Contractive Functions on Infinite Data Structures

Venanzio Capretta Graham Hutton
Functional Programming Lab
University of Nottingham, UK
{venanzio.capretta,graham.hutton}@nottingham.ac.uk

Mauro Jaskelioff
CIFASIS–CONICET, Argentina
Universidad Nacional de Rosario, Argentina
mauro@fceia.unr.edu.ar

Abstract

Coinductive data structures, such as streams or infinite lists, have many applications in functional programming and type theory, and are naturally defined using recursive equations. But how do we ensure that such equations make sense, i.e. that they actually generate a productive infinite object? A standard means to achieve productivity is to use Banach's fixed-point theorem, which guarantees the unique existence of solutions to recursive equations on metric spaces under certain conditions. Functions satisfying these conditions are called contractions. In this article, we give a new characterization of contractions on streams in the form of a sound and complete representation theorem, and generalize this result to a wide class of non-well-founded structures, first to infinite binary trees, then to final coalgebras of container functors.

These results have important potential applications in functional programming, where coinduction and corecursion are successfully deployed to model continuous reactive systems, dynamic interactivity, signal processing, and other tasks that require flexible non-well-founded data. Our representation theorems provide a definition paradigm to compactly compute with such data and easily reason about them.

1. Introduction

Coinductive types, data structures with potentially infinite unfolding, are becoming a standard feature of functional programming languages and type theoretic systems. The most well-studied example is streams, infinite sequences of elements. There is a considerable literature devoted to streams, covering their theoretical foundation and programming techniques (Gibbons and Hutton 2005; Rutten 2005; Hinze 2008a,b). In recent years, research has explored more varied kinds of non-well-founded structures (Mendler et al. 1986; Coquand 1993), including infinite trees, interactive processes and games, and, in logical systems, reflexive modalities (Capretta 2007; Cîrstea et al. 2011) and infinitary proof rules. Two recent books give an overview of the area (Sangiorgi 2012a,b).

A crucial issue in programming and reasoning with coinductive types is the convergence of recursive equations. Given a recursive equation that specifies an element of a coinductive type in terms of itself, under what conditions is the existence of a unique solution certain? In the dual case of recursive definitions over inductive types, we are interested in the eventual termination of the unfolding of the equation: the structure should be well-founded and the computation should eventually yield a completed element. On the other hand, the corecursive case requires that the unfolding continuously produces new parts of the structure without getting stuck. This property is known as *productivity*. Intensive research is dedicated to the identification of criteria to ensure productivity.

The basic principle of corecursive programming comes from the categorical characterization of coinductive types as final coalgebras of functors (Jacobs and Rutten 1997b). We can define a unique function into a coinductive type by giving a coalgebra on the domain. This is a simple and theoretically transparent technique, but it does not apply directly to most cases of interest and forces the programmers to rewrite their code, often requiring complex intermediate data structures (Rutten 2003; Endrullis et al. 2010; Capretta 2010, 2011).

A slightly more permissive method allows equations that are *guarded* (Coquand 1993), in which we admit recursive calls as long as they occur under a constructor that ensures that part of the structure is generated before iterating the equation. This methodology is implemented in type-theoretic systems such as Coq (Giménez 1994). It is based on the syntactic form of the recursive definition, and applies to definitions whose productivity can be checked easily by a one-step algorithm. Recent work on definition schemes (Lochbihler and Hölzl 2014), extends the range of functions that are permissible in a proof assistant by exploiting the double nature of lazy lists as both producers and consumers of data; this work also provides associated reasoning principles.

A more comprehensive and mathematically elegant approach appeals to topological and metric concepts. In particular, we can associate to a coinductive type a notion of distance between its elements and exploit standard mathematical theorems that ensure the existence of solutions. The chief among these is Banach's theorem, which states that every contractive function on a complete metric space has a unique fixed point.

Our interest focuses on the application of Banach's theorem to the particular setting of non-well-founded data types. The metric structure, introduced for infinite trees by Arnold and Nivat (1980), uses a notion of distance that measures the similarities between elements: two elements are near if they have a common initial segment. It is easy to verify that the types then become complete metric spaces. A function is contractive if it always decreases the distance of its arguments by a factor smaller than one.

The main original contribution of our work is a new representation theorem: contractive functions are in one-to-one correspondence with elements of an appropriate inductive type. We initially focus on streams, in which setting we provide a simple and effective representation of contractions. We prove that it is sound and complete: it exactly captures the notion of contractive function. We then extend the characterization to richer data structures, first to binary trees and then to final coalgebras of container functors. Although the representation for streams is straightforward, its abstraction to general final coalgebras is far from obvious. We show that most of the conceptual framework that we developed for streams still applies. A complex non-well-founded object can be seen as a stream of *slices*, each adding all the structure needed at a certain depth, an idea by Ghani et al. (2006, 2009a). In full generality this requires the slices to have a type dependent on the previous section of the structure. Our main result provides a sound and complete representation of contractions on a wide class of final coalgebras.

2. Metric Spaces and Banach's theorem

Banach's theorem was originally discovered as a useful tool to prove the unique existence of solutions to differential equations (Banach 1922). The theorem applies in *complete metric spaces*, which are given by a pair (X, d) of a set X and a real-valued function d that measures the distance between two points of the set X. Completeness in this context expresses that every Cauchy sequence converges to a point, where a sequence is Cauchy if the distance between points becomes arbitrarily close.

A *contraction* is a function from the set X to itself that shrinks the distances by a factor smaller than 1 (called the Lipschitz constant). Banach's theorem states that every contraction has a unique fixed point. Its proof is constructive: we can begin with any point and iterate the function, obtaining a Cauchy sequence that converges to the fixed point. Traditionally, the choice of X is a space of analytic functions and the contraction is given by a differential equation. The fixed point is the unique solution to the equation.

Banach's theorem has also proved very useful in theoretical computer science, specifically in domain theory. It is used to give the semantics of recursive types (MacQueen et al. 1984) and the solution of recursive equations on them (Gianantonio and Miculan 2003, 2004). It has previously been applied to streams and infinite trees (Buchholz 2005), with important results in the semantics of reactive programs (Krishnaswami and Benton 2011). In these applications, X is usually a semantic domain, often a space of functions denoted by programs. The distance then measures the information separation between data structures. Banach's theorem provides a method to ensure the convergence of iterative programs and recurrence relations. For an introduction, see Section 6 of Smyth (1992). An alternative approach consists in using a family of converging equivalence relations (Matthews 1999).

Our work in this article follows this line of application to infinite data structures. The space X is a type of non-well-founded elements. The distance d is a measure of the difference between two infinite objects, inversely dependent on the size of their common finite initial segment.

Given the central position of contractive functions in recursive programming with coinductive data, it is important to have a simple characterization of the class of contractions. A straightforward definition imposes the contractivity predicate on a generic function, but a direct representation as a data type is desirable. We provide a concrete characterization of contractions in terms of their computational structure, leading to effective versions of Banach's theorem that can be deployed in concrete programming and reasoning practice.

The original contribution of our work is a sound and complete representation theorem for contractive functions on streams and on final coalgebras of containers.

3. Contractions on Streams

In this section we introduce contractions for the particular case of streams of values over a given type A:

$$\mathsf{codata\ Stream}\,A = \{\mathsf{head} : A\} \triangleleft \{\mathsf{tail} : \mathsf{Stream}\,A\}$$

According to this definition, every stream $t : \mathsf{Stream}\,A$ has the shape $x \triangleleft xs$, where x is an element of the parameter type A and xs is another stream of the same type. Whereas in lists we have a constructor for the empty list, in streams we do not, and therefore every stream must continue forever. If we add such a constructor to the codata definition, we obtain *lazy lists*, which comprise both finite and infinite sequences.

3.1 Recursive Equations

Streams are naturally defined using recursive equations. For example, the constant stream $\mathsf{ones} = 1 \triangleleft 1 \triangleleft 1 \triangleleft \cdots$ can be defined as a single one followed by the stream itself, by means of the following recursive equation:

$$\begin{array}{l} \mathsf{ones} : \mathsf{Stream}\,\mathbb{N} \\ \mathsf{ones} = 1 \triangleleft \mathsf{ones} \end{array}$$

In turn, the stream of natural numbers $\mathsf{nats} = 0 \triangleleft 1 \triangleleft 2 \triangleleft \cdots$ can be defined by starting with the value zero, and then mapping the successor function $(+1)$ over each element of the stream itself to produce the remaining stream of values:

$$\begin{array}{l} \mathsf{nats} : \mathsf{Stream}\,\mathbb{N} \\ \mathsf{nats} = 0 \triangleleft \mathsf{map}\,(+1)\,\mathsf{nats} \\ \\ \mathsf{map} : (A \to B) \to \mathsf{Stream}\,A \to \mathsf{Stream}\,B \\ \mathsf{map}\,f\,(x \triangleleft xs) = f\,x \triangleleft \mathsf{map}\,f\,xs \end{array}$$

(The definition of map is itself recursive: it applies f to the first element of the stream and recursively calls itself on the tail.)

Unfortunately, not all recursive stream equations make sense as definitions of streams. For example, the following is well-typed,

$$\begin{array}{l} \mathsf{loop} : \mathsf{Stream}\,A \\ \mathsf{loop} = \mathsf{tail}\,\mathsf{loop} \end{array}$$

but does not actually define a stream because unfolding the definition loops forever without ever producing any values.

Similarly, attempting to redefine

$$\begin{array}{l} \mathsf{ones} : \mathsf{Stream}\,\mathbb{N} \\ \mathsf{ones} = 1 \triangleleft \mathsf{tail}\,\mathsf{ones} \end{array}$$

is also invalid, because it produces a single one and then loops. However, not all uses of tail in definitions are problematic. For example, the stream ones can be defined as a single 1 followed by the result of interleaving alternative elements from the stream and its tail:

$$\begin{array}{l} \mathsf{ones} : \mathsf{Stream}\,\mathbb{N} \\ \mathsf{ones} = 1 \triangleleft \mathsf{interleave}\,\mathsf{ones}\,(\mathsf{tail}\,\mathsf{ones}) \\ \\ \mathsf{interleave} : \mathsf{Stream}\,A \to \mathsf{Stream}\,A \to \mathsf{Stream}\,A \\ \mathsf{interleave}\,(x \triangleleft xs)\,ys = x \triangleleft \mathsf{interleave}\,ys\,xs \end{array}$$

However, if we swapped the order of the arguments to interleave in the above definition for ones, the definition would again become invalid. This brings us to the following fundamental question: when does a recursive stream equation actually define a stream? In the next few sections we introduce the technical machinery that underlies our particular approach to answering this question.

3.2 Fixed Points

In the previous section we reviewed the idea of streams and stream equations. In this section we consider what these notions mean from a more formal perspective, in terms of solutions of equations and fixed points of functions.

First of all, recall that inductive types are defined as the *least* solution of some equation. For example, the type $\mathbb{N}$ of natural numbers can be defined as the least set X for which there is a bijection $X \cong 1 + X$, where 1 is a singleton set with element $*$, and $+$ is disjoint union of sets with injections inl and inr. The right-to-left component of the bijection, $f : 1 + \mathbb{N} \to \mathbb{N}$, gives the constructors for $\mathbb{N}$ by defining $\mathsf{zero} = f\,(\mathsf{inl}\,*)$ and $\mathsf{succ}\,n = f\,(\mathsf{inr}\,n)$. The left-to-right component $g : \mathbb{N} \to 1 + \mathbb{N}$ gives a form of case analysis, mapping zero to $\mathsf{inl}\,*$ and $\mathsf{succ}\,n$ to $\mathsf{inr}\,n$.

Dually, coinductive types are defined as the *greatest* solution of some equation. In this case the solutions considered are those satisfying the coinduction principle, which states that bisimilar objects are equal: Intuitively, when two entities are indistinguishable by the structure of the equation, they must be equal. For example, the equation $X \cong 1 + X$ also has a greatest solution, given by the type $\mathbb{N}^\infty$ of natural numbers together with an infinite value $inf = \mathsf{succ}\,inf$. (It is possible to construct larger solutions by having many infinite values, but the principle of coinduction will decree that they must all be equal.) In a similar manner, the coinductive type $\mathsf{Stream}\,A$ of streams of type A can be defined as the greatest set X for which there is a bijection $X \cong A \times X$, where $\times$ is Cartesian product of sets with projections fst and snd. The left-to-right component of the bijection, $f : \mathsf{Stream}\,A \to A \times \mathsf{Stream}\,A$, gives rise to the destructors for streams by defining $\mathsf{head}\,xs = \mathsf{fst}\,(f\,xs)$ and $\mathsf{tail}\,xs = \mathsf{snd}\,(f\,xs)$. The right-to-left component $g : A \times \mathsf{Stream}\,A \to \mathsf{Stream}\,A$ gives rise to the constructor for streams by defining $x \lhd xs = g\,(x, xs)$.

Just as types can be defined using equations, so too can values. Consider a recursive equation $xs = f\,xs$ that defines a stream xs in terms of itself and some function f. Any stream that solves this equation for xs is a *fixed point* of f. Hence, solving a stream equation means finding a fixed point of a function on streams. However, not all such functions have fixed points. For example, $\mathsf{map}\,(+1)$ has no fixed point, which corresponds to the fact that $xs = \mathsf{map}\,(+1)\,xs$ is not a valid definition for a stream. (Here we're talking of streams of $\mathbb{N}$; in $\mathbb{N}^\infty$ there exists a fixed point.)

Moreover, some functions have many fixed points. For example, the identity function has any stream as a fixed point, which corresponds to the fact that the equation $xs = xs$ is also an invalid definition for a stream. Note that there is no general notion of ordering on streams, so it does not make sense to consider least or greatest fixed points in this context.

What then makes a valid definition? Our approach is to only consider functions on streams that have a *unique* fixed point, denoted by $\mathsf{fix}\,f$, which is adopted as the semantics of the corresponding recursive equation. For example, the function $\lambda xs.\ 1 \lhd xs$ has a unique fixed point given by the constant stream of ones, which corresponds to the fact that the equation $\mathsf{ones} = 1 \lhd \mathsf{ones}$ is a valid definition for a stream. In conclusion, the question of when a recursive stream equation actually defines a stream can now be rephrased as follows: when does a function $f : \mathsf{Stream}\,A \to \mathsf{Stream}\,A$ have a unique fixed point $\mathsf{fix}\,f : \mathsf{Stream}\,A$?

3.3 Contractive Functions

Our approach to this question is based on an idea from topology: contractive functions. The first step in defining contractions for streams is to provide a measure of the distance between any two streams. The distance between two streams is given by the inverse of the exponential of the length of their longest common prefix. More formally, we define a family of equivalence relations $=_n$ on streams of the same type as follows:

$$\frac{}{xs =_0 ys} \qquad\qquad \frac{x = y \quad xs =_n ys}{(x \lhd xs) =_{n+1} (y \lhd ys)}$$

The distance function between two streams of the same type is then

$$d\,(xs, ys) = \begin{cases} 0 & \text{if } xs = ys \\ \frac{1}{2^m} & \text{otherwise} \end{cases}$$

where $m = \max\{n \mid xs =_n ys\}$, i.e. m is the length of the longest prefix where the two streams coincide.

Checking that the metric space of streams is complete is a matter of routine verification. The notion of contractivity for streams can now be reformulated as follows (see also the notion of *causal stream function* by Hansen et al. (2006)):

Lemma 1 (contractive functions). *A function $f : \mathsf{Stream}\,A \to \mathsf{Stream}\,B$ is* contractive *if and only if $xs =_n ys$ implies $f\,xs =_{n+1} f\,ys$ for all natural numbers n and streams xs and ys of type A.*

Proof. It is easy to see that this notion of contraction is equivalent to the metric one. A Lipschitz constant of $1/2$ will always work. □

The above result states that a function on streams is contractive if, when we appply it to two streams are equal for their first n elements, the results give streams that are equal for $n+1$ elements. We denote the type of contractive functions between streams by $\mathsf{Stream}\,A \to_c \mathsf{Stream}\,B$.

Theorem 1 (Banach's theorem for streams). *Every contractive $f : \mathsf{Stream}\,A \to_c \mathsf{Stream}\,A$ has a unique fixed point $\mathsf{fix}\,f$.*

In summary, the notion of contractivity provides a sufficient condition for the (unique) existence of fixpoints, thus guaranteeing that we obtain well-defined streams from recursive equations. Note, however, that contractivity is only a sufficient condition, as not every function between streams with a unique fixpoint is contractive. For example, if we define

$$\begin{array}{lcl} f & : & \mathsf{Stream}\,\mathbb{N} \to \mathsf{Stream}\,\mathbb{N} \\ f\,xs & = & \mathsf{head}\,(\mathsf{tail}\,xs) \lhd 1 \lhd xs \end{array}$$

then the function f has a unique fixed point, given by the constant stream of ones, but is not contractive. In particular, in the case of $n = 0$ contractivity requires that $\mathsf{head}\,(\mathsf{tail}\,xs) = \mathsf{head}\,(\mathsf{tail}\,ys)$, which is not always true. Notice that this depends on the particular definition of distance that we adopted. With different metrics, we could have different sets of contractions; it is in fact possible to define a distance that makes the above definition satisfy the conditions for application of Banach's theorem.

It is natural to ask what contractivity actually means, i.e. what is being expressed in its definition? More generally, we can ask what kind of functions are contractive, i.e. can the class of contractive functions be characterised in a precise manner? The next section answers this question by providing a sound and complete representation theorem for contractive functions on streams.

4. Representation Theorem

Our main goal is to give a representation of the class of contractive functions as a coinductive data type. In this section we achieve this objective for functions on streams. In later sections we generalize it to infinite binary trees and to a wide class of non-well-founded data structures.

For our purposes, we need that the type $\mathsf{Stream}\,A$ contains at least one element any_A. So we assume that A itself is non-empty and has a distinguished element a_0. This will allow us to build a stream consisting of repetitions of a_0. In our construction, it is completely indifferent what any_A is, we just need to know that there is some element in $\mathsf{Stream}\,A$.

Let us consider a contractive function $f : \mathsf{Stream}\,A \to_c \mathsf{Stream}\,B$. Looking at Lemma 1, we see that contractivity requires that

$$\forall xs, ys. \quad xs =_0 ys \quad \text{implies} \quad f\,xs =_1 f\,ys$$

Since $xs =_0 ys$ is true for any two streams xs and ys, this condition reduces to $\forall xs, ys.\, f\,xs =_1 f\,ys$, i.e. the head of the output should be the same regardless of the input.

Similarly, given two streams xs, ys such that $xs =_1 ys$, i.e. they coincide on the first element, call it $x : A$, then $f\,xs =_2 f\,ys$, so the second element of the output stream can only depend on x. This suggest the following representation:

$$\mathsf{codata\ Gen}\,A\,B = \mathsf{Step}\,\{\mathsf{output} : B;\ \mathsf{cont} : A \to \mathsf{Gen}\,A\,B\}$$

A *generating tree* $t : \mathsf{Gen}\,A\,B$ is a structure that represents a contraction that immediately outputs an element $(\mathsf{output}\,t)$, then reads an element of the input stream, x and continues the computation using the generating tree $(\mathsf{cont}\,t\,x)$ on the tail of the input stream. This is very close to the representation of continuous functions by Ghani et al. (2009b): the difference is that here the actions of producing an output and reading an input are strictly alternated, while in their version it is possible to read several elements at a time without producing a result. The restriction is necessary to obtain a contraction, rather than just a continuous function: only contractions are guaranteed to have fixed points.

More precisely, we define a function gen that takes a generating tree and produces a contraction as follows:

$$\begin{array}{lcl} \mathsf{gen} & : & \mathsf{Gen}\,A\,B \to \mathsf{Stream}\,A \to_c \mathsf{Stream}\,B \\ \mathsf{gen}\,t\,(x \triangleleft xs) & = & \mathsf{output}\,t \triangleleft \mathsf{gen}\,(\mathsf{cont}\,t\,x)\,xs \end{array}$$

The validity of this definition, which requires that the resulting function is contractive, is established by the following result:

Lemma 2. *If t is a generating tree then* gen t *is contractive.*

The proof of the lemma, and of the following theorem, are straightforward. They also follow from the general results for final coalgebras. Dually, every contractive function can be represented as a generating tree by means of the following definition:

$$\begin{array}{lcl} \mathsf{rep} & : & (\mathsf{Stream}\,A \to_c \mathsf{Stream}\,B) \to \mathsf{Gen}\,A\,B \\ \mathsf{rep}\,f & = & \mathsf{Step}\,(\mathsf{head}\,(f\,\mathsf{any}_A), \lambda x.\,\mathsf{rep}\,(\mathsf{tail} \circ f \circ (x\ \triangleleft))) \end{array}$$

The first output will not depend on the input, so we obtain it by applying f to an arbitrary stream of As, which we call any_A. This stream can be constructed for non-empty A, e.g. as a constant stream. The continuation of the tree receives the head x of the input and returns, recursively, the representation of the function on streams that: prepends x, applies f, and takes the tail. The recursive call to rep is valid because it is guarded by the constructor Step; the application of tail is in this case not problematic, since it is under the recursive call.

Using the two conversion functions, we can now formalize the idea that contractions and generating trees are in one-to-one correspondence, i.e. every contraction can be uniquely represented by a generating tree, and vice versa.

Theorem 2 (representation theorem)**.** *The functions* gen *and* rep *form an isomorphism* $\mathsf{Gen}\,A\,B \cong \mathsf{Stream}\,A \to_c \mathsf{Stream}\,B$.

The representation theorem states that instead of defining a function and checking that it is a contraction, we can write a generating tree.

The type $\mathsf{gen}\,A\,B$ is the same used by Altenkirch (2001) to represent functions on lists. Specifically, we have that $\mathsf{gen}\,A\,B \cong \mathsf{List}\,A \to B$, thus obtaining another representation of contractions by list functions. The intuition is that the $(n+1)$st entry of the output is calculated from the list of the first n entries of the input.

How easy is it to build a generating tree? In order to answer this question, we note that generating trees are a coinductively defined data type and therefore can naturally be understood by means of coalgebras. The type of generating trees $\mathsf{Gen}\,A\,B$ is (the carrier of) the final coalgebra for the functor $G\,C = B \times (A \to C)$. This means that it comes equipped with a canonical means of producing generating trees, in the form of an *unfold* (or *anamorphism*) operator (Meijer et al. 1991; Gibbons and Jones 1998). In order to define this operator, we first introduce the notion of a *coalgebra* (Jacobs and Rutten 1997a) for generating trees:

$$\mathsf{Coalg}\,A\,B\,C \ = \ (C \to B) \times (C \to A \to C)$$

That is, a coalgebra for the type $\mathsf{Gen}\,A\,B$ comprises two functions that respectively turn a value of type C into a value of type B and a function of type $A \to C$. We can think of an element of C as an automaton which produces a value in B, then waits for an input in A before making a transition and changing its state.

The unfold operator for producing trees is then defined as follows:

$$\begin{array}{lcl} \mathsf{unfold}_{\mathsf{Gen}} & : & \mathsf{Coalg}\,A\,B\,C \to C \to \mathsf{Gen}\,A\,B \\ \mathsf{unfold}_{\mathsf{Gen}}\,(h,t)\,z & = & \mathsf{Step}\,(h\,z,\ \lambda x.\,\mathsf{unfold}_{\mathsf{Gen}}\,(h,t)\,(t\,z\,x)) \end{array}$$

That is, given a coalgebra $(h : C \to B, t : C \to A \to C)$ and a *seed* value $z : C$, the label of the resulting tree produced by the unfold is given by applying h to the seed z, and the branching function is given by applying t to the seed z and the branching value $x : A$ to obtain a new seed that is then used to produce the remaining levels of the tree in the same manner.

5. Examples

Combining the representation theorem with the use of unfold provides a means of producing contractive functions on streams. In particular, given a coalgebra and a seed value, we first apply unfold to produce a generating tree, then apply gen to turn it into a contractive function. We encapsulate this idea as follows:

$$\begin{array}{l} \mathsf{generate} : \mathsf{Coalg}\,A\,B\,C \to C \to (\mathsf{Stream}\,A \to_c \mathsf{Stream}\,B) \\ \mathsf{generate}\,(h,t)\,z = \mathsf{gen}\,(\mathsf{unfold}\,(h,t)\,z) \end{array}$$

From the point of view of improving efficiency, however, it is desirable to fuse the two functions in this definition together to give a direct recursive definition for generate:

$$\begin{array}{l} \mathsf{generate} : \mathsf{Coalg}\,A\,B\,C \to C \to (\mathsf{Stream}\,A \to_c \mathsf{Stream}\,B) \\ \mathsf{generate}\,(h,t)\,z\,(x \triangleleft xs) = h\,z \triangleleft \mathsf{generate}\,(h,t)\,(t\,z\,x)\,xs \end{array}$$

It is useful now to think of the seed value z as a *state* that represents the input history of the resulting contractive function. In this manner, the above definition expresses that the first value in the output stream is given by applying h to the current state (as it cannot depend on the current or future input values, to ensure contractivity), and the remaining output values are given by applying t to the current state and the first input value x to obtain a new state that is then used to process the tail xs of the input stream in the same way.

Note that one can work with coalgebras instead of generating trees without loss of generality, because generating trees are a particular instance of a coalgebra:

$$\begin{array}{l} \mathsf{treeAsCoalg} : \mathsf{Coalg}\,A\,B\,(\mathsf{Gen}\,A\,B) \\ \mathsf{treeAsCoalg} = (\mathsf{output}, \mathsf{cont}) \end{array}$$

We encapsulate the idea of defining a stream as the unique fixed point of a contractive function produced using the function generate by means of a new fixed point operator (fix is the fixed point operator given by Theorem 1):

$$\begin{array}{l} \mathsf{cfix} : \mathsf{Coalg}\,A\,A\,C \to C \to \mathsf{Stream}\,A \\ \mathsf{cfix}\,(h,t)\,z = \mathsf{fix}\,(\mathsf{generate}\,(h,t)\,z) \end{array}$$

When we define a stream using cfix, we can choose an appropriate state type to represent the history of previous values in the stream, and then define a suitable starting value and coalgebra for this type. Ideally, the state should be compact in terms of space, and the coalgebra should be efficient in terms of time.

Note that we have a choice of how to access previous outputs of the function. The notion of contraction, and the Gen type semantics, allows direct access to the previous element of the output.

Alternatively, we can use the state to store the information about the present output required for the next iteration.

For example, we can define the stream of natural numbers in two ways, both using a natural number as state, which represents the next output value, with starting value zero, and a coalgebra $(h_{\mathsf{nats}}, t_{\mathsf{nats}})$. The first updates the state by replacing it with the successor of the present output value; the second updates it by incrementing it and does not use the present output value at all:

$$\begin{array}{ll}
\mathsf{nats} : \mathsf{Stream}\,\mathbb{N} & \mathsf{nats} : \mathsf{Stream}\,\mathbb{N} \\
\mathsf{nats} = \mathsf{cfix}\,(\mathsf{h_{nats}}, \mathsf{t_{nats}})\,0 & \mathsf{nats} = \mathsf{cfix}\,(\mathsf{h_{nats}}, \mathsf{t_{nats}})\,0 \\
\quad \mathsf{where}\ \ \mathsf{h_{nats}} : \mathbb{N} \to \mathbb{N} & \quad \mathsf{where}\ \ \mathsf{h_{nats}} : \mathbb{N} \to \mathbb{N} \\
\qquad\qquad \mathsf{h_{nats}}\,z = z & \qquad\qquad \mathsf{h_{nats}}\,z = z \\
\qquad\qquad \mathsf{t_{nats}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & \qquad\qquad \mathsf{t_{nats}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\qquad\qquad \mathsf{t_{nats}}\,z\,x = x + 1 & \qquad\qquad \mathsf{t_{nats}}\,z\,x = z + 1
\end{array}$$

The two coalgebras correspond to the following simple generating trees.

$$\begin{array}{l}
\mathsf{fromTree}_1, \mathsf{fromTree}_2 : \mathbb{N} \to \mathsf{Gen}\,\mathbb{N}\,\mathbb{N} \\
\mathsf{fromTree}_1\,n = \mathsf{Step}\,n\,(\lambda x.\mathsf{fromTree}_1\,(x+1)) \\
\mathsf{fromTree}_2\,n = \mathsf{Step}\,n\,(\lambda x.\mathsf{fromTree}_1\,(n+1))
\end{array}$$

Similarly, the stream of Fibonacci numbers can be defined using a state that comprises the next two values, starting value (0,1), and a simple coalgebra on this state:

$$\begin{array}{l}
\mathsf{fibs} : \mathsf{Stream}\,\mathbb{N} \\
\mathsf{fibs} = \mathsf{cfix}\,(\mathsf{h_{fibs}}, \mathsf{t_{fibs}})\,(0,1) \\
\quad \mathsf{where}\ \ \mathsf{h_{fibs}} : (\mathbb{N}, \mathbb{N}) \to \mathbb{N} \\
\qquad\qquad \mathsf{h_{fibs}}(z_0, z_1) = z_0 \\
\qquad\qquad \mathsf{t_{fibs}} : (\mathbb{N}, \mathbb{N}) \to \mathbb{N} \to (\mathbb{N}, \mathbb{N}) \\
\qquad\qquad \mathsf{t_{fibs}}\,(z_0, z_1)\,x = (z_1, z_1 + x)
\end{array}$$

Note that we used the value x, the present output element, to update the state, rather than z_0. The two values are always identical, so the two versions are equivalent. A difference in the two approaches arises if we want to have a more compact state space that does not necessarily store all the information about the output. In that case, we may decide to use the present output value to generate the new state, but we are free to lose information.

As a more complex example, we consider the sequence of Hamming numbers (Dijkstra 1976), natural numbers of the form $2^i 3^j 5^k$, with $i, j, k \in \mathbb{N}$, i.e. natural numbers with no prime factors other than 2, 3, and 5. The sequence consists of Hamming numbers, in increasing order, without repetitions.

For simplicity, we will only consider numbers of the form $2^i 3^j$. The sequence hamming begins with 1, and the rest of the values can be obtained by recursively merging in increasing order the Hamming sequences obtained by multiplying the values in the sequence by two and three respectively:

$$\mathsf{hamming} = 1 \triangleleft (\mathsf{map}\,(2*)\,\mathsf{hamming} \mid\mid \mathsf{map}\,(3*)\,\mathsf{hamming})$$

Given two increasing sequences without duplicates, the operator $||$ constructs an increasing sequence without duplicates:

$$(||) : \mathsf{Stream}\,A \to \mathsf{Stream}\,A \to \mathsf{Stream}\,A$$

$$(x \triangleleft xs) \mid\mid (y \triangleleft ys) = \begin{cases} x \triangleleft (xs \mid\mid (y \triangleleft ys)) & \text{if } x < y \\ y \triangleleft ((x \triangleleft xs) \mid\mid ys) & \text{if } x > y \\ x \triangleleft (xs \mid\mid ys) & \text{if } x = y \end{cases}$$

In order to see hamming as the fixpoint of a contractive function, we note that one may define the $||$ operator using a coalgebra $(\mathsf{h}_{||}, \mathsf{t}_{||})$ and compose it with the function $\lambda x.\,(2*x, 3*x)$:

$$\begin{array}{l}
\mathsf{h_{hamming}} : (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \to \mathbb{N} \\
\mathsf{h_{hamming}} = \mathsf{h}_{||} \\[1ex]
\mathsf{t_{hamming}} : (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \to \mathbb{N} \to (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \\
\mathsf{t_{hamming}}\,s\,x = \mathsf{t}_{||}\,s\,(2*x, 3*x) \\[1ex]
\mathsf{hamming} = \mathsf{cfix}\,(\mathsf{h_{hamming}}, \mathsf{t_{hamming}})\,(1, [], [])
\end{array}$$

The coalgebra for the merge operator has a triple as state, where the first component of the triple is the next item to be outputted, and the other two are finite lists of the unused inputs so far:

$$\begin{array}{l}
\mathsf{h}_{||} : (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \to \mathbb{N} \\
\mathsf{h}_{||}(a, _, _) = a \\[1ex]
\mathsf{t}_{||} : (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \to (\mathbb{N}, \mathbb{N}) \to (\mathbb{N}, [\mathbb{N}], [\mathbb{N}]) \\
\mathsf{t}_{||}\,(_, xs, ys)\,(x, y) = h\,(xs +\!\!+ [x], ys +\!\!+ [y]) \\
\quad \mathsf{where}\,h\,(x :: xs, y :: ys) = \begin{cases} (x, xs, y :: ys) & \text{if } x < y \\ (y, x :: xs, ys) & \text{if } x > y \\ (x, xs, ys) & \text{if } x = y \end{cases}
\end{array}$$

The coalgebra presentation makes it obvious that such a stream has a memory leak, as the history increases whenever the next item on both lists differ and it does not decrease in the other case. This issue is also present in the original formulation, although not in an explicit manner: in practice, however, we need to keep the output stream in memory and have two pointers to the different positions of the next elements to be multiplied by 2 and 3.

6. Contractions on Final Coalgebras

In the previous sections we defined a representation for contractive functions on streams and proved that it is a complete characterization. The facility to define a contraction by a simple coinductive object enhances the practicality of Banach's fixed point theorem to define streams. The theorem says that every contraction on a complete metric space has a unique fixed point. It can be applied in a programming language context by turning a data structure, specifically streams, into a metric space by associating a distance between elements that measures how much they differ. A contraction is a function that shrinks the distances by a factor smaller than 1. Banach's theorem guarantees that we can define a total program by specifying a contraction.

Now we extend these results to richer (dependent) types, providing techniques to construct fixed points on coinductive types defined by *containers*, a general form of data constructors. We characterize the contractive functions between final coalgebras of container functors, using ideas about the representation of continuous functions from Ghani et al. (2006, 2009a,b).

A final coalgebra is the greatest fixed point of a functor $F : \mathsf{Set} \to \mathsf{Set}$ satisfying the bisimulation principle (bisimilar objects are equal). Intuitively, F specifies a collection of forms to build elements and the final coalgebra, νF, is the set of elements obtained by iterating these forms in a potentially infinite structure.

We use the notation $(\nu F, \mathsf{out}_\nu)$ for it, where $\mathsf{out}_\nu : \nu F \to F\,(\nu F)$ is the actual coalgebra. Intuitively, it unpacks the top structure of an element, exposing its overall form and subobjects. Its defining property is that, for every other coalgebra $f : X \to F\,X$ there exists a unique $\phi_f : X \to \nu F$ that commutes with the coalgebras out_ν and f: $\mathsf{out}_\nu \circ \phi_f = (F\,\phi_f) \circ f$. In type theory, coinductive types are often defined by constructors, similarly to inductive types. So the final coalgebra is specified by giving its inverse algebra $\mathsf{in}_\nu : F\,(\nu F) \to \nu F$. This is equivalent since, by Lambek's lemma (Lambek 1968), final coalgebras are always invertible. In more modern approaches (Abel et al. 2013; Abel and Pientka 2013), they are presented by *copatterns*, which are a syntactic equivalent of the components of the final coalgebra, and they are explicitly stratified into *sized types*. See also Kurz et al. (2015) for a categorical account of the construction of parametric coinductive types by stages. A still different account (Atkey and McBride 2013), inspired by the recursion modality of Nakano (2000), uses *clock variables* to represent coinductive elements as processes evolving in time.

The universal property of final coalgebras is the standard definition scheme for functions that produce coinductive objects. Our goal is to extend the range of acceptable definition schemes. Instead of looking for a coalgebra, a user should be able to write down a recursive equation and have it be accepted, provided that it satisfies some conditions. Inspired by the work on streams, we propose that this condition is that the operator given by the recursive equation has to be a contractive function.

In order to generalize the notion of contraction, we need to restrict the class of functors that can be used. Containers are functors whose constructors consist of a shape containing positions where the elements of the base type are inserted. We will see that members of the final coalgebras of containers are, in a sense that we will make precise, generalized dependently typed streams. Therefore the representation of contractions on streams can be adapted once we take into account the way that the dependency of the element type varies along the sequence. We first look at a specific instantiation, non-well-founded binary trees, in the next section. Then we give the full generalization to containers.

7. Contractions on Binary Trees

Our first generalization step is to adapt the results on contractive functions to richer data structures. We start by considering infinite binary trees with nodes labelled by elements of a type A:

$$\mathsf{codata}\ \mathsf{BTree}\,A = \mathsf{Node}\,\{\mathsf{get} : A; \mathsf{left}, \mathsf{right} : \mathsf{BTree}\,A\}$$

Every element of this type, $t : \mathsf{BTree}\,A$, has the shape of a node with two children, $t = \mathsf{Node}\,x\,t_1\,t_2$, where x is an element of the parameter type A and t_1, t_2 are recursive subtrees in $\mathsf{BTree}\,A$. The record functions extract the components of the tree: $\mathsf{get}\,t = x$, $\mathsf{left}\,t = t_1$, $\mathsf{right}\,t = t_2$. Because there is no leaf constructor, the trees are non-well-founded: t_1 and t_2 must also have a node structure with two children each, and so on. In this manner, every path from the root will continue forever.

Our goal now is to precisely characterize the concept of contractive functions between two types of trees, $\mathsf{BTree}\,A$ and $\mathsf{BTree}\,B$. Intuitively, a contraction computes a certain part of the output from a strictly smaller part of the input. A node of the output tree at depth n should depend only on nodes of the input tree at depths less than n. We can imagine the trees as made of subsequent *slices*, each slice consisting of the node elements at the same depth. Then a function is contractive if it computes the nth slice of the output from the slices of the input up to the $(n-1)$th.

We don't deviate much from the stream case: we view trees as streams of slices. The difference is that the type of each slice is different. In particular, a slice at depth n is given by a 2^n-tuple.

The definition of distance between trees is parallel to that between streams, with the only difference in the notion of the family of equivalence relations up to a certain depth. If we see lists of Booleans as paths inside trees, we can define a function extracting the node in the position pointed by the path:

$$\begin{array}{l}
\mathsf{nodeAt} : [\mathbb{B}] \to \mathsf{BTree}\,A \to A \\
\mathsf{nodeAt}\,\mathsf{nil}\,t = \mathsf{get}\,t \\
\mathsf{nodeAt}\,(\mathsf{true} :: bs)\,t = \mathsf{nodeAt}\,bs\,(\mathsf{left}\,t) \\
\mathsf{nodeAt}\,(\mathsf{false} :: bs)\,t = \mathsf{nodeAt}\,bs\,(\mathsf{right}\,t)
\end{array}$$

Then two trees are equivalent at level n if all their nodes with paths of length smaller than n are equal. That is:

$$\begin{array}{l}
t_1 =_n t_2 \quad \text{if and only if} \\
\quad \forall p : [\mathbb{B}], (\mathsf{length}\,p < n) \to \mathsf{nodeAt}\,p\,t_1 = \mathsf{nodeAt}\,p\,t_2
\end{array}$$

Then the definition of distance between trees is exactly the same as the distance between streams and we get the same characterization of contractive functions as previously:

Lemma 3 (contractive functions)**.** *A function $f : \mathsf{BTree}\,A \to \mathsf{BTree}\,B$ is* contractive *if and only if $t_1 =_n t_2$ implies $f\,t_1 =_{n+1} f\,t_2$ for all natural numbers n and trees t_1 and t_2 of type $\mathsf{BTree}\,A$.*

In particular, the root element of the output has depth 0, so it shouldn't depend on the input at all. The function must therefore first of all print this root element. Then it can read the root element of the input and use it in the computation of the rest of the output.

To make this observation into a recursive definition, we use a trick to view the children of a tree as a single double tree. The children of a tree of type $\mathsf{BTree}\,A$ are given by two trees, but they can also be viewed as a single tree with pairs of labels on the nodes, $\mathsf{BTree}\,(A^2)$. Imagine superimposing the two trees: they have the same overall shape with different labels on the nodes; we can encode them into a tree with the same shape with coupled labels. So our contractive function, after producing the root of the output and reading the root of the input, can continue recursively as a contractive function on trees of pairs.

We can encode the above intuition in the following representation, similar to the one we gave in Section 4 for streams:

$$\begin{array}{l}
\mathsf{codata}\ \mathsf{TGen}\,A\,B = \\
\quad \mathsf{Step}\,\{\mathsf{output} : B; \mathsf{cont} : A \to \mathsf{TGen}\,(A^2)\,(B^2)\}
\end{array}$$

An element of $\mathsf{TGen}\,A\,B$ has the form $\mathsf{Step}\,b\,f$: the element b goes in the root of the output tree; the function f expects the root a of the input tree and returns a new contraction on trees of pairs that will be applied to the children of the input tree. Given such a code, we unfold it as a function from trees to trees. To formulate this computation, we use zipping/unzipping operations on trees, defined in a straightforward recursive fashion.

$$\begin{array}{l}
\mathsf{zipTree} : \mathsf{BTree}\,A \to \mathsf{BTree}\,A \to \mathsf{BTree}\,(A^2) \\
\mathsf{unZipTree} : \mathsf{BTree}\,(A^2) \to (\mathsf{BTree}\,A)^2
\end{array}$$

These functions can be defined because BTree has a single constructor and therefore all trees have the same structure. It would not work with data types whose elements can have different shapes. However, we will see later that it is possible to define contractions for data types with different shapes, without the need of such zipping and unzipping operations.

The interpretation of an element of $\mathsf{TGen}\,A\,B$ as a function on trees is given by a computation operator:

$$\begin{array}{l}
\mathsf{genT} : \mathsf{TGen}\,A\,B \to \mathsf{BTree}\,A \to_c \mathsf{BTree}\,B \\
\mathsf{genT}\,(\mathsf{Step}\,b\,f)\,(\mathsf{Node}\,a\,t_1\,t_2) = \mathsf{Node}\,b\,u_1\,u_2 \\
\quad\quad \mathsf{where}\ (u_1, u_2) = \mathsf{unZipTree}\,(\mathsf{genT}\,(f\,a)\,(\mathsf{zipTree}\,t_1\,t_2))
\end{array}$$

When elaborating an input tree $\mathsf{Node}\,a\,t_1\,t_2$, the contraction generates an output of form $\mathsf{Node}\,b\,\cdot\,\cdot$, without the need to look at the input at all. The shape is the only possible shape, the node element b is dictated by the contraction. The computation of the output children may need information from the input. The label a determines the contraction $(f\,a)$ that is used for the continuation. The input children are zipped together into a single tree $(\mathsf{zipTree}\,t_1\,t_2)$ that is elaborated by the continuation contraction. This returns a tree of pairs, that needs to be unzipped to obtain the output children.

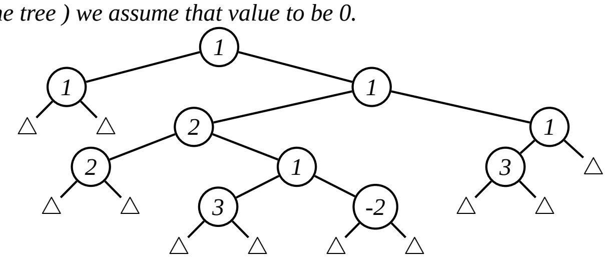
Dually, every contractive function can be represented by a code in $\mathsf{TGen}\,A\,B$. The definition is again similar to that for streams, except that we need some zipping and unzipping, and the type of the representation function depends on the type parameters of the trees. (We use an arbitrary tree any_A, which could be a constant tree with all labels occupied by a designated element of A.)

$$\begin{aligned}
&\mathsf{repT}_{A,B} : (\mathsf{BTree}\,A \to_c \mathsf{BTree}\,B) \to \mathsf{TGen}\,A\,B \\
&\mathsf{repT}_{A,B}\,f = \mathsf{Step}\,(\mathsf{get}\,(f\,\mathsf{any}_A), \lambda x.\,\mathsf{repT}_{A^2,B^2}\,f_x) \\
&\quad \mathsf{where}\ \ f_x : \mathsf{BTree}\,(A^2) \to_c \mathsf{BTree}\,(B^2) \\
&\qquad\qquad f_x\,t = \mathsf{let}\,(t_1,t_2) = \mathsf{unZipTree}\,t \\
&\qquad\qquad\qquad\qquad t_B = f\,(\mathsf{Node}\,x\,t_1\,t_2) \\
&\qquad\qquad\qquad \mathsf{in}\ \mathsf{zipTree}\,(\mathsf{left}\,t_B)\,(\mathsf{right}\,t_B)
\end{aligned}$$

The two conversion functions form an isomorphism which shows that contractions on trees and generating codes are in one-to-one correspondence, i.e. every contraction can be uniquely represented by a generating code, and vice versa.

Theorem 3 (representation theorem)**.** *The functions* genT *and* repT *form an isomorphism* $\mathsf{TGen}\,A\,B \cong \mathsf{BTree}\,A \to_c \mathsf{BTree}\,B$.

Example 1. *Let us illustrate the use of fixpoints of contractions to construct infinite binary trees. We build a tree of integers where the children of a node are, respectively, the sum of its value with its left neighbour and the difference of its value with the right neighbour. When there are no neighbours (on leftmost and rightmost spines of the tree) we assume that value to be 0.*



This example is interesting in this context because the children are not generated simply by their parent, but also they depend on the values of other elements at the same depth.

We can define a function $\mathsf{mnplslice}$ *that yields a new tree slice. The tree generators work on input types that are structured tuples, e.g.* $(A^2)^2$. *We use the notation* $\mathsf{PowType}\,A\,2$ *for such type:*

$$\begin{aligned}
&\mathsf{PowType} : \mathsf{Set} \to \mathbb{N} \to \mathsf{Set} \\
&\mathsf{PowType}\,A\,0 = A \\
&\mathsf{PowType}\,A\,(n+1) = (\mathsf{PowType}\,A\,n)^2
\end{aligned}$$

Then we can easily define the one-step function

$$\mathsf{mnplslice} : (n : \mathbb{N}) \to (\mathsf{PowType}\,A\,n) \to (\mathsf{PowType}\,A\,(n+1))$$

which computes the sums/differences of adjacent elements, e.g.

$$\begin{aligned}
&\mathsf{mnplslice}\,2\,\langle\langle a_0,a_1\rangle,\langle a_2,a_3\rangle\rangle = \\
&\quad\langle\ \ \langle\langle a_0, a_0 - a_1\rangle, \langle a_0 + a_1, a_1 - a_2\rangle\rangle, \\
&\qquad \langle\langle a_1 + a_2, a_2 - a_3\rangle, \langle a_2 + a_3, a_3\rangle\rangle\ \rangle
\end{aligned}$$

The code of the contraction that we need to define our tree is then:

$$\begin{aligned}
&\mathsf{mpgen} : (n : \mathbb{N}) \to (\mathsf{PowType}\,\mathbb{Z}\,n) \to \\
&\qquad\qquad \mathsf{TGen}\,(\mathsf{PowType}\,\mathbb{Z}\,n)\,(\mathsf{PowType}\,\mathbb{Z}\,n) \\
&\mathsf{mpgen}\,n\,v = \\
&\quad \mathsf{Step}\,v\,(\lambda w.\,\mathsf{mpgen}\,(n+1)\,(\mathsf{mnplslice}\,n\,w))
\end{aligned}$$

Remark 1. *We can generalize the representation of contractive functions by using any final coalgebra as codomain, in place of* $\mathsf{BTree}\,B$. *Let* G *be any functor for which the final coalgebra* νG *exists; now we want to characterize the contractions of type* $\mathsf{BTree}\,A \to \nu G$.

Let $\mathsf{in}_G : G(\nu G) \to \nu G$ *be the inverse of the final coalgebra for* G. *The type of contractions is now defined by:*

$$\mathsf{codata}\ \mathsf{TGen}_G\,A = \mathsf{Step}\,(G\,(A \to \mathsf{TGen}_G\,(A^2)))$$

Let us see how to interpret elements of this type as computable functions. An element of $\mathsf{TGen}_G\,A$ *has the form* $\mathsf{Step}\,g$, *where* g *is in* $G\,(A \to \mathsf{TGen}_G\,(A^2))$. *We often see functors as specifying the shape of a data structure, with positions in the shape where substructures are inserted. We will make this intuition formal when we consider containers. We can view* G *as providing the top shape of the output in* νG, *with the positions occupied by functions of type* $A \to \mathsf{TGen}_G\,(A^2)$. *After generating the top shape, without reading any input, the contraction can read the label* a *of the input tree and feed it to these functions, each of which produces a new contraction that can run on the zipping of the children of the input tree. Formally, this spells out the following computation operator.*

$$\begin{aligned}
&\mathsf{genT}_G : \mathsf{TGen}_G\,A \to \mathsf{BTree}\,A \to \nu\,G \\
&\mathsf{genT}_G\,(\mathsf{Step}\,g)\,(\mathsf{Node}\,a\,t_1\,t_2) = \\
&\quad \mathsf{in}_G\,(\mathsf{map}_G\,(\lambda f.\,\mathsf{genT}_G\,(f\,a)\,(\mathsf{zipTree}\,t_1\,t_2))\,g)
\end{aligned}$$

The way it works will be clearer if we instantiate to the previous case of contractions that map trees to trees. In the special case when the output is $\mathsf{BTree}\,B$, *we have* $G\,X = B \times X^2$, $\mathsf{in}_G = \mathsf{Node}$. *In a contraction code of form* $\mathsf{Step}\,g$, *the parameter* g *has type* $B \times (A \to \mathsf{TGen}_G\,(A^2))^2$, *so it will be a triple* $\langle b, f_1, f_2\rangle$. *We unfold the definition of* genT_G.

$$\begin{aligned}
&\mathsf{genT}_G\,(\mathsf{Step}\,\langle b, f_1, f_2\rangle)\,(\mathsf{Node}\,a\,t_1\,t_2) = \\
&\quad \mathsf{Node}\,b\ \ (\mathsf{genT}_G\,(f_1\,a)\,(\mathsf{zipTree}\,t_1\,t_2)) \\
&\qquad\qquad (\mathsf{genT}_G\,(f_2\,a)\,(\mathsf{zipTree}\,t_1\,t_2))
\end{aligned}$$

With respect to our previous definition of genT, *we see that now we use two distinct functions* f_1 *and* f_2 *to produce the left and right child of the output, whereas previously we had a single function* f *that produced a tree of pairs that needed to be unzipped. Otherwise the functions are equivalent. We do not give an inverse representation operator and theorem for this generalization. This requires associating a metric space to the final coalgebra* νG. *We see how to do this when* G *is a container functor in the next section.*

A drawback of this evaluation function is that it is inefficient, because of the zipping and unzipping of trees. We avoided the unzipping of the output in the second version, but we still need to zip the input. We may think of applying some standard fusion techniques to resolve this problem. However, a more elegant solution will come to light when we generalize the construction even further to work on container functors. As the generalization of the codomain type to any final coalgebra produced an optimization at the output side of the computation, a similar generalization of the domain will produce an optimization at the input side.

8. Contractions on Containers

Now we generalize the notion of contraction and the representation theorem to a large class of non-well-founded structures. We want to characterize contractive functions between final coalgebras of general functors. To do this, we need to have a metric on such coalgebras. As before, this can be done if we have a notion of depth and a way of pointing at the parts of the data structure that lie at a given depth. This is possible if the functor has a specific form, which is the case for most commonly used final coalgebras.

A *container* (Abott et al. 2005), also called *dependent polynomial functor* (Gambino and Hyland 2003) in the categorical literature, is a pair $\langle S, P\rangle$ with S : Set, a set of *shapes*, and $P : S \to \mathsf{Set}$, a family of *positions* for every shape. Every container defines a functor:

$$\begin{aligned}
&(S \rhd P) : \mathsf{Set} \to \mathsf{Set} \\
&(S \rhd P)\,X = \Sigma s : S.\,P\,s \to X
\end{aligned}$$

So an element of $(S \rhd P)\,X$ is a pair $\langle s, xs\rangle$ where $s : S$ is a shape and $xs : P\,s \to X$ is a function assigning an element of X to every position in the shape s. The final coalgebra of a container, $\nu(S \rhd P)$, is inhabited by trees with nodes decorated by shapes and with positions giving their branching type:

$$\begin{array}{l}\mathsf{codata}\ \nu(S \rhd P) = \\ \quad \mathsf{in}_\nu\ \{\mathsf{shape} : S; \mathsf{subs} : P\,(\mathsf{shape}\,t) \to \nu(S \rhd P)\}\end{array}$$

So every element of $t : \nu(S \rhd P)$ is uniquely given by a shape, $\mathsf{shape}\,t : S$, and a family of subelements, $\mathsf{subs}\,t : P\,(\mathsf{shape}\,t) \to \nu(S \rhd P)$.

We are interested in characterizing the contractive functions between final coalgebras of containers: if $\langle S, P\rangle$ and $\langle T, Q\rangle$ are two containers, what are the contractions in $\nu(S \rhd P) \to \nu(T \rhd Q)$? We extend the intuition that we gained from streams and trees: a contraction produces the output structure up to depth n by looking only at the structure of the input at depths lower than n.

Ghani et al. (2009a) study the related question of characterizing the continuous functions of this same type. Their technique is useful for our purpose as well. They approximate the elements of the final coalgebra by another container $\langle S^\natural, P^\natural\rangle$, whose shapes are iterations of the functor up to a fixed depth and whose positions are the holes where new shapes can be inserted. We call the elements of $S^\natural$ *hangers* and the elements of $(P^\natural\ s)$ *pegs*, for some hanger s. Intuitively, a hanger is an incomplete structure, a well-founded approximation to a completed infinite tree. The pegs are those places in the incomplete structure where subtrees need to be inserted to complete the tree.

Hangers and pegs are defined by induction-recursion (Dybjer 2000; Dybjer and Setzer 1999, 2003). This is a type definition paradigm where we simultaneously define a well-founded type and a recursive function on it. When constructing an element, we can already use the function on its subterms. Induction-recursion is available in the dependently-typed language Agda and can be mimicked, for the small types that we consider, in other type-theoretic systems by an inductive family.

$$\begin{array}{ll} S^\natural : \mathsf{Set} & P^\natural : S^\natural \to \mathsf{Set} \\ \bullet : S^\natural & P^\natural\, \bullet = 1 \\ (;) : \Pi s : S^\natural.\, (P^\natural\ s \to S) \to S^\natural & \\ & P^\natural\ (s\ ;\sigma) = \Sigma p : P^\natural\ s.\, P\,(\sigma\, p) \end{array}$$

The simplest hanger, $\bullet$, is a completely uninformative approximation, a hook with one peg where the whole tree needs to be added. Given a hanger s with pegs $P^\natural\ s$, we can extend it by placing a new shape at each peg. So we give a function $\sigma : P^\natural\ s \to S$, which we think of as a new *slice* of the structure, specifying all the data at the next level. The new hanger is denoted by $(s\ ;\sigma)$ and its pegs are the disjoint union of the positions of all the new shapes.

We can approximate $\nu(S \rhd P)$ and $\nu(T \rhd Q)$ by stages using $\langle S^\natural, P^\natural\rangle$ and $\langle T^\natural, Q^\natural\rangle$. A contraction is a function for which the approximation of the output at a certain stage only depends on approximations of the input at lower stages. We can in fact summon again the intuition that we had for streams. Think of an element of $\nu(S \rhd P)$ as a stream of slices. Using stream notation, we can express it as

$$\begin{array}{l} \bullet \lhd \sigma_0 \lhd \sigma_1 \lhd \sigma_2 \lhd \cdots \\ \text{where } \ \sigma_0 : P^\natural\, \bullet \to S \\ \qquad\quad\ \ \sigma_1 : P^\natural\,(\bullet\,;\sigma_0) \to S \\ \qquad\quad\ \ \sigma_2 : P^\natural\,(\bullet\,;\sigma_0\,;\sigma_1) \to S. \end{array}$$

Most of our previous definitions and results are still valid, once we make the adjustments necessitated by the more complex type structure of the stream entries.

First of all, we can modify the family of equivalences up to depth n and use them to define the metric on the final coalgebra. We just give a function that truncates an element of the coalgebra to a hanger by cutting it at a given depth. We can cut a tree at level n into an upper part, given by a hanger, and a lower part, given by a family of trees to be inserted in the pegs.

$$\begin{array}{l} \mathsf{cut} : \nu(S \rhd P) \to \mathbb{N} \to \Sigma s : S^\natural.\, \left(P^\natural\ s \to \nu(S \rhd P)\right) \\ \mathsf{cut}\, t\, 0 = \langle \bullet, \lambda p.\, t\rangle \\ \mathsf{cut}\, t\,(n+1) = \ \ \mathsf{let}\ \langle s, \tau\rangle = \mathsf{cut}\, t\, n \\ \qquad\qquad\qquad\qquad\quad \sigma = \lambda p.\, \mathsf{shape}\,(\tau\, p) \\ \qquad\qquad\qquad\qquad\quad \tau' = \lambda p.\, \mathsf{subs}\,(\tau\, p) \\ \qquad\qquad\qquad\quad \mathsf{in}\ \langle s\ ;\sigma, \lambda\langle p, q\rangle.\, \tau'\, p\, q\rangle \end{array}$$

Keeping only the hanger part of this splitting (the first component of the pair) we get the truncation of a tree at level n.

$$\begin{array}{l} \mathsf{truncate} : \nu(S \rhd P) \to \mathbb{N} \to S^\natural \\ \mathsf{truncate}\, t = \mathsf{fst}\,(\mathsf{cut}\, t\, n) \end{array}$$

Using this notion, two elements of $\nu(S \rhd P)$ are then defined to be equivalent at level n if their n-truncations are the same:

$$t_1 =_n t_2 \quad \text{if and only if} \quad \mathsf{truncate}\, t_1\, n = \mathsf{truncate}\, t_2\, n$$

As in the case of trees, the definition of distance on $\nu(S \rhd P)$ is the same as the distance between streams and we get the same characterization of contractive functions.

Lemma 4 (contractive functions)**.** *A function $f : \nu(S \rhd P) \to \nu(T \rhd Q)$ is* contractive *if and only if $t_1 =_n t_2$ implies $f\, t_1 =_{n+1} f\, t_2$ for all natural numbers n and trees t_1 and t_2 in $\nu(S \rhd P)$.*

Example 2. *One of the most interesting applications of contractions on final coalgebras of containers is to realize the notion of higher-order recursion. For example, we may want to realize parametric fixed points on streams:*

$$\begin{array}{l} \mathsf{pfix} : (\mathsf{Stream}\,(A \times B) \to_c \mathsf{Stream}\, B) \\ \qquad\quad \to (\mathsf{Stream}\, A \to_c \mathsf{Stream}\, B) \\ \mathsf{pfix}\, f\, as = f\,(\mathsf{zip}\, as\,(\mathsf{pfix}\, f\, as)) \end{array}$$

In Section 4 we showed that contractive functions on streams can be represented by codes, so the parametric fixed point operator defined above can be lifted to the codes:

$$\begin{array}{l} \mathsf{pfix\_code} : \mathsf{Gen}\,(A \times B)\, B \to \mathsf{Gen}\, A\, B \\ \mathsf{pfix\_code}\,(\mathsf{Step}\, b\, g) = \mathsf{Step}\, b\,(\lambda a.\mathsf{pfix\_code}\,(g\,\langle a, b\rangle)) \end{array}$$

Furthermore, $\mathsf{Gen}\, A\, B$ *is a final coalgebra of a container with shapes B and positions $\lambda b.\, A$. Although* $\mathsf{pfix\_code}$ *is not itself a contraction, it is clear from its definition that it generates a slice for every slice of the input, so it preserves distances. This indicates that, when composed with contractions, it will yield a contraction.*

The representation theorem for contractions on final coalgebras of containers is complicated slightly by the fact that each slice of the structure has a different complex type. Contractions need to be defined *locally*, that is, given two hangers for the input and output, $s : S^\natural$ and $t : T^\natural$, we define the type of contractions from the extensions of s to the extension of t. In other words: assume that we have already read s in input and we have produced t in output, we define how the rest of the input is mapped to an extension of the output. We use the notation $\mathsf{CGen}\, s\, t$ for the set of codes for contractions from the *points* of s to the points of t. By *points of s* we mean elements of $\nu(S \rhd P)$ approximated by s.

A contraction must first of all produce part of the output without reading any input. The part of the output produced is a *slice* that puts a new shape in every peg: $\tau : Q^\natural\ t \to T$. Then the contraction reads a slice of the input $\sigma : P^\natural\, s \to S$ and, according to this value, specifies how to continue the computation by giving a new contraction between the refinements: $\mathsf{CGen}\,(s\ ;\sigma)\,(t\ ;\tau)$. Since both input and output are potentially infinite, the type of codes for contractions is also a final coalgebra defined by the following coinductive family, which generalizes the representations for streams in Section 4 and for trees in Section 7:

$$
\begin{array}{l}
\mathsf{codata} \\
\quad \mathsf{CGen} : S^{\natural} \to T^{\natural} \to \mathsf{Set} \\
\quad \mathsf{CGen}\, s\, t = \mathsf{Step}\, \{\ \mathsf{output} : Q^{\natural}\, t \to T; \\
\qquad\qquad\qquad\qquad \mathsf{cont} : \Pi\sigma : P^{\natural}\, s \to S. \\
\qquad\qquad\qquad\qquad\qquad \mathsf{CGen}\,(s\,;\sigma)\,(t\,;\mathsf{output})\ \}.
\end{array}
$$

Finally, the set of all contractions from $\nu(S \rhd P)$ to $\nu(T \rhd Q)$ is represented by $\mathsf{CGen}\ \bullet\ \bullet$.

We have seen earlier, in defining the function cut, that an element of $\nu(S \rhd P)$ can be split into a hanger $s : S^{\natural}$ and a family of substructures to be inserted in each peg of s. Let us call the set of all such possible families the *extension* of s: $\mathsf{Ext}\, s = P^{\natural}\, s \to \nu(S \rhd P)$. This type is isomorphic to the subtype of $\nu(S \rhd P)$ of those elements that are approximated by s.

We can widen the notion of contraction to functions between extensions. We write $\mathsf{Ext}\, s \to_c \mathsf{Ext}\, t$ to denote a contraction on the possible evolution of the input and output hangers, s and t. The definition is similar to that of contraction at the top level, except that we count depth from the next level below the hangers.

$$
\begin{array}{l}
\mathsf{genC}\, s\, t : \mathsf{CGen}\, s\, t \to \mathsf{Ext}\, s \to_c \mathsf{Ext}\, t \\
\mathsf{genC}\, s\, t\, (\mathsf{Step}\, \tau\, f)\, g = \lambda r : Q^{\natural}\, t. \\
\quad \mathsf{in}_{\nu}\, (\tau\, r)\, (\lambda q : Q\,(\tau\, r).\, \mathsf{genC}\,(s\,;\sigma)\,(t\,;\tau)\,(f\,\sigma) \\
\qquad\qquad\qquad\qquad\qquad (\lambda\langle p,u\rangle.\, \mathsf{subs}\,(g\, p)\, u)\ \langle r,q\rangle) \\
\quad \mathsf{where}\quad \sigma = \mathsf{shape} \circ g
\end{array}
$$

The above definition is rather involved, but the intuitive idea is similar to the special case of streams. A generating code has the form $(\mathsf{Step}\, \tau\, f)$, where τ is the slice that has to be sent to output immediately and f is the interaction function specifying how to continue the computation according to the value of the next input slice. Their respective types are

$$\tau : Q^{\natural}\, t \to T \qquad f : \Pi\sigma : P^{\natural}\, s \to S.\, \mathsf{CGen}\,(s\,;\sigma)\,(t\,;\tau)$$

So f reads a new input slice σ and decrees accordingly how to continue the computation between the two extended hangers.

The contractive function associated to this code maps the extension of s to the extension of t. The next argument is $g : \mathsf{Ext}\, s = P^{\natural}\, s \to \nu(S \rhd P)$. We need to produce an element in $\mathsf{Ext}\, t$, that is, $Q^{\natural}\, t \to \nu(T \rhd Q)$. The next argument to our function is then $r : Q^{\natural}\, t$ and we have to produce an element of $\nu(T \rhd Q)$. We use the canonical constructor in_{ν} for coinductive types: The top shape is given by the output slice in the appropriate positions, $(\tau\, r)$. The substructures must map every position $q : Q\,(\tau\, r)$ in this shape to an element of $\nu(T \rhd Q)$. Intuitively, we have produced a slice τ in output and we can read a new slice σ from input. We must now produce the part of the tree below the hanger $(t;\tau)$. We are allowed to use the next slice of the input to do this. The function g generates the whole continuation of the input. We extract just the first slice by taking only its top shapes:

$$\sigma = \mathsf{shape} \circ g : P^{\natural}\, s \to S$$

The function f applied to this slice produces a new code for a contraction between the extensions of $(s\,;\sigma)$ and $(t\,;\tau)$. We can recursively apply the generating function to this code:

$$\mathsf{genC}\,(s\,;\sigma)\,(t\,;\,\tau)\,(f\,\sigma) : \mathsf{Ext}\,(s\,;\sigma) \to_c \mathsf{Ext}\,(t\,;\,\tau)$$

This function takes an element of $\mathsf{Ext}\,(s\,;\sigma)$, whose structure can be seen by unfolding definitions as follows:

$$
\begin{array}{rl}
\mathsf{Ext}\,(s\,;\sigma) & = P^{\natural}\,(s\,;\sigma) \to \nu(S \rhd P) \\
& = (\Sigma p : P^{\natural}\, s.\, P\,(\sigma\, p)) \to \nu(S \rhd P)
\end{array}
$$

We already have an argument g in $\mathsf{Ext}\, s$, so we can just lop off the first slice: $\lambda\langle p,u\rangle.\, \mathsf{subs}\,(g\, p)\, u : \mathsf{Ext}\,(s\,;\sigma)$. Putting it all together, we have an element of $\mathsf{Ext}\,(t\,;\tau)$ and we can instantiate it to the right peg r and position q in the output tree.

In the other direction, we seek a representation operator that associates a code to every contractive function between the extensions of two hangers. As in previous incarnations, we need an arbitrary element $\mathsf{any}_s : \mathsf{Ext}\, s$. This will certainly exist if S is non-empty, that is, the input container has at least one shape. We assume this is the case in the following. The representation operator is defined as follows:

$$
\begin{array}{l}
\mathsf{repC}\, s\, t : (\mathsf{Ext}\, s \to_c \mathsf{Ext}\, t) \to \mathsf{CGen}\, s\, t \\
\mathsf{repC}\, s\, t\, \phi = \\
\quad \mathsf{Step}\, \tau\, (\lambda\sigma.\, \mathsf{repC}\,(s\,;\sigma)\,(t\,;\tau)\,(\lambda h.\, \lambda\langle r,q\rangle.\, \mathsf{subs}\,(\phi\, v\, r)\, q)) \\
\quad \mathsf{where}\ \ \tau = \mathsf{shape} \circ (\phi\, \mathsf{any}_s) \\
\qquad\qquad\ v = \lambda p.\, \mathsf{in}_{\nu}\,(\sigma\, p)\,(\lambda q.\, h\,\langle p,q\rangle)
\end{array}
$$

Remember that the function ϕ is assumed to be contractive, which means that the first slice it produces (which is the only part of $(\phi\, \mathsf{any}_s)$ that we need) does not actually depend on the argument any_s. The code for the contraction prescribes that the first slice of the output, τ, consists of the shapes of the result of ϕ on any_s (or indeed on any other element of $\mathsf{Ext}\, s$).

The continuation must be a function that maps the next slice of the input $\sigma : P^{\natural}\, s \to S$ to the code for the contraction on the extensions, with type $\mathsf{CGen}\,(s\,;\sigma)\,(t\,;\tau)$. Here we are allowed to use recursively the operator repC, because we are guarded by τ. We must give it a contraction between $\mathsf{Ext}\,(s\,;\sigma)$ and $\mathsf{Ext}\,(t\,;\tau)$. So let h be an extension of $(s\,;\sigma)$, that is:

$$
\begin{array}{rl}
h : \mathsf{Ext}\,(s\,;\sigma) & = P^{\natural}\,(s\,;\sigma) \to \nu(S \rhd P) \\
& = (\Sigma p : P^{\natural}\, s.\, P\,(\sigma\, p)) \to \nu(S \rhd P)
\end{array}
$$

First we use it to make an extension of s by simply gluing σ on top:

$$
\begin{array}{l}
g : \mathsf{Ext}\, s = P^{\natural}\, s \to \nu(S \rhd P) \\
g = \lambda p : P^{\natural}\, s.\, \mathsf{in}_{\nu}\,(\sigma\, p)\,(\lambda q : P\,(\sigma\, p).\, h\,\langle p,q\rangle)
\end{array}
$$

We can now apply the original contraction to this extension:

$$(\phi\, g) : \mathsf{Ext}\, t = Q^{\natural}\, t \to \nu(T \rhd Q)$$

We can split it into the first slice and the rest. Note that the first slice $\mathsf{shape} \circ (\phi\, g)$ must be equal to τ because f is a contraction. This is essential to check that the following type-checks correctly.

$$
\begin{array}{rl}
\lambda\langle r,q\rangle.\, \mathsf{subs}\,(\phi\, g\, r)\, q : & \mathsf{Ext}\,(t\,;\tau) \\
& = (\Sigma r : Q^{\natural}\, t.\, Q\,(\tau\, r)) \to \nu(T \rhd Q)
\end{array}
$$

This concludes the definition of the contraction between the extensions, therefore we can safely apply repC to it.

As in the case of streams and binary trees, the generation and representation operators are mutually inverse functions. The isomorphism is up to extensionality for functions and bisimilarity for coinductive objects. This means that we consider contractive functions and functional arguments of recursive data equal if they are equal pointwise. Elements of final coalgebras are considered equal if they are bisimilar. This allows us to use the method of proof by bisimulation: when proving that two structures are equal, we just have to show that the top shapes are equal and we can invoke the statement recursively on the substructures.

Theorem 4 (representation theorem)**.** *The functions* $\mathsf{genC}\, s\, t$ *and* $\mathsf{repC}\, s\, t$ *form an isomorphism* $\mathsf{CGen}\, s\, t \cong \mathsf{Ext}\, s \to_c \mathsf{Ext}\, t$.

At the top level, this gives us a representation isomorphism for contractions on final coalgebras:

$$\mathsf{CGen}\ \bullet\ \bullet \cong \nu(S \rhd P) \to_c \nu(T \rhd Q)$$

Proof. In one direction, given a contraction $\phi : \mathsf{Ext}\, s \to_c \mathsf{Ext}\, t$, we prove that $(\mathsf{genC}\, s\, t\, (\mathsf{repC}\, s\, t\, \phi)) = \phi$. Let us call ϕ' the left-hand side of this equality for short. We want to show that these two functions are extensionally equal. To this end, we let $g : \mathsf{Ext}\, s$ and $r : Q^\natural\, t$, and aim to prove that $\phi'\, g\, r = \phi\, g\, r$. These two terms are in the inductive type $\nu(T \triangleright Q)$, so their equality can be demonstrated by bisimulation: we prove that the top shape is the same and we invoke the statement of the theorem recursively to show that the substructures are also equal.

The top shapes are identical: $\mathsf{shape}\,(\phi'\, g\, r) = \mathsf{shape}\,(\phi\, g\, r)$. In fact, by construction, $\mathsf{shape}\,(\phi'\, g\, r) = \mathsf{shape}\,(\phi\, \mathsf{any}_s\, r)$. Continuity of ϕ guarantees that this result does not depend on the argument any_s, so replacing it with g gives the same shape, as desired.

The substructures are equal: $\mathsf{subs}\,(\phi'\, g\, r) = \mathsf{subs}\,(\phi\, g\, r)$. The coinduction principle, which allows us to prove equalities by bisimulation, tells us that we can recursively use the statement of the theorem to prove this. That is, we are allowed to assume that $\mathsf{genC}\,(s\,;\,\sigma)\,(t\,;\tau)$ and $\mathsf{repC}\,(s\,;\sigma)\,(t\,;\tau)$ are inverse of each other for appropriate σ and τ. We call this the *coinductive hypothesis*.

By definition of repC and genC we have that:

$$
\begin{aligned}
&\mathsf{subs}\,(\phi'\, g\, r) = \lambda v.\, \mathsf{genC}\,(s\,;\,\sigma)\,(t\,;\tau)\,(f\,\sigma)\, e\, \langle r, q\rangle \\
&\quad \text{where } \sigma = \mathsf{shape} \circ g \\
&\qquad\qquad \tau = \mathsf{shape} \circ (\phi\, \mathsf{any}_s) \\
&\qquad\qquad f = \lambda\sigma.\, \mathsf{repC}\,(s\,;\sigma)\,(t\,;\tau)\,\psi \\
&\qquad\qquad \psi = \lambda h.\, \lambda\langle r, q\rangle.\, \mathsf{subs}\,(f\, g'\, r)\, q \\
&\qquad\qquad g' = \lambda p.\, \mathsf{in}_\nu\,(\sigma\, p)\,(\lambda q.\, h\,\langle p, q\rangle) \\
&\qquad\qquad e = \lambda\langle p, u\rangle.\, \mathsf{subs}\,(g\, p)\, u
\end{aligned}
$$

We can now apply the coinduction hypothesis to obtain

$$
\begin{aligned}
&\mathsf{subs}\,(\phi'\, g\, r) \\
&= \lambda v.\, \mathsf{genC}\,(s\,;\,\sigma)\,(t\,;\tau)\,(\mathsf{repC}\,(s\,;\sigma)\,(t\,;\tau)\,\psi)\, e\,\langle r, q\rangle \\
&= \lambda v.\, \psi\, e\,\langle r, q\rangle \\
&= \mathsf{subs}\,(\phi\, g'\, r)\, q
\end{aligned}
$$

We can conclude by noting that $g' = g$ since

$$
\begin{aligned}
\sigma\, p &= \mathsf{shape}\,(g\, p) \\
e\,\langle p, q\rangle &= \mathsf{subs}\,(g\, p)\, q
\end{aligned}
$$

This completes the proof of one direction of the isomorphism. The opposite direction can be checked similarly, by just unfolding definitions and using extensional equality for functions and bisimulation for coinductive objects. □

Remark 2. *As for binary trees, we can generalize the construction and use any final coalgebra as codomain. For any functor G, we define a family of contractions from every hanger $s : S^\natural$ to νG:*

$$
\begin{aligned}
&\mathsf{codata}\ \ (\ \blacktriangleright G) : S^\natural \to \mathsf{Set} \\
&\qquad\quad \mathsf{Step} : G\,(\Pi\sigma : P^\natural\, s \to S.\,(s\,;\sigma) \blacktriangleright G) \to (s \blacktriangleright G)
\end{aligned}
$$

9. Instantiations for Streams and Trees

We show how the abstract representation of contractions on final coalgebras instantiates to the cases of streams and binary trees. What we obtain is equivalent to the ad hoc versions that we defined in Sections 4 and 7.

Streams can be represented as the final coalgebra of a container:

$$\mathsf{Stream}\, A \cong \nu(A \triangleright \lambda x.1)$$

In this case the type of hangers $A^\natural$ is just ($\mathsf{List}\ A$), and every hanger always has just one peg. Extension simply consists in attaching a new element at the end of a list. After simplification (the function type $1 \to A$ is isomorphic to A), the type of codes for contractions becomes

$$
\begin{aligned}
&\mathsf{codata} \\
&\quad \mathsf{CGen} : \mathsf{List}\ A \to \mathsf{List}\ B \to \mathsf{Set} \\
&\quad \mathsf{CGen}\, as\, bs \\
&\qquad = \mathsf{Step}\,\{\ \mathsf{output} : B; \\
&\qquad\qquad\qquad \mathsf{cont} : \Pi a : A.\mathsf{CGen}\,(as\,;a)\,(bs\,;\mathsf{output})\ \}
\end{aligned}
$$

Since the arguments as and bs occur only in the type specification, we have that each element of this family is isomorphic and essentially the same as $\mathsf{Gen}\, A\, B$.

The final coalgebra representation of infinite binary trees is

$$\mathsf{BTree}\, A \cong \nu(A \triangleright \lambda x.2)$$

The corresponding hangers are complete binary trees of fixed depth with elements of A in the internal nodes. Let us denote by $|s|$ the depth of such a hanger $s : A^\natural$. The pegs are the leaves of the tree s, therefore the hanger s will have $2^{|s|}$ pegs. We see this by simplifying the definitions in this particular case:

$$
\begin{array}{ll}
A^\natural : \mathsf{Set} & P^\natural : A^\natural \to \mathsf{Set} \\
\bullet : A^\natural & P^\natural\, \bullet = 1 \\
(;) : \Pi s : A^\natural.\, A^{2^{|s|}} \to A^\natural & P^\natural\,(s\,;\sigma) = \Sigma p : P^\natural\, s.2
\end{array}
$$

where we directly used the observation that $P^\natural\, s$ is a type with $2^{|s|}$ elements to define $A^\natural$ independently of $P^\natural$ ($P^\natural\, s \to A \cong A^{2^{|s|}}$). The type of codes for contractions simplifies to

$$
\begin{aligned}
&\mathsf{codata} \\
&\quad \mathsf{CGen} : A^\natural \to B^\natural \to \mathsf{Set} \\
&\quad \mathsf{CGen}\, s\, t \\
&\qquad = \mathsf{Step}\,\{\ \mathsf{output} : B^{2^{|t|}}; \\
&\qquad\qquad\qquad \mathsf{cont} : \Pi\sigma : A^{2^{|s|}}.\mathsf{CGen}\,(s\,;\sigma)\,(t\,;\mathsf{output})\ \}
\end{aligned}
$$

From this simplification, it is clear that $\mathsf{CGen}\, s\, t$ is isomorphic to $\mathsf{TGen}\, A^{2^{|s|}}\, B^{2^{|t|}}$.

10. Summary and Conclusion

In this article, we developed sound and complete representations of contractive functions on streams, non-well-founded binary trees, and final coalgebras of containers. In all three cases, a contraction is represented by a code. Such a code is itself an element of a coinductive type, and comprises two fields.

The first component, called output, gives the portion of the result that must be produced immediately, before reading any input. In the case of streams, it consists of the next element of the sequence; in the case of binary trees, it consists of the nodes on the next depth level; in the case of final coalgebras, it consists of the next *slice* of the structure.

The second component, called cont, specifies how the rest of the coinductive structure will be generated according to the value read in input. This input token is again the next element of the sequence for streams, a tuple of the nodes of the next depth level for trees, and the next slice of data for final coalgebras. The continuation is a function mapping this value to a recursive code.

We gave generation operators that unpack codes into contractive functions and representation mappings that synthesize a code from a contraction. We proved that the generation and representation operators are mutual inverses, showing that the representation is both sound and complete.

Our development yields a precise characterization of contractive functions on a wide class of coinductive data structures. This result provides the theoretical framework to deploy Banach's fixed point theorem to prove that recursive definitions of non-well-founded objects are guaranteed to produce a unique solution. We illustrated the application of our results by means of some simple examples. We expect to deploy them fruitfully on more complex and realistic applications in the future. In particular, they have the potential to facilitate the definition of highly recursive objects and to offer powerful proof methods for reasoning about them.

Acknowledgments

Graham Hutton was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

References

A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In Morrisett and Uustalu (2013), pages 185–196. ISBN 978-1-4503-2326-0.

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 27–38. ACM, 2013. ISBN 978-1-4503-1832-7.

M. Abott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

T. Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 8 – 21, 2001.

A. Arnold and M. Nivat. The metric space of infinite trees. algebraic and topological properties. *Fundam. Inform.*, 3(4):445–476, 1980.

R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In Morrisett and Uustalu (2013), pages 197–208. ISBN 978-1-4503-2326-0.

S. Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fund. Math.*, 3:133–181, 1922.

W. Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Ann. Pure Appl. Logic*, 136(1-2):75–90, 2005.

V. Capretta. Common knowledge as a coinductive modality. In E. Barendsen, H. Geuvers, V. Capretta, and M. Niqui, editors, *Reflections on Type Theory, Lambda Calculus, and the Mind*, pages 51–61. ICIS, Faculty of Science, Radbout University Nijmegen, 2007. Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday.

V. Capretta. Bisimulations generated from corecursive equations. *Electronic Notes in Theoretical Computer Science*, 265:245–258, 2010.

V. Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

C. Cîrstea, A. Kurz, D. Pattinson, L. Schröder, and Y. Venema. Modal logics are coalgebraic. *Comput. J.*, 54(1):31–41, 2011.

T. Coquand. Infinite objects in type theory. In H. Barendregt and T. 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.

E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of TLCA 1999*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, 1999.

P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Logic*, 124(1-3):1–47, 2003.

J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. *Theor. Comput. Sci.*, 411(4-5):765–782, 2010. doi: 10.1016/j.tcs.2009.10.014. URL `http://dx.doi.org/10.1016/j.tcs.2009.10.014`.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003. ISBN 3-540-22164-6.

N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 164(1):141–155, 2006. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006).

N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 249:3–18, 2009a. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

N. Ghani, P. Hancock, and D. Pattinson. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5 (3), 2009b.

P. D. Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In H. Geuvers and F. Wiedijk, editors, *Proc. TYPES'02*, volume 2646 of *LNCS*, pages 148–161. Springer-Verlag, 2003.

P. D. Gianantonio and M. Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In I. Walukiewicz, editor, *Proc. FOSSACS'04*, volume 2987 of *LNCS*, pages 136–150. Springer, 2004.

J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundam. Inform.*, 66(4):353–366, 2005.

J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, 1998.

E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

H. H. Hansen, D. Costa, and J. J. M. M. Rutten. Synthesis of mealy machines using derivatives. *Electr. Notes Theor. Comput. Sci.*, 164(1):27–45, 2006. doi: 10.1016/j.entcs.2006.06.003. URL `http://dx.doi.org/10.1016/j.entcs.2006.06.003`.

R. Hinze. Functional pearl: streams and unique fixed points. In J. Hook and P. Thiemann, editors, *ICFP 2008*, pages 189–200. ACM, 2008a. ISBN 978-1-59593-919-7.

R. Hinze. Concrete stream calculus: An extended study. *J. Funct. Program.*, 20(5-6):463–535, 2008b.

B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997a.

B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997b.

N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 257–266. IEEE Computer Society, 2011.

A. Kurz, A. Pardo, D. Petrisan, P. Severi, and F.-J. de Vries. Approximation of Nested Fixpoints – A Coalgebraic View of Parametric Dataypes. In L. S. Moss and P. Sobocinski, editors, *CALCO 2015*, volume 35 of *LIPIcs*, pages 205–220, Dagstuhl, Germany, 2015.

J. Lambek. A fixpoint theorem for complete categories. *Math. Zeitschr.*, 103:151–161, 1968.

A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP*, volume 8558 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 2014. ISBN 978-3-319-08969-0.

D. B. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In K. Kennedy, M. S. V. Deusen, and L. Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 165–174. ACM Press, 1984.

J. Matthews. Recursive function definition over coinductive types. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 1999. ISBN 3-540-66463-7.

E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

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.

G. Morrisett and T. Uustalu, editors. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, 2013. ACM. ISBN 978-1-4503-2326-0.

H. Nakano. A modality for recursion. In *LICS*, pages 255–266. IEEE Computer Society, 2000. ISBN 0-7695-0725-5. URL `http://tinyurl.com/huzq7gl`.

J. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15:93–147, 2005.

J. J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comput. Sci.*, 308 (1-3):1–53, 2003.

D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012a.

D. Sangiorgi, editor. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012b.

M. Smyth. Topology. In *Handbook of Logic in Computer Science*, volume 2, pages 641–761. Oxford University Press, 1992.