of the final coalgebra. One can define its action on arrows (the so called `map` function), using a mediating coalgebra morphism arising from finality (the `unfold` map).

However, a more systematic approach to defining the parametrised datatype is to consider families of fixpoints in one go. So instead of considering a $\mathcal{C}$-endofunctor F_X with a parameter we consider the bifunctor $F : \mathcal{C}^2 \to \mathcal{C}$ given by $F(X, Y) = X \times Y$. This induces the endofunctor $F \circ \langle 1, - \rangle$ on the category $[\mathcal{C}, \mathcal{C}]$ of endofunctors on $\mathcal{C}$, given by $G \mapsto F \circ \langle 1, G \rangle$, as in [20]. The functor `Stream` can be defined as the greatest fixpoint of the higher-order functor $F \circ \langle 1, - \rangle : [\mathcal{C}, \mathcal{C}] \to [\mathcal{C}, \mathcal{C}]$, where we write 1 for identity functors.



We have a canonical natural transformation unfold and for every X in $\mathcal{C}$ the morphism $\mathsf{unfold}_X : \mathtt{Stream}(X) \to X \times \mathtt{Stream}(X)$ is the structure map of the final F_X-coalgebra.

Higher-order functors have been used in [6, 21] to obtain a categorical semantics for nested datatypes, that is, parametrised datatypes defined inductively and in whose declarations the type parameter changes. This approach – inherently 2-categorical – is essential for formalising the relation between the different categories of coalgebras whose final objects are considered.

It remains to understand what is the precise formulation of the Bekič rule that we can apply. The result that allows to transform nested fixpoints into a single one is originally due to Bekič, see [16], and was formulated in terms of least fixpoints of continuous maps on cpo's. Categorical fixpoints rules have been established by Lehmann and Smyth [20] and, under the assumption of algebraic completeness, by Freyd [13] and Fiore [12]. Stronger versions of Freyd's results are given in [2]. We are also indebted to an unpublished note of Pitts [22] which develops a 2-categorical calculus for fixpoints and covers, albeit without proof, simultaneous least fixpoints.

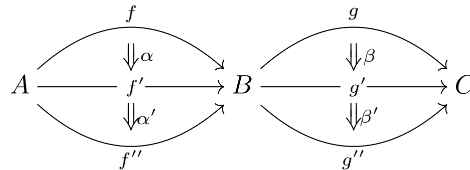
The 2-categorical theory of nested initial algebra has been developed in [7, 11]. Of course these results can be recast in the dual coalgebraic setting. In particular the dual of the Bekič rule we consider in this paper is called the "pairing identity" in [7]. Nevertheless, our proof of this identity (see Theorem 3 and Corollary 4) is rather different and relies on exhibiting some adjunctions between categories of coalgebras (Lemma 5 and Lemma 6). The latter results are interesting in their own right and can also be used for reducing multi-sorted coalgebras to one-sorted ones. The paper is organised as follows. Section 2 establishes adjunctions between several categories of coalgebras and as a consequence we obtain a 2-categorical proof of the Bekič rule. Section 3 explains how to obtain truncations of elements of a final coalgebra. Section 4 gives a generic implementation of the truncations as a function that can operate on a large class of data types and applies the theory developed in the previous sections to prove its correctness.

2 A coalgebraic treatment of the Bekič rule

As argued in the introduction, we need higher-order functors to deal with parametricity, and thus our setting is 2-categorical. For the convenience of the reader we briefly recall basic definitions, but we emphasise that very little of this theory is required to understand our results. Moreover, the basic example of a 2-category is Cat – the category of small categories. The reader unfamiliar with 2-categories is asked to instantiate *0-cell* with *category* (or data type), *1-cell* with *functor* (or parametrised type) and *2-cell* with *natural transformation* (or polymorphic function). Formally, a 2-category $\mathbb{C}$ consists of the following data:

- a class of objects or *0-cells*, denoted $A, B, C, \ldots$
- for any 0-cells A, B a category $\mathbb{C}(A, B)$. Objects of $\mathbb{C}(A, B)$ are called *1-cells* and are denoted by $f : A \to B$, while morphisms in $\mathbb{C}(A, B)$, usually denoted by $\alpha : f \Rightarrow g : A \to B$, or simply $\alpha : f \Rightarrow g$, are called *2-cells*. Composition in $\mathbb{C}(A, B)$ is denoted by $\circ$ and is called *vertical composition*. The identity arrow on $f : A \to B$ will be denoted 1_f, or sometimes just f.
- for any 0-cells A, B, C a functor $* : \mathbb{C}(A, B) \times \mathbb{C}(B, C) \to \mathbb{C}(A, C)$, called *horizontal composition*. The horizontal composition of 2-cells $\alpha : f \Rightarrow f' : A \to B$ and $\beta : g \Rightarrow g' : B \to C$ is denoted by $\beta\alpha : gf \Rightarrow g'f' : A \to C$.
- for any 0-cell A an identity 1-cell $1_A : A \to A$, called the identity on A.

Furthermore, both horizontal and vertical composition are required to be associative and unitary. The graphical representation of 2-cells is very useful since one can compose or *paste* 2-cells in any order. This is sound because from the functoriality of the horizontal composition we obtain the so called interchange law: $(\beta'\alpha') \circ (\beta\alpha) = (\beta' \circ \beta)(\alpha' \circ \alpha)$ for any 2-cells as in the diagram.

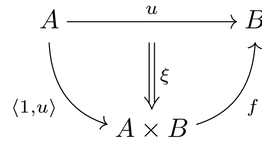


To improve readability of diagrams and for consistency with the 2-category theoretical literature, we use small letters for 1-cells. But, when we instantiate $\mathbb{C}$ with a category of categories, 1-cells correspond to functors, and will be denoted by capital letters.

In what follows we will consider a 2-category $\mathbb{C}$ that has 2-products. This implies the existence of products at the level of 0-cells satisfying the usual universal property; so a pair of 1-cells $p : A \to B$ and $q : A \to C$ yields a unique 1-cell $\langle p, q\rangle : A \to B \times C$. Moreover, any 2-cell of the form $f \Rightarrow g : A \to B \times C$ is essentially a pair $\langle \xi, \zeta\rangle$ with $\xi : \pi_1 f \Rightarrow \pi_1 g : A \to B$ and $\zeta : \pi_2 f \Rightarrow \pi_2 g : A \to C$. For the exact definition, see [17]. We need 2-products to incorporate parameters.

Given a 1-cell $f : A \times B \to B$ we will consider, as motivated by the example given in the introduction, the functor $f\langle 1, -\rangle : \mathbb{C}(A, B) \to \mathbb{C}(A, B)$, that maps a 1-cell $u : A \to B$ to the 1-cell $f\langle 1_A, u\rangle$. To simplify the notation, we will denote the categories of coalgebras for the $\mathbb{C}(A, B)$-functor $f\langle 1, -\rangle$ by $\mathsf{Coalg}^A_B(f)$ and, we will call an $f\langle 1, -\rangle$-coalgebra simply an f-coalgebra. Objects of this category are of the form (u, ξ) where $u : A \to B$ is a 1-cell in $\mathbb{C}$

and ξ is a 2-cell as in the next figure.



A morphism between coalgebras (u,ξ) and (u',ξ') is a 2-cell $\alpha : u \Rightarrow u'$ such that $\xi' \circ \alpha = f\langle 1,\alpha\rangle \circ \xi$.

As an example, instantiate $\mathbb{C}$ to Cat, put $A = B = \mathsf{Set}$ and let the 1-cell f be the bifunctor $F : \mathsf{Set}^2 \to \mathsf{Set}$ mapping (X,Y) to $X \times Y$. The final coalgebra in $\mathsf{Coalg}_B^A(f)$ in this instance is the `Stream` functor described in the introduction.

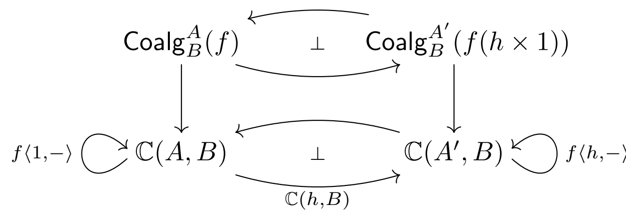
A final object in the category $\mathsf{Coalg}_B^A(f)$, when it exists, will be denoted by $(\nu f, \mathsf{ufld})$. Notice that $\nu f : A \to B$ is a 1-cell.

Aside form having 2-products, we require the 2-categories we consider to have a *stability of fixpoints* property (2) that was also required in [22].

Stability of fixpoints. Consider 1-cells $f : A \times B \to B$ and $h : A' \to A$. Using h we obtain a 1-cell $f(h \times 1) : A' \times B \to B$. This in turn gives rise to the endofunctor $f\langle h,-\rangle$ on $\mathbb{C}(A',B)$, for which we consider the corresponding category of coalgebras $\mathsf{Coalg}_B^{A'}(f(h \times 1))$. Then we require that the final coalgebra $\nu(f(h \times 1))$ exists whenever the final coalgebra νf does, and moreover

$$\nu(f(h \times 1)) = (\nu f)h \tag{2}$$

▶ **Remark 1.** *If for a 1-cell $h : A' \to A$ the functor $\mathbb{C}(h,B) : \mathbb{C}(A,B) \to \mathbb{C}(A',B)$ mapping $u : A \to B$ to $uh : A' \to B$ has a left adjoint,*[1] *then (2) is satisfied. Indeed, since $\mathbb{C}(h,B) \circ (f\langle 1,-\rangle) \simeq (f\langle h,-\rangle) \circ \mathbb{C}(h,B)$, the adjunction between $\mathbb{C}(A,B)$ and $\mathbb{C}(A',B)$ lifts to an adjunction between the corresponding categories of coalgebras:*



In particular, since right adjoints preserve all limits, and hence final objects, the final coalgebra $\nu(f(h \times 1))$ exists whenever νf does, and moreover (2) holds.

The situation we are interested in is the following. Assume we have two 1-cells

$$f : A \times B \times C \to B \qquad g : A \times B \times C \to C$$

such that there exists a final coalgebra in $\mathsf{Coalg}_C^{A \times B}(g)$. We can 'plug in' the 1-cell $\nu g : A \times B \to C$ into f to obtain a 1-cell that we will denote by $f \lhd \nu g : A \times B \to B$, which is

[1] As an example, let $\mathbb{C}$ be the 2-category of locally finitely presentable categories. In this case, left Kan extensions exist, so the functors $\mathbb{C}(h,B)$ have left adjoints. However, in general, the existence of left Kan extensions is a stronger requirement than (2) and is not required in the proof of the Bekič rule.

formally defined as the composition $f\langle \pi_A, \pi_B, \nu g\rangle$. The 1-cell of interest is the final coalgebra

$$\nu(f\langle \pi_A, \pi_B, \nu g\rangle) \tag{3}$$

In Cat when we instantiate f and g to functors $F : \mathcal{A} \times \mathcal{B} \times \mathcal{C} \to \mathcal{B}$ and $G : \mathcal{A} \times \mathcal{B} \times \mathcal{C} \to \mathcal{C}$, the 1-cell in (3) actually gives a functor from $\mathcal{A}$ to $\mathcal{B}$ that for a parameter X in $\mathcal{A}$ computes $\nu Y.F(X, Y, \nu Z.G(X, Y, Z))$. The aim of the generalised Bekič rule is to establish that this fixpoint is the first projection of the greatest fixpoint of the many-sorted functor given by

$$\begin{pmatrix} Y \\ Z \end{pmatrix} \mapsto \begin{pmatrix} F(X,Y,Z) \\ G(X,Y,Z) \end{pmatrix}.$$

Coming back to the 2-categorical setting, we want to find the connection between the final objects (when they exist) of the categories $\mathsf{Coalg}^A_B(f\langle \pi_A, \pi_B, \nu g\rangle)$, respectively $\mathsf{Coalg}^A_{B\times C}(\langle f, g\rangle)$.

▶ **Notation 2.** *Consider 1-cells $f : A \times B \times C \to B$ and $h : A \times B \to C$. Let $f \lhd h$ denote the composition $f\langle \pi_A, \pi_B, h\rangle : A \times B \to B$ and, we abbreviate by $\langle f, h\rangle$ the pair $\langle f, h\langle \pi_A, \pi_B\rangle\rangle : A \times B \times C \to B \times C$. The natural way to obtain from f and h a 1-cell with codomain $B \times C$ is to precompose h with projections and then use the standard product pairing. Denoting this 1-cell by $\langle f, h\rangle$ is only a mild abuse of notation.*

Using this notation, we study the diagram

$$\begin{array}{ccccc}
\mathsf{Coalg}^A_B(f \lhd \nu g) & \underset{I}{\overset{L}{\rightleftarrows}}\ \bot & \mathsf{Coalg}^A_{B\times C}(\langle f, \nu g\rangle) & \xleftarrow{(-)^\dagger} & \mathsf{Coalg}^A_{B\times C}(\langle f, g\rangle) \\
\downarrow & & \downarrow & & \downarrow \\
\mathbb{C}(A, B) & \xleftarrow{\pi_1 -} & \mathbb{C}(A, B \times C) & \xleftarrow{Id} & \mathbb{C}(A, B \times C)
\end{array}$$

where the vertical arrows are forgetful functors and π_1- denotes composition with the projection $\pi_1 : B \times C \to B$. The functors L and I are explained in Lemmas 5, while the functor $(-)^\dagger$ is introduced in Lemma 6. From these lemmas we obtain:

▶ **Theorem 3.** *The diagram above commutes. Further, $\mathsf{Coalg}^A_B(f \lhd \nu g)$ is a full reflective subcategory of $\mathsf{Coalg}^A_{B\times C}(\langle f, \nu g\rangle)$ and $(-)^\dagger$ creates final objects.*

Before stating the two lemmas let us show how the Bekič rule follows.

▶ **Corollary 4.** *Assume $\nu(f \lhd \nu g)$ exists. Then $\nu\langle f, g\rangle$ also exists and we have*

$$\nu(f \lhd \nu g) = \pi_1 \nu\langle f, g\rangle. \tag{4}$$

Proof. If $\mathsf{Coalg}^A_B(f \lhd \nu g)$ has a final object, then by Lemma 5, I preserves it, since as a right adjoint it preserves all limits, see [8, Prop 3.2.2]. Moreover, L also preserves the final coalgebra, since by Lemma 5 L is a reflector, and thus we can use [8, Prop 3.5.3]. By Lemma 6 the functor $(-)^\dagger$ creates final objects, so $\nu\langle f, g\rangle$ exists. We conclude that $L \circ (-)^\dagger$ maps the final coalgebra in $\mathsf{Coalg}^A_{B\times C}(\langle f, g\rangle)$ to the final coalgebra in $\mathsf{Coalg}^A_B(f \lhd \nu g)$. But the composite $L \circ (-)^\dagger$ acts on the carrier 1-cells of the final coalgebras by composing with the projection on the first component. Therefore we obtain (4). ◀

▶ **Lemma 5.** *Consider 1-cells $f : A \times B \times C \to B$ and $h : A \times B \to C$. Then $\mathsf{Coalg}^A_B(f \lhd h)$ is isomorphic to a full reflective subcategory of $\mathsf{Coalg}^A_{B\times C}(\langle f, h\rangle)$.*

Sketch. We exhibit an adjunction $L \dashv I : \mathsf{Coalg}^A_B(f \lhd h) \to \mathsf{Coalg}^A_{B\times C}(\langle f, h\rangle)$ and show that I is full and faithful. I acts on an $f \lhd h$-coalgebra ξ by

$$\left(A \xrightarrow{u} B,\ A \xrightarrow{\langle 1,u\rangle} A\times B \xrightarrow{f\lhd h} B,\ \Downarrow\xi\right) \mapsto \left(A \xrightarrow{\langle u,h\langle 1,u\rangle\rangle} B\times C,\ A \xrightarrow{\langle 1,u,h\langle 1,u\rangle\rangle} A\times B\times C \xrightarrow{\langle f,h\rangle} B\times C,\ \Downarrow\langle \xi,1\rangle\right)$$

A coalgebra homomorphism $\alpha : u \Rightarrow u' : A \to B$ in $\mathsf{Coalg}^A_B(f \lhd h)$ is mapped by I to $\langle \alpha, h\langle 1, \alpha\rangle\rangle$ and it is immediate to verify that this is indeed a coalgebra morphism in $\mathsf{Coalg}^A_{B\times C}(\langle f, h\rangle)$.

The functor L acts on an $\langle f, h\rangle$-coalgebra $\langle \xi, \zeta\rangle$ as follows. We use that $f \lhd h = f\langle 1, h\rangle$.

$$\left(A \xrightarrow{\langle u,v\rangle} B\times C,\ A \xrightarrow{\langle 1,\langle u,v\rangle\rangle} A\times B\times C \xrightarrow{\langle f,h\rangle} B\times C,\ \Downarrow\langle \xi,\zeta\rangle\right) \mapsto \left(A \xrightarrow{u} B,\ A \xrightarrow{\langle 1,u,v\rangle} A\times B\times C \xrightarrow{f} B \ \Downarrow\xi,\ A \xrightarrow{\langle 1,u\rangle} A\times B \xrightarrow{\langle 1,h\rangle} A\times B\times C \ \Downarrow\langle 1,1,\zeta\rangle,\ A\times B \xrightarrow{f\lhd h} B\right)$$

It is routine to check that when $\langle \alpha, \beta\rangle$ is a morphism in $\mathsf{Coalg}^A_{B\times C}(\langle f, h\rangle)$ then α is a morphism in $\mathsf{Coalg}^A_B(f \lhd h)$. Moreover, one readily verifies that L is left adjoint to I. ◀

▶ **Lemma 6.** *Consider 1-cells $\langle f, g\rangle : A \times B \times C \to B \times C$. Then there exists a faithful functor $(-)^\dagger : \mathsf{Coalg}^A_{B\times C}(\langle f, g\rangle) \to \mathsf{Coalg}^A_{B\times C}(\langle f, \nu g\rangle)$ that creates final objects.*

Sketch. Consider a coalgebra $\langle \xi, \zeta\rangle$ in $\mathsf{Coalg}^A_{B\times C}(\langle f, g\rangle)$ as in the left diagram in (5). Then $\zeta : v \Rightarrow g\langle 1, u, v\rangle$ is a coalgebra in $\mathsf{Coalg}^A_C(g(\langle 1, u\rangle \times 1))$. By the stability condition (2), the final object in the latter category is isomorphic to

$$\left(A \xrightarrow{\langle 1,u\rangle} A\times B \xrightarrow{\nu g} C,\ A \xrightarrow{\langle 1,\nu g\langle 1,u\rangle\rangle} A\times C \xrightarrow{\langle 1,u\rangle\times 1} A\times B\times C \xrightarrow{g} C,\ A\times B \xrightarrow{\langle 1,\nu g\rangle} A\times B\times C,\ =,\ \Downarrow\mathsf{ufld}\right)$$

Thus there exists a unique coalgebra morphism $\zeta^\dagger : v \Rightarrow \nu g\langle 1, u\rangle$ such that

$$(\mathsf{ufld}\langle 1, u\rangle) \circ \zeta^\dagger = (g(\langle 1, u\rangle \times 1)\langle 1, \zeta^\dagger\rangle) \circ \zeta$$

We can now define the functor $(-)^\dagger$ on objects by

$$\left(A \xrightarrow{\langle u,v\rangle} B\times C,\ A \xrightarrow{\langle 1,\langle u,v\rangle\rangle} A\times B\times C \xrightarrow{\langle f,g\rangle} B\times C,\ \Downarrow\langle \xi,\zeta\rangle\right) \mapsto \left(A \xrightarrow{\langle u,v\rangle} B\times C,\ A \xrightarrow{\langle 1,\langle u,v\rangle\rangle} A\times B\times C \xrightarrow{\langle f,\nu g\rangle} B\times C,\ \Downarrow\langle \xi,\zeta^\dagger\rangle\right) \tag{5}$$

On arrows the functor $(-)^\dagger$ is defined as identity. One can show that $(-)^\dagger$ creates final objects. ◀

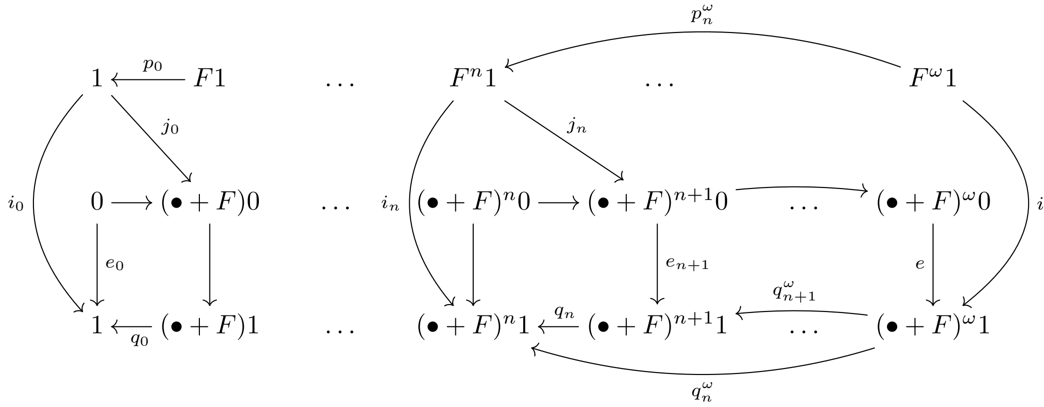


Figure 1 Approximating $F^\omega 1$.

3 Truncating elements of final coalgebras

Having eliminated fixpoints using the Bekič rule, the remaining problem is approximating elements of the final F-coalgebra, for a polynomial F on Set^n for some positive integer n. Even though it is essential for us to allow Set^n for $n > 1$ to account for nested fixpoints, the following simple example is illustrative.

▶ **Example 7.** Let $F : \mathsf{Set} \to \mathsf{Set}$ be given by $FX = \{a, b\} \times X \times X$. We want to approximate infinite binary trees with nodes a, b, that is, elements of the final F-coalgebra, the carrier of which we write as $F^\omega 1$. As a data type of approximants we choose the initial $(\{\bullet\}+F)$-algebra, the carrier of which we write as $(\bullet + F)^\omega 0$. We then have injections i and e

$$F^\omega 1 \xrightarrow{i} (\bullet + F)^\omega 1 \xleftarrow{e} (\bullet + F)^\omega 0$$

In the following we will need the final sequence of F consisting of projections of elements of $F^\omega 1$ and the initial sequence of $(\{\bullet\} + F)$ consisting of the possible truncations of elements of $F^\omega 1$. Both sequences embed into the final sequence of $(\{\bullet\} + F)$, which will be used to define the metric that allows us to capture the approximation of infinite elements of $F^\omega 1$ by truncations.

Writing $\bullet$ for $\{\bullet\}$, this data is made visible in Fig. 1. We consider the horizontal arrows in the middle row as inclusions and do not give them names. Similarly, we will often treat i and e as inclusions and drop them from our notation. i_0, p_0, e_0, q_0 are uniquely determined by their types and j_0 maps the element of 1 to $\bullet$. We put $p_{n+1} = Fp_n$ and for $f \in \{i, e, j, q\}$ we let $f_{n+1} = \mathsf{inr} \circ Ff_n$, where inr is a right coprojection map. The p_n^ω, q_n^ω, and i are determined by $F^\omega 1$ and $(\bullet + F)^\omega 1$ being limits.

▶ **Proposition 8.** *In Fig. 1, we have* $q_n \circ e_{n+1} \circ j_n = i_n$.

We call elements of $(\bullet + F)^n 0$ *truncations* (or approximations).

▶ Remark. Why do we use the $(\bullet + F)^n 0$ and not the $F^n 1$ as the range of truncations? As will become clear in Section 4, in our code we need a datatype for truncations in order to print them. To this end we need to use a constructor, which, in our code, is given by $\bullet$. As far as the implementation is concerned, it is indisputable that we need $\bullet$. The only question that remains is how to interpret the $\bullet$ in the code from a semantic perspective. In particular,

why don't we interpret the $\bullet$ of the program as the element of 1 in $\coprod F^n 1$? There are two reasons for this.

1. It is true that the final sequence suggests considering truncations as elements of $\coprod F^n 1$. However, truncations are required to be finite terms, so from a conceptual point of view, it is natural to regard them as elements of an initial algebra or an inductively defined type. Since the initial algebra of F may be empty, as in the case of streams, we are using the initial algebra of $\bullet + F$. This section carries out the indispensable analysis of the relation between the initial sequence of $\bullet + F$ and the final sequence of F.
   A benefit of this analysis is that we solve a question raised by Barrâs theorem, namely how to approximate a final coalgebra by an initial algebra if the initial algebra sequence is empty (due to $F0 = 0$). To replace F by $\bullet + F$ is an obvious idea, but one needs to deal with the fact that bullets can appear now at all levels and that is what we do in this section.
2. Our methodology of proving the productivity of programs in Section 4 should be able to support correctness proofs of any implementation of printing. It is true that there always is an implementation that prints the $\bullet$ exactly where the final sequence has a $\star$ (if $\star \in 1$ is the element of 1 in the final sequence $F^n 1$). But there are many other implementations. This becomes particularly important in the case of nested fixpoints. The implementation corresponding to the final sequence corresponds to a quite particular strategy of when each fixpoint is unfolded.
   To reinforce this point, let us consider an as an example streams of streams of Int. After applying Bekič, we get the equations

$$T = S \times T$$
$$S = \mathsf{Int} \times S$$

   telling us that S is the type of streams over Int and T is the type of streams over S. Let us develop the final sequence for the functor

$$F(T, S) = (S \times T, \mathsf{Int} \times S).$$

   We use "o" for the 1 of type T and \$ for the 1 of type S. And we write "," for product. The first four elements of the sequence $F^n 1$ are of the following types, with the first line referring to the first component (the T-component) and the second line referring to the second component (the S-component):

$$\begin{array}{llll} o & (\$, o) & (\mathsf{Int}, \$), (\$, o) & (\mathsf{Int}, \mathsf{Int}, \$), (\mathsf{Int}, \$), (\$, o) \\ \$ & (\mathsf{Int}, \$) & (\mathsf{Int}, \mathsf{Int}, \$) & (\mathsf{Int}, \mathsf{Int}, \mathsf{Int}, \$) \end{array} \quad (6)$$

   This corresponds to an implementation that prints n elements of the first stream, $n - 1$ elements of the second stream and so on. But there are many other ways of printing streams of streams. For example, we may want to say that "we want to see more of early streams" and implement printing $2n$ elements of the first stream, $2(n - 1)$ elements of the second stream, etc. So we need truncations of type $((\mathsf{Int}, \mathsf{Int}, \mathsf{Int}, \mathsf{Int}, \$), (\mathsf{Int}, \mathsf{Int}, \$), (\$, o))$ which are not in any $F^n 1$, that is, they don't appear in the upper row of Diagram (6).

▶ **Example 9.** Let $FX = \{a,b\} \times X \times X$. Then (7) shows a truncation that cannot be obtained from any of the F^n1.

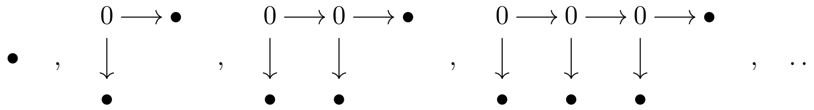
The set of truncations, that is, $(\bullet + F)^\omega 0$, carries a metric induced by the embedding $e : (\bullet + F)^\omega 0 \to (\bullet + F)^\omega 1$ and the final sequence $(\bullet + F)^n 1$ as in Barr [3, Proposition 3.1]. Explicitly, for $t, s \in (\bullet + F)^\omega 1$ let $d(t,s) = 0$ if $t = s$ and else $d(t,s) = 2^{-n}$ where n is the largest natural number such that $q_n^\omega(t) = q_n^\omega(s)$.

Consequently, we have a notion of convergence and Cauchy sequence we can apply to sequences of truncations. For an example consider some $t \in F^\omega 1$ and its *canonical sequence of truncations* $j_n(p_n^\omega(t))_{n<\omega}$:

▶ **Proposition 10.** *For all* $t \in F^\omega 1$ *the sequence* $j_n(p_n^\omega(t))_{n<\omega}$ *converges to* t.

Not all converging sequences of truncations converge to an element in $F^\omega 1$:

▶ **Example 11.** Let $FX = \{0,1\} \times X \times X$. Then the following is a converging sequence of truncations that does not converge to an element of $F^\omega 1$:



The following definition captures the intuition that a productive sequence is a sequence in which all bullets get eventually eliminated (because $j_n(t_n)$ has no bullets below level n):

▶ **Definition 12.** A sequence $(a_k)_{k<\omega}$ is called *productive* if it is Cauchy in $(\bullet + F)^\omega 0$ and if for all n there is $t_n \in F^n 1$ and $k < \omega$ such that $q_n^\omega(e(a_k)) = i_n(t_n)$.

We read $q_n^\omega(e(a_k)) = i_n(t_n)$ as t_n 'is below' a_k. Observe that the t_n in the definition necessarily converge to the same limit as the a_k. For an example note that for any $t \in F^\omega 1$ the canonical sequence converging to t introduced Proposition 10 is productive (the proof uses Proposition 8).

▶ **Corollary 13.** *If* $(a_k)_{k<\omega}$ *is 'a productive sequence of truncations of* t*', that is, if for all* n *there is* k *such that* $q_n^\omega(e(a_k)) = i_n(p_n^\omega(t))$*, then* $\lim a_k = t$.

This shows that all elements of $F^\omega 1$ are approximated by a productive sequence. A stronger statement is:

▶ **Proposition 14.** *Let* $(a_k)_{k<\omega}$ *be a sequence in* $(\bullet + F)^\omega 0$*. Then,* $\lim a_k \in F^\omega 1$ *iff* $(a_k)_{k<\omega}$ *is productive.*

Proof. If $\lim a_n = t \in F^\omega 1$, then for all n there is k such that $q_n^\omega(e(a_k)) = i_n(t_n)$ as required by Def. 12. Conversely, if a_k is productive, then it converges against the same limit as the t_n from Def. 12. Now $q_n^\omega(e(a_k)) = i_n(t_n)$ implies that there is $t \in F^\omega 1$ such that $\lim t_n = t$, hence $\lim a_k = t$. ◀

Our analysis improves somewhat over Barr's original result as we do not have to assume that $F0 \neq 0$ (which excludes e.g. streams). This comes at the cost that $(\bullet + F)^\omega 1$ has 'spurious elements' that are not in $F^\omega 1$. The following summarises two satisfactory characterisations of $F^\omega 1$ as a subset of $(\bullet + F)^\omega 1$.

▶ **Theorem 15.** *Let $F : \mathsf{Set}^n \to \mathsf{Set}^n$ be a many-sorted polyomial functor. An element of $(\bullet + F)^\omega 1$ is in $F^\omega 1$ iff it is approximated by a productive sequence iff it is bullet-free.*[2]

Proof. The first 'iff' is Proposition 14. The second 'only if' is obvious. For the other direction, we show by induction that if an element in $(\bullet + F)^n 1$ is bullet-free then it is already in $F^n 1$, or, in other words, that the image of i_n contains all bullet-free elements of $(\bullet + F)^n 1$. ◀

4 Implementing truncations in Haskell

We apply the theory developed in the previous sections to prove correctness of an effective procedure for printing infinite objects in Haskell. A naive attempt to printing the infinite normal form of the stream of streams `(matrix 0)` only shows an infinite stream of `0`'s and we never see the other streams. Our solution is to print the sequence of truncations of a term. The truncation of a term at depth n is obtained from the tree representation of the term by replacing the subterms at depth n by $\bullet$. Printing the sequence of all truncations is a *faithful* way of printing infinite data if we prove the following two correctness properties:

- the truncations are always finite (and hence, printable in a finite amount of time) and
- the sequence of truncations has length ω and converges to the infinite normal form of the program at issue.

To this end, we assume that our programs are infinitary normalising (productive). For example, we exclude programs that do not have infinite normal form at all such as `(novalue 0)` defined as follows.

```
loop = loop      novalue n = Cons loop (matrix n)
```

The set of (potentially infinite) normal forms can be seen as the carrier of a final coalgebra of the form $F^\omega 1$. Section 4.2 gives the correctness proof of our implementation and relies on the results of Section 3. The full implementation is available at [18].

We give the type declaration of a class `Trunc` introducing a new data type

```
B :: * -> Nat -> *
```

used for implementing the truncations. The truncation of a term of type `a` at level `n` will have type `B a n`. The dependency of `(B a n)` on `n :: Nat` ensures that inhabitants of this data type are finite, and thus amenable to printing.

```
data Nat = Zero | Succ Nat
data SNat n where
  SZ :: SNat Zero
  SS :: SNat n -> SNat (Succ n)
```

```
class Trunc (a :: *) where
  data B :: * -> Nat -> *
  trunc :: SNat n -> a -> B a n
```

[2] A tree is *bullet-free* if it contains no occurrence of $\bullet$.

Note that we fake dependent types in Haskell using data promotion and singletons [26, 10]. The data type `Nat` is promoted to kinds. The singleton type `(SNat n)` can be thought of as having one inhabitant, intuitively, the natural number n.

To illustrate how the function `trunc` works we first present non-generic implementations. We make the parametric datatype `(Stream a)` an instance of the class `Trunc` provided the parameter `a` is also an instance of the class `Trunc`.

```
data B (Stream a) n where
  Bullet :: B (Stream a) Zero
  ConsS :: B a n -> B (Stream a) n -> B (Stream a) (Succ n)

trunc SZ _ = Bullet
trunc (SS n) (Cons x xs) = ConsS (trunc n x) (trunc n xs)
```

We also consider the data type for rose trees which is a multi-way tree structure in which each node may have an arbitrary number of children [5].

```
data RoseTree a = RoseTree a [RoseTree a]
```

The definition of `trunc` needs more care, the following would be wrong:

```
trunc SZ   x          = BulletRose
trunc (SS n) (RoseTree x xs) = RoseTreeB (trunc n x) (map (trunc n) xs)
```

because `(map (trunc n) xs)` is infinite if `xs` is infinite. In order to truncate rosetrees correctly, we replace the second line in the above code by the following:

```
trunc (SS n) (RoseTree x xs) = RoseTreeB (trunc n x) (trunc n xs)
```

where the last truncation is applied to the list `xs`. The theoretical foundation behind this solution is the Bekič rule on datatypes as explained in Section 2, which allows us to rewrite the definition of rose trees $\nu X.A \times (\nu Y.1 + X \times Y)$ as $\Pi_1(\nu(X,Y).(A \times Y, 1 + X \times Y))$. In other words, `(Rosetree a)` is written as the solution of two mutually corecursive equations:

$$\begin{aligned} X &= A \times Y \\ Y &= 1 + X \times Y \end{aligned} \tag{8}$$

After applying the Bekič rule, the many-sorted functor associated to rosetrees is

$$\begin{pmatrix} X \\ Y \end{pmatrix} \xrightarrow{\text{FRT}_A} \begin{pmatrix} A \times Y \\ 1 + X \times Y \end{pmatrix}.$$

In our implementation, since we are using overloading polymorphism we do not see that we actually have two different versions of the function `trunc`, one for each recursive equation (assuming A is a basic type).

4.1 A generic implementation of truncations

In this section, we give a uniform implementation of truncation for a wide class of data types. The data types should have the form $T_1(T_2(\ldots T_n(Int)\ldots))$ where $T_i(X) = \nu Y.F_i(X,Y)$ and F_i is a polynomial functor[3]. The view of data types as fixpoints of functors is implemented through a type class called `Rep` similar to the class `Regular` but for bifunctors instead of functors [23, 25]. Associated to this class there is a type family `FunctorRep`.

---

[3] Actually, *Int* could be replaced by any basic data type.

```
type family FunctorRep (t :: * -> *) :: * -> * -> *
```

This type family can be seen as a function that given a parametric datatype T, it gives a polynomial bifunctor F such that $T(X) = \nu Y.F(X,Y)$. Polynomial bifunctors are represented by the following type constructors which are all made instances of the class `BiFunctor`:

- `U` for constant,
- `P1` for first projection,
- `P2` for second projection,
- `:**:` for product and `:++:` for sum.

```
class Bifunctor f where
 bimap :: (a -> c) -> (b -> d) -> (f a b -> f c d)
```

For our applications, we restrict the type `a` in the constant functor `U a` to be a basic type whose elements are all finitely normalising (i.e. printable in a finite amount of time) such as `Int`. We can now associate `Stream` to the functor `FStream` by means of the type family `FunctorRep` as follows.

```
 type instance FunctorRep Stream = FStream
 type FStream = P1 :**: P2
```

The class `Rep` has two methods that witness the isomorphism in the fixpoint equation $T(A) \cong F(A, T(A))$.

```
class Rep t where
 getRep :: t a -> (FunctorRep t) a (t a)
 fromRep :: FunctorRep t a (t a) -> t a
```

We make `Stream` an instance of the class `Rep` as follows.

```
 getRep (Cons x xs) = P1 x :**: P2 xs
 fromRep (P1 x :**: P2 xs) = Cons x xs
```

Figure 2 illustrates the correspondence between our generic Haskell implementation of truncations and the semantic view given in the previous sections. In our code we have the following definition for the function `trunc`, which we explain below and in Figure 2 step by step.

```
 trunc SZ x = Bullet
 trunc (SS n) x = NextStep (bimap (trunc n) (trunc n) (getRep x))
```

In the second case we define the truncation at level $n+1$ of a term `x` of type `t a`. Semantically, this is a coinductive type $T(A) = \nu Y.F(A,Y)$ obtained as the greatest fixpoint of a functor F, which syntactically is given by `FunctorRep t`.

To compute the truncation of `x` at level $n+1$, first we "unfold" `x` via the function `getRep`. Then we truncate at level n the terms obtained from the unfolding, and we apply F. In our code the application of a bifunctor F to two morphisms is done using the function `bimap`. To be able to use `bimap` in this case, we should require that `FunctorRep t` belongs to the class `Bifunctor`.

The type `B (t a) n` of the truncation is defined as a data type with two constructors `Bullet` and `NextStep` as follows.

```
 data B (t a) n where
  Bullet :: B (t a) Zero
  NextStep :: FunctorRep t (B a n) (B (t a) n) -> B(t a)(Succ n)
```

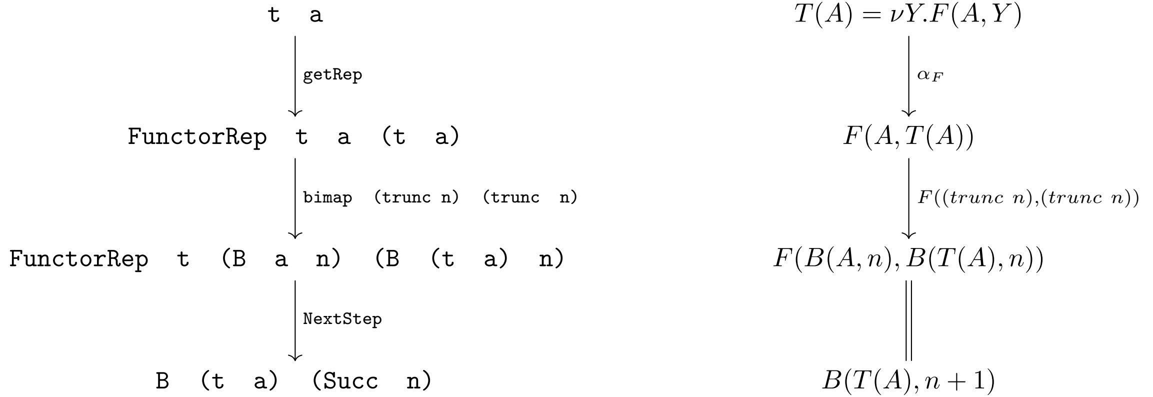


Figure 2 Generic definition of truncations in Haskell.

At a semantic level, $B(T(A),n)$ is defined inductively as follows. For the base type *Int* we have

$$B(Int,0) = \{\bullet\} \qquad B(Int,n+1) = Int$$

while for a coinductive type $T(A) = \nu Y.F(A,Y)$ we have

$$B(T(A),0) = \{\bullet\} \qquad B(T(A),n+1) = F(B(A,n),B(T(A),n))$$

This generic definition of `trunc` does not work if F itself contains a fixpoint. For example, in the case of rosetrees where $F(A,X) = A \times (\nu Y.1 + X \times Y)$ lists are not truncated but remain infinite. We explain briefly how to extend this generic implementation to include data types of the form $T_1(T_2(\ldots T_n(Int)))$ where $T_i(X) = \nu Y.F_i(X,Y)$ and F_i contains a fixed point. As opposed to polynomial F_i, if the functors F_i contain fixpoints, we need to make use of the Bekič rule in the implementation (as well as in the proof of its correctness). This makes it necessary to deal with mutually recursive equations, something that our implementation in terms of the Rep class is not able to manipulate at the moment. There exists a Haskell package that deals with mutually recursive equations generically [24]. We would need to extend this package to include parametric data types.

4.2 Correctness of the implementation

For every natural number n the truncation `(trunc n p)` has a finite normal form $\mathtt{v_n}$ independently of whether the normal form `v` to which `p` evaluates is finite or not. This is proved by induction on n. Assume that the program `p` has type $T_1(T_2(..T_m(Int)))$ where $T_i(X) = \nu Y_i.F_i(X,Y_i)$ and F_i is a polynomial functor for all $1 \leq i \leq m$. To complete the correctness proof we need to show:

$$\mathtt{v} = \lim_{\mathtt{n}\to\infty} \mathtt{v}_n \ .$$

Using Bekič rule (Corollary 4), we have that

$$T_1(T_2(..T_n(Int))) = \Pi_1 \circ \nu \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_m \end{pmatrix} . \begin{pmatrix} F_1(Y_2,Y_1) \\ F_2(Y_3,Y_2) \\ \vdots \\ F_m(Int,Y_m) \end{pmatrix} .$$

Let $F(Y_1, \ldots, Y_m) = (F_1(Y_2, Y_1), F_2(Y_3, Y_2), \ldots F_m(Int, Y_m))$. We also consider the vector $\overrightarrow{\texttt{trunc n}} = (\texttt{trunc\ \ n}, \ldots, \texttt{trunc\ \ n})$ of length m. Then, it is not difficult to show by induction on n and using the definition of `trunc` given in Section 4.1 that `trunc` satisfies the following:

$$\begin{aligned} \overrightarrow{\texttt{trunc\ \ 0}}\ t &= (\bullet, \ldots, \bullet) \\ \overrightarrow{\texttt{trunc\ \ (n+1)}} &= F(\overrightarrow{\texttt{trunc\ \ n}}) \circ \alpha_F \end{aligned}$$

From the above, it is easy to prove by induction on n that $\overrightarrow{\texttt{trunc n}}\ t$ is equal to the canonical sequence $j_n \circ p_n^\omega(t)$ of truncations as defined in Section 3. By Proposition 10, the sequence $\overrightarrow{\texttt{trunc\ \ n}}\ t$ converges to t. It is enough to take a t whose first component is `v` where `v` is the infinite normal form of our original program p of type $T_1(T_2(\ldots T_n(Int)))$.

5 Conclusion

Whereas some of the techniques we used are well known, there are original theoretical contributions (Theorems 3 and 15) as well as a novel solution to the problem of printing infinite datatypes in Haskell. Hutton and Gibbons generalised the approximation lemma from [5] to a certain class of datatypes that includes the polynomial ones [15]. Their definition of approximant does not cover the case of parametric datatypes. Danielsson *et al.* implemented their notion of approximant in the ChasingBottoms package [14], which is implemented using a style of generic programming called *Scrap Your Boilerplate* [19]. Our approach is different, as we use the class `Regular` which views datatypes as fixed points of functors [25].

It will be interesting to compare our work with [1, 9]. It is currently not clear to us whether their recursion schemes are strong enough to define truncations of rose trees.

Of course, printing can be seen as an illustrative example only and others, such as the incremental sending of infinite data over a channel, will be pursued in the future. Moreover, there are many topics we didn't touch upon, such as nested datatypes in the sense of [6], higher order functors, or dependent types, as well as other functional programming languages such as Agda or Coq. It will also be of interest to investigate type theories with an explicit Bekič rule.

Acknowledgements. The fourth author would like to acknowledge a Daphne Jackson fellowship sponsored by EPSRC and the University of Leicester.

References

1 R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25–27, 2013*, pages 197–208, 2013.

2 R. C. Backhouse, M. Bijsterveld, R. van Geldrop, and J. van der Woude. Categorical fixed point calculus. In *Category Theory and Computer Science*, 1995.

3 M. Barr. Terminal coalgebras for endofunctors on sets. *Theoretical Computer Science*, 114(2):299–315, 1999.

4 R. S. Bird and O. de Moor. *Algebra of programming.* Prentice Hall, 1997.

5 R. S. Bird. *Introduction to Functional Programming using Haskell (second edition).* Prentice Hall, 1998.

6 R. S. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2), 1999.

7 S.L. Bloom, Z. Ésik, A. Labella, and E. G. Manes. Iteration 2-theories. *Applied Categorical Structures*, 9(2):173–216, 2001.
8 F. Borceux. *Handbook of Categorical Algebra I.* Cambridge University Press, 1994.
9 A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14, San Diego, CA, USA, January 20-21, 2014*, pages 361–372, 2014.
10 R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Copenhagen*, pages 117–130, 2012.
11 Z. Ésik and A. Labella. Equational properties of iteration in algebraically complete categories. *Theoretical Computer Science*, 195(1):61–89, 1998.
12 M. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, University of Edinburgh, 1994.
13 P. Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science*, volume 77 of *London Math. Soc. Lecture Notes Series*, pages 95—-106. Cambridge University Press, 1992.
14 Haskell Chasing Bottoms Package: For testing partial and infinite values. `http://hackage.haskell.org/package/ChasingBottoms`. Online: accessed March 2015.
15 G. Hutton and J. Gibbons. The generic approximation lemma. *Inf. Process. Lett.*, 79(4):197–201, 2001.
16 C. B. Jones, editor. *Programming Languages and Their Definition – Hans Bekic (1936–1982)*, LNCS 177. Springer, 1984.
17 G. M. Kelly and R. Street. Review of the elements of 2-categories. In *Category Seminar (Proc. Sem. Sydney 1972/73)*, LNM 420, pages 75–103. Springer, 1974.
18 A. Kurz, D. Petrişan, A. Pardo, P. Severi, and F.-J. de Vries. Haskell code for this paper. `http://www.cs.le.ac.uk/people/ps56/code.xml`, 2015.
19 R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI'03*, 2003.
20 D. J. Lehmann and M. B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.
21 C. E. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Asp. Comput.*, 16(1):19–35, 2004.
22 A. Pitts. An elementary calculus of approximations. Unpublished note.
23 Haskell Regular: Generic programming library for regular datatypes. `http://hackage.haskell.org/package/regular`. Online accessed: March 2015.
24 A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP*, 2009.
25 T. van Noort, A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, B. Heeren, and J.P. Magalhães. A lightweight approach to datatype-generic rewriting. *J. Funct. Program.*, 20(3-4):375–413, 2010.
26 B. A. Yorgey, S. Weirich, J. Cretin, S. L. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI 2012*, pages 53–66, 2012.