Under consideration for publication in the Journal of Functional Programming

Representing Contractive Functions on Streams

GRAHAM HUTTON
University of Nottingham, UK

MAURO JASKELIOFF
Universidad Nacional de Rosario, Argentina

Abstract

Streams, or infinite lists, have many applications in functional programming, and are naturally defined using recursive equations. But how do we ensure that such equations make sense, i.e. that they actually produce well-defined streams? In this article we present a new approach to this problem, based upon the topological notion of contractive functions on streams. In particular, we give a sound and complete representation theorem for contractive functions on streams, illustrate the use of this theorem as a practical means to produce well-defined streams, and show how the efficiency of the resulting definitions can be improved using another representation of contractive functions.

1 Introduction

Inductively defined types are a central concept in modern programming, with a wide variety of applications. In recent years, it has become increasingly clear that the dual notion of coinductive types are just as useful (Jacobs *et al.*, 2010). Examples of coinductive types include infinite lists, infinite trees, transition systems, abstract datatypes, and more generally, a wide variety of state-based dynamical systems (Jacobs & Rutten, 1997). In this article we focus our attention on infinite lists, known more concisely as *streams*.

In a functional language, streams are naturally defined using recursive equations. For example, if we write $x \lhd xs$ for the stream formed by prepending the value x to the stream xs, then the constant stream $1 \lhd 1 \lhd 1 \lhd \cdots$ can be defined by $ones = 1 \lhd ones$. But how do we ensure that such equations make sense, i.e. that they actually produce well-defined streams? For example, if we write *tail* for the function that removes the first value from a stream, then the equation $loop = tail\ loop$ is well-typed, but does not produce a well-defined stream because unfolding the equation loops forever without producing any values.

Most approaches to this problem are based upon some form of guardedness, a syntactic condition first proposed for use in process calculi by Milner (1989). In the case of stream equations, the most basic form of guardedness states that all recursive uses of the stream being defined must be guarded by the constructor $\lhd$, in the sense of occurring directly as its second argument. For example, the equation $ones = 1 \lhd ones$ is guarded, whereas the equation $loop = tail\ loop$ is not. While guardedness provides a simple syntactic means of ensuring that stream equations are well-defined, it also excludes many valid definitions. For example, if we write *merge* for the function that selects alternative values from two

streams, then the stream of ones can also be defined by $ones = 1 \lhd merge\ ones\ (tail\ ones)$, but this equation fails to satisfy the guardedness condition.

In this article we present a new approach to the problem of ensuring that recursive stream equations are well-defined, based upon the topological notion of contractive functions on streams. More precisely, the article makes the following contributions:

- We give an accessible presentation and proof of Banach's fixed point theorem for contractive functions on streams, which provides a simple but powerful semantic means of ensuring that stream equations are well-defined (section 4).
- We present a sound and complete representation theorem for contractive functions on streams, which formalises such functions as precisely those whose output at any point in time only depends on inputs at strictly earlier times (section 5).
- We illustrate the use of this theorem as a practical means to produce well-defined streams (sections 6 and 7), and show how the efficiency of the resulting definitions can be improved using another representation theorem (sections 8 and 9).

The article is aimed at readers who are familiar with the basics of functional programming with streams, say to the level of chapter twelve of (Hutton, 2007), but no previous experience with topological methods is assumed. The techniques are presented using a Haskell-like syntax for a total functional language in which types are sets and programs are total functions between sets, in the manner of (Turner, 1995). An extended version of the article that includes all the proofs is available from the authors' web pages.

2 Streams

In this section we review the idea of programming with streams, and introduce our notation. We write *Stream a* for the type of streams of values of type *a*, which are constructed using an infix operator $\lhd$ that prepends a value to a stream of the same type, and are destructed using functions *head* and *tail* that select and remove the first value from a stream:

$$\begin{array}{lcl} head & :: & Stream\ a \to a \\ head\ (x \lhd xs) & = & x \\ tail & :: & Stream\ a \to Stream\ a \\ tail\ (x \lhd xs) & = & xs \end{array}$$

Streams themselves are naturally defined using recursive equations. For example, if we write *Nat* for the type of natural numbers then the constant stream $1 \lhd 1 \lhd 1 \lhd \cdots$ can be defined as a single one followed by the stream itself:

$$\begin{array}{lcl} ones & :: & Stream\ Nat \\ ones & = & 1 \lhd ones \end{array}$$

In turn, the stream of natural numbers can be defined by starting with zero, and then mapping the successor function $(+1)$ over each element of the stream itself:

$$\begin{array}{lcl} nats & :: & Stream\ Nat \\ nats & = & 0 \lhd map\ (+1)\ nats \\ map & :: & (a \to b) \to Stream\ a \to Stream\ b \\ map\ f\ (x \lhd xs) & = & f\ x \lhd map\ f\ xs \end{array}$$

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

$$\begin{array}{lcl} \mathit{loop} & :: & \mathit{Stream}\ a \\ \mathit{loop} & = & \mathit{tail}\ \mathit{loop} \end{array}$$

but does not actually define a stream because unfolding the definition loops forever without ever producing any values. Similarly, attempting to redefine

$$\begin{array}{lcl} \mathit{ones} & :: & \mathit{Stream}\ \mathit{Nat} \\ \mathit{ones} & = & 1 \lhd \mathit{tail}\ \mathit{ones} \end{array}$$

is also invalid, because it produces a single one and then loops. On the other hand, not all uses of *tail* are problematic. For example, *ones* can be defined as a single one followed by the result of merging alternative elements from the stream itself and its tail:

$$\begin{array}{lcl} \mathit{ones} & :: & \mathit{Stream}\ \mathit{Nat} \\ \mathit{ones} & = & 1 \lhd \mathit{merge}\ \mathit{ones}\ (\mathit{tail}\ \mathit{ones}) \\ \mathit{merge} & :: & \mathit{Stream}\ a \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\ \mathit{merge}\ (x \lhd \mathit{xs})\ \mathit{ys} & = & x \lhd \mathit{merge}\ \mathit{ys}\ \mathit{xs} \end{array}$$

However, if we swapped the order of the arguments to *merge* in the above definition for *ones*, the definition again becomes invalid. This brings us to the central question of this article: when does a recursive stream equation actually define a stream? In the next few sections we introduce the technical machinery that underlies our approach.

3 Fixed points

In the previous section we informally 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 *Nat* 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 injection functions *inl* and *inr*. The right-to-left component of the bijection, $f :: 1 + \mathit{Nat} \to \mathit{Nat}$, gives rise to the constructors for naturals by defining $\mathit{Zero} = f\ (\mathit{inl}\ *)$ and $\mathit{Succ}\ n = f\ (\mathit{inr}\ n)$. In turn, the left-to-right component $g :: \mathit{Nat} \to 1 + \mathit{Nat}$ gives a form of case analysis, mapping *Zero* to $\mathit{inl}\ *$ and $\mathit{Succ}\ n$ to $\mathit{inr}\ n$.

Dually, coinductive types are defined as the *greatest* solution of some equation. For example, the equation $X \cong 1 + X$ also has a greatest solution, given by the type Nat^∞ of natural numbers together with an infinite value defined by $\mathit{inf} = \mathit{Succ}\ \mathit{inf}$. In a similar manner, the coinductive type $\mathit{Stream}\ A$ 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 projection functions *fst* and *snd*. The left-to-right component of the bijection, $f :: \mathit{Stream}\ A \to A \times \mathit{Stream}\ A$, gives rise to the deconstructors for streams by defining $\mathit{head}\ \mathit{xs} = \mathit{fst}\ (f\ \mathit{xs})$ and $\mathit{tail}\ \mathit{xs} = \mathit{snd}\ (f\ \mathit{xs})$. In turn, the right-to-left component $g :: A \times \mathit{Stream}\ A \to \mathit{Stream}\ A$ gives rise to the constructor for streams by defining $x \lhd \mathit{xs} = g\ (x, \mathit{xs})$.

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 called 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, the function $map\ (+1)$ has no fixed point, which corresponds to the fact that the equation $xs = map\ (+1)\ xs$ is not a valid definition for a stream. 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. Note that there is no general notion of ordering on streams, so it doesn't 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 $fix\ f$, which is adopted as the semantics of the corresponding recursive equation. For example, the function $(1 \lhd)$ has a unique fixed point given by the constant stream of ones, which corresponds to the fact that the equation $ones = 1 \lhd ones$ is a valid definition. In conclusion, the question of when a recursive stream equation defines a stream can now be rephrased as follows: when does a function $f :: Stream\ a \to Stream\ a$ have a unique fixed point $fix\ f :: Stream\ a$?

4 Contractive functions

Our approach to this question is based upon an idea from topology, in the form of contractive functions. In order to define this notion for streams, we first define a function *take* that returns the finite list comprising the first n elements of a stream:

$$\begin{array}{lcl} take & :: & Nat \to Stream\ a \to [a] \\ take\ 0\ xs & = & [\,] \\ take\ (n+1)\ (x \lhd xs) & = & x : take\ n\ xs \end{array}$$

For example, $take\ 3\ ones = [1,1,1]$. In turn, we will use the notation $xs =_n ys$ when two streams are equal for their first n values. More formally, we define a family of equivalence relations $=_n$ on streams of the same type as follows:

$$xs =_n ys \;\Leftrightarrow\; take\ n\ xs = take\ n\ ys$$

For the purposes of proofs, however, we will usually find it more convenient to use the following definition by explicit recursion:

$$\begin{array}{rcl} xs =_0 ys & \Leftrightarrow & True \\ (x \lhd xs) =_{n+1} (y \lhd ys) & \Leftrightarrow & x = y \wedge xs =_n ys \end{array}$$

The notion of contractivity can now be defined as follows:

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

That is, a function on streams is contractive if whenever two streams are equal for their first n elements, applying the function gives streams that are equal for $n+1$ elements. We denote the type of contractive functions by $Stream\ a \to_c Stream\ b$. The key property of such functions is captured in the followed fixed point theorem (Banach, 1922):

Theorem 1 (*Banach's theorem*)
Every contractive function $f :: \mathit{Stream}\ a \to_c \mathit{Stream}\ a$ has a unique fixed point.

In order to prove this result we first show that there can only be at most one fixed point (uniqueness), and then that there is at least one fixed point (existence).

Proof (uniqueness). Suppose that a contractive function f has two fixed points xs and ys, i.e. $f\ xs = xs$ and $f\ ys = ys$. We are then required to show that $xs = ys$. Using the *take lemma* (Bird & Wadler, 1988), $xs = ys$ is equivalent to showing that $xs =_n ys$ for all natural numbers n, which property can then be verified by induction on n. The base case is trivial, because $xs =_0 ys$ is always true. For the inductive case, we calculate as follows:

$$
\begin{array}{ll}
 & xs =_{n+1} ys \\
\Leftrightarrow & \quad \{\ xs \text{ and } ys \text{ are fixed points of } f\ \} \\
 & f\ xs =_{n+1} f\ ys \\
\Leftarrow & \quad \{\ f \text{ is contractive}\ \} \\
 & xs =_n ys \\
\Leftarrow & \quad \{\ \text{induction hypothesis}\ \} \\
 & \mathit{True}
\end{array}
$$

Note that in the second step, contractivity is precisely the condition that is required to allow the induction hypothesis to be used to complete the proof. □

Proof (existence). We begin our construction by writing any_a for an arbitrary stream of type a, and defining a family S_n of streams indexed by natural numbers:

$$
\begin{array}{lcl}
S_0 & = & any_a \\
S_{n+1} & = & f\ S_n
\end{array}
$$

That is, the stream S_n is given by the n-fold application of the function f to an arbitrary stream; for example, $S_3 = f\ (f\ (f\ any_a))$. Note that existence of a stream any_a requires that the type a is non-empty, otherwise there are no streams of this type, but does not require the axiom of choice because once we have a witness for the type a being non-empty this value can simply be replicated to produce a constant stream.

The intuition behind the above definition is that the first n elements of the stream S_n will form the first n elements of a fixed point of f. To formalise this idea, we first note that the family S_n forms a chain-like structure in the sense that $S_n =_n S_{n+1}$, i.e. the first n elements of the stream S_n coincide with the first n elements of the next stream S_{n+1}. Now let us define an indexing operator !! that selects the nth element in a stream:

$$
\begin{array}{lcl}
(!!) & :: & \mathit{Stream}\ a \to \mathit{Nat} \to a \\
(x \lhd xs)\,!!\,0 & = & x \\
(x \lhd xs)\,!!\,(n+1) & = & xs\,!!\,n
\end{array}
$$

Using this operator, together with the fact that a stream can be defined by specifying its value for any index using the isomorphism $\mathit{Stream}\ a \cong \mathit{Nat} \to a$ (Altenkirch, 2001), we then define a stream s whose nth element is given by the nth element of S_{n+1}:

$$
s\,!!\,n \quad = \quad S_{n+1}\,!!\,n
$$

This definition ensures that $s =_n S_n$, i.e. the first n elements of the streams s and S_n coincide, using which property it is now straightforward to show that the stream s is a fixed point of the function f, thereby establishing the existence of a fixed point:

$$
\begin{array}{ll}
 & f\ s = s \\
\Leftrightarrow & \quad \{ \text{ take lemma } \} \\
 & \forall\, n\,.\, f\ s =_n s \\
\Leftrightarrow & \quad \{ \ xs =_0 ys \text{ is always true } \} \\
 & \forall\, n\,.\, f\ s =_{n+1} s \\
\Leftarrow & \quad \{ \text{ above property, } =_n \text{ is transitive } \} \\
 & \forall\, n\,.\, f\ s =_{n+1} S_{n+1} \\
\Leftrightarrow & \quad \{ \text{ applying } S \} \\
 & \forall\, n\,.\, f\ s =_{n+1} f\ (S_n) \\
\Leftarrow & \quad \{ f \text{ is a contractive } \} \\
 & \forall\, n\,.\, s =_n S_n \\
\Leftrightarrow & \quad \{ \text{ above property } \} \\
 & \mathit{True}
\end{array}
$$

This completes the proof of Banach's fixed point theorem for streams. □

In summary, contractivity provides a sufficient condition for producing well-defined streams, because contractive functions have precisely one fixed point. For example, a simple calculation shows that the function $(1 \lhd)$ is contractive and hence has a unique fixed point, which ensures that $ones = fix\ (1 \lhd)$ is a valid definition for the constant stream of ones. Similarly, the function $f = (0 \lhd) \circ map\ (+1)$ is also contractive, which ensures that $nats = fix\ f$ is a valid definition for the stream of natural numbers.

In contrast, the function *tail* is not contractive. For example, in the case of $n = 0$ the contractivity property reduces to $head\ (tail\ xs) = head\ (tail\ ys)$, which is not true in general because two arbitrary streams *xs* and *ys* of type *a* need not have the same second element, except in the trivial cases where the type *a* is empty or contains a single possible value. Hence, Banach's fixed point theorem does not apply to the function *tail*, and we reject the equation $loop = fix\ tail$ as being an invalid definition of a stream.

Contractivity ensures that functions have a unique fixed point, but raises some important questions. First of all, does the converse theorem also hold: is every function on streams with a unique fixed point contractive? The answer is no. For example, if we define

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

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 $head\ (tail\ xs) = head\ (tail\ ys)$, which as we have just seen is not always true.

And secondly, it is natural to ask what the notion of 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.

5 Representation theorem

Suppose that we view a stream as a value that may vary over time, where time is given by a natural number index into the stream using the isomorphism $\mathit{Stream}\ a \cong \mathit{Nat} \to a$. Then contractive functions on streams are precisely those functions whose output value at any time only depends on input values at *strictly earlier* times. For example, the output at time 3 can only depend on the inputs at times 0, 1 and 2, and no other times. In order to formalise this idea, we introduce the following notion:

Definition 2 (*generating functions*)
A *generating function* is a function of type $[a] \to b$.

That is, a generating function maps a finite list of values to a single value. The intuition is that the argument list records all earlier inputs for a contractive function (its *input history*), and the result value is the next output of this function. We denote the type $[a] \to b$ of generating functions by *Gen a b*. It is now straightforward to convert a generating function into the corresponding contractive function on streams:

$$\begin{array}{lcl} \mathit{gen} & :: & \mathit{Gen}\ a\ b \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{gen}\ g\ (x \lhd xs) & = & g\ [\,] \lhd \mathit{gen}\ (g \circ (x{:}))\ xs \end{array}$$

This definition expresses that the output value at any time n (counting from zero) is obtained by applying the generating function to the first n values of the input stream. For example, applying *gen g* to the stream $0 \lhd 1 \lhd 2 \lhd \cdots$ gives the output stream:

$$g\,[\,] \lhd g\,[0] \lhd g\,[0,1] \lhd g\,[0,1,2] \lhd \cdots$$

The validity of the definition of *gen*, which requires that the resulting function on streams is contractive, is established by the following result.

Lemma 1
If g is a generating function then *gen g* is contractive.

Conversely, every contractive function can be represented as a generating function:

$$\begin{array}{lcl} \mathit{rep} & :: & (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \to \mathit{Gen}\ a\ b \\ \mathit{rep}\ f\ [\,] & = & \mathit{head}\ (f\ \mathit{any}_a) \\ \mathit{rep}\ f\ (x{:}xs) & = & \mathit{rep}\ (\mathit{tail} \circ f \circ (x \lhd))\ xs \end{array}$$

Given an input history *xs* of length n, this definition expresses that the next output produced by *rep f* will be the nth value in the result of applying f to an argument stream that begins with *xs*. Contractivity ensures that the result is independent of the remainder of the argument stream, for which purposes we utilise an arbitrary stream in the base case. For example, applying *rep f* to the input history $[0,1,2]$ gives the output value:

$$f\ (0 \lhd 1 \lhd 2 \lhd \mathit{any})\ !!\ 3$$

The validity of the recursive call in the definition of *rep*, which requires that its argument function is contractive, is established by the following result.

Lemma 2
If f is contractive then $\mathit{tail} \circ f \circ (x \lhd)$ is contractive.

Using the two conversion functions, we can now formalise the idea that contractive and generating functions are in one-to-one correspondence, i.e. every contractive function can be uniquely represented by a generating function, and vice-versa.

Theorem 2 (*representation theorem*)
The functions *gen* and *rep* form an isomorphism $Gen\ a\ b \cong Stream\ a \rightarrow_c Stream\ b$.

Proof: we verify the two parts of the isomorphism separately. For the first part, $rep \circ gen = id$, we are required to show that $rep\ (gen\ g)\ xs = g\ xs$ for any generating function g and finite list xs, which can be verified by induction on xs.

Base case:

$$\begin{array}{ll} & rep\ (gen\ g)\ [\,] \\ = & \quad \{ \text{ applying } rep \} \\ & head\ (gen\ g\ any_a) \\ = & \quad \{ \text{ applying } gen \text{ and } head \} \\ & g\ [\,] \end{array}$$

Inductive case:

$$\begin{array}{ll} & rep\ (gen\ g)\ (x : xs) \\ = & \quad \{ \text{ applying } rep \} \\ & rep\ (tail \circ gen\ g \circ (x \lhd))\ xs \\ = & \quad \{ \text{ applying } \circ \} \\ & rep\ (\lambda ys \rightarrow tail\ (gen\ g\ (x \lhd ys)))\ xs \\ = & \quad \{ \text{ applying } gen \text{ and } tail \} \\ & rep\ (\lambda ys \rightarrow gen\ (g \circ (x:))\ ys)\ xs \\ = & \quad \{ \text{ eta reduction } \} \\ & rep\ (gen\ (g \circ (x:)))\ xs \\ = & \quad \{ \text{ induction hypothesis } \} \\ & (g \circ (x:))\ xs \\ = & \quad \{ \text{ applying } \circ \} \\ & g\ (x : xs) \end{array}$$

For the second part, $gen \circ rep = id$, we are required to show that $gen\ (rep\ f)\ xs = f\ xs$ for any contractive function f and stream xs, which can be verified by (guarded) coinduction (Turner, 1995) on xs, which we assume has the form $xs = x \lhd xs'$:

$$\begin{array}{ll} & gen\ (rep\ f)\ (x \lhd xs') \\ = & \quad \{ \text{ applying } gen \} \\ & rep\ f\ [\,] \lhd gen\ (rep\ f \circ (x:))\ xs' \\ = & \quad \{ \text{ applying } rep \} \\ & head\ (f\ any_a) \lhd gen\ (rep\ (tail \circ f \circ (x \lhd)))\ xs' \\ = & \quad \{ \text{ coinduction hypothesis } \} \\ & head\ (f\ any_a) \lhd (tail \circ f \circ (x \lhd))\ xs' \\ = & \quad \{ \text{ applying } \circ \} \\ & head\ (f\ any_a) \lhd tail\ (f\ (x \lhd xs')) \\ = & \quad \{ xs = x \lhd xs' \} \end{array}$$

$$
\begin{array}{ll}
 & \mathit{head}\ (f\ \mathit{any}_a) \lhd \mathit{tail}\ (f\ \mathit{xs}) \\
= & \quad \{\ \text{Lemma 3, } f \text{ is contractive}\ \} \\
 & \mathit{head}\ (f\ \mathit{xs}) \lhd \mathit{tail}\ (f\ \mathit{xs}) \\
= & \quad \{\ \text{streams}\ \} \\
 & f\ \mathit{xs}
\end{array}
$$

□

The lemma used in the second calculation above states that the first output value produced by a contractive function does not depend on the input stream.

Lemma 3
If f is contractive then $\mathit{head}\ (f\ \mathit{xs}) = \mathit{head}\ (f\ \mathit{ys})$ for any streams *xs* and *ys*.

6 Practical applications

Our representation theorem for contractive functions on streams has two main practical applications. First of all, it simplifies the process of deciding if a function is contractive. For example, it is easy to see that the function $(1 \lhd)$ is contractive, because the first output (the value 1) requires no inputs, and each subsequent output only depends on the input at one step earlier in time. That is, $(1 \lhd)$ acts as a one-step delay function. Conversely, the function *tail* is not contractive, because the output at any time depends on the input at one step later in time. That is, *tail* acts as a one-step lookahead function.

Secondly, our theorem provides a practical means of producing streams that are guaranteed to be well-defined. In particular, because contractive functions are in one-to-one correspondence with generating functions, rather than defining a stream directly as the fixed point of a contractive function (which requires verifying that the function is indeed contractive) we can simply provide a generating function, from which the corresponding function on streams (which is guaranteed to be contractive and hence have a unique fixed point) can be generated automatically by applying the conversion function *gen* from the previous section. We encapsulate this idea by defining a new fixed point operator:

$$
\begin{array}{lcl}
\mathit{gfix} & :: & \mathit{Gen}\ a\ a \to \mathit{Stream}\ a \\
\mathit{gfix}\ g & = & \mathit{fix}\ (\mathit{gen}\ g)
\end{array}
$$

For example, the constant stream of ones can be defined using *gfix* as follows (where the function *last* returns the last element of a non-empty list):

$$
\begin{array}{lcl}
\mathit{ones} & :: & \mathit{Stream}\ \mathit{Nat} \\
\mathit{ones} & = & \mathit{gfix}\ \mathit{gones} \\
\mathit{gones} & :: & \mathit{Gen}\ \mathit{Nat}\ \mathit{Nat} \\
\mathit{gones}\ [\,] & = & 1 \\
\mathit{gones}\ \mathit{xs} & = & \mathit{last}\ \mathit{xs}
\end{array}
$$

In this example, the generating function *gones* expresses that the first value in the stream (when the history list of previous values is empty) is one, and each subsequent value is the same as the last, i.e. most recent, value in the history. Alternatively, we could simply state

that the next value in the stream is always the constant value one:

$$\begin{array}{lcl} \mathit{gones} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gones\ xs} & = & 1 \end{array}$$

Note that the two definitions for *gones* above give rise to different contractive functions under the conversion function *gen*; it's only their fixed points that are equal.

Similarly, the natural numbers can be defined using a generating function that expresses that the first value is zero, and each subsequent value is the successor of the last:

$$\begin{array}{lcl} \mathit{nats} & :: & \mathit{Stream\ Nat} \\ \mathit{nats} & = & \mathit{gfix\ gnats} \\ \mathit{gnats} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gnats}\ [\,] & = & 0 \\ \mathit{gnats\ xs} & = & \mathit{last\ xs} + 1 \end{array}$$

Alternatively, the next value is also given by the length of the history:

$$\begin{array}{lcl} \mathit{gnats} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gnats\ xs} & = & \mathit{length\ xs} \end{array}$$

Now consider the stream of Fibonacci numbers $0 \lhd 1 \lhd 1 \lhd 2 \lhd 3 \lhd 5 \lhd \cdots$, in which the first two values are zero and one, and each subsequent value is the sum of the preceding two values. This behaviour translates directly into the following definition (where the function *penu* returns the penultimate, i.e. next to last, element of a list):

$$\begin{array}{lcl} \mathit{fibs} & :: & \mathit{Stream\ Nat} \\ \mathit{fibs} & = & \mathit{gfix\ gfibs} \\ \mathit{gfibs} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gfibs}\ [\,] & = & 0 \\ \mathit{gfibs}\ [x] & = & 1 \\ \mathit{gfibs\ xs} & = & \mathit{penu\ xs} + \mathit{last\ xs} \end{array}$$

As with the stream of natural numbers, the generating function for *fibs* can also be defined in terms of the length of the history (where the indexing operator !! selects the nth element of a list, in a similar manner to corresponding operator for streams):

$$\begin{array}{lcl} \mathit{gfibs} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gfibs\ xs} & = & \textbf{case}\ \mathit{length\ xs}\ \textbf{of} \\ & & \quad 0 \rightarrow 0 \\ & & \quad 1 \rightarrow 1 \\ & & \quad n \rightarrow \mathit{xs}\,!!\,(n-2) + \mathit{xs}\,!!\,(n-1) \end{array}$$

For our final example, let us return to the other definition for the constant stream of ones in the introductory section, which defines the stream as a single one followed by the result of merging alternative elements from the stream itself and its tail:

$$\begin{array}{lcl} \mathit{ones} & :: & \mathit{Stream\ Nat} \\ \mathit{ones} & = & 1 \lhd \mathit{merge\ ones\ (tail\ ones)} \end{array}$$

Unlike the previous examples, it may not be immediately obvious how this behaviour can be captured using a generating function. However, if we expand out the behaviour of *merge ones* (*tail ones*) using indexing, we obtain the following stream:

$$\mathit{ones}\,!!\,0 \;\triangleleft\; \mathit{ones}\,!!\,1 \;\triangleleft\; \mathit{ones}\,!!\,1 \;\triangleleft\; \mathit{ones}\,!!\,2 \;\triangleleft\; \mathit{ones}\,!!\,2 \;\triangleleft\; \cdots$$

From this expansion it is now evident that the behaviour of the *merge* can be obtained by indexing using a *slowed down* version of the natural numbers that only increments the number every other step $(0,1,1,2,2,\cdots)$, which corresponds to the fact that the rate at which values are consumed from each of the input streams to *merge* is half the rate at which values are produced to the output stream. Hence, the above definition for *ones* translates into using a generating function that indexes into the history at half its length (where the division operator $\div$ rounds down to the nearest natural number):

$$\begin{array}{lcl} \mathit{ones} & :: & \mathit{Stream\ Nat} \\ \mathit{ones} & = & \mathit{gfix\ gones} \\ \mathit{gones} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{gones}\,[\,] & = & 1 \\ \mathit{gones\ xs} & = & \mathit{xs}\,!!\,(\mathit{length\ xs} \div 2) \end{array}$$

In conclusion, many recursive stream equations (such as those for *ones*, *nats* and *fibs*) have a simple and natural definition in terms of generating functions, while some stream equations (such as the alternative definition of *ones* using *merge*) may require some thought to express the desired behaviour in terms of the history of previous values of the stream. However, defining streams in this manner has the important benefit that there is no longer any need to check that the resulting stream is well-defined, because this is achieved *by construction* using our representation theorem. Moreover, because our theorem establishes an isomorphism between generating functions and contractive functions, we know that *any* stream that arises as the unique fixed point of a contractive function can be defined in this manner, and hence that the technique is generally applicable.

7 Improving efficiency

We now turn our attention to the issue of efficiency. Using the Glasgow Haskell Compiler, some simple experiments show that the standard definitions for *ones*, *nats* and *fibs* using recursive equations take linear time to produce their first *n* values, as we would expect. In contrast, our new definitions using generating functions take quadratic time, which is unsatisfactory. There are two reasons for this behaviour. First of all, the function *gen* that converts generating functions into contractive functions maintains the entire input history, even though only a small part of this may be required to produce the next output value. For example, in the case of *fibs* we only require the last two values. And secondly, it takes linear time to access the most recent values in the history list, due to the fact that it is maintained in time order, i.e. with the most recent value at the end of the list.

The second problem can easily be solved by maintaining the input history in reverse time order, i.e. with the most recent value at the start. In order to achieve this, we begin by replacing the term $(x{:})$ that prefixes a value to the history in the definition of the conversion

function *gen* by the term $(+\!+\,[x])$ that postfixes a value:

$$\begin{array}{lcl} \mathit{rgen} & :: & \mathit{Gen}\ a\ b \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{rgen}\ g\ (x \lhd xs) & = & g\ [\,] \lhd (g \circ (+\!+\,[x]))\ xs \end{array}$$

For example, applying *rgen g* to input $0 \lhd 1 \lhd 2 \lhd \cdots$ gives the output:

$$g\ [\,] \lhd g\ [0] \lhd g\ [1,0] \lhd g\ [2,1,0] \lhd \cdots$$

In turn, we then eliminate the use of the inefficient append operator $+\!+$ in the above definition by using an extra argument to accumulate the history:

$$\begin{array}{lcl} \mathit{rgen} & :: & \mathit{Gen}\ a\ b \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{rgen}\ g & = & \mathit{rgen}'\ g\ [\,] \\ \mathit{rgen}' & :: & \mathit{Gen}\ a\ b \to [a] \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{rgen}'\ g\ ys\ (x \lhd xs) & = & g\ ys \lhd \mathit{rgen}'\ g\ (x\!:\!ys)\ xs \end{array}$$

It is now straightforward to redefine generating functions to take their input history in reverse order. For example, the generating function for *fibs* now has direct access to the two most recent values in the input history by simple pattern matching:

$$\begin{array}{lcl} \mathit{rgfibs} & :: & \mathit{Gen\ Nat\ Nat} \\ \mathit{rgfibs}\ [\,] & = & 0 \\ \mathit{rgfibs}\ [x] & = & 1 \\ \mathit{rgfibs}\ (x\!:\!y\!:\!zs) & = & y+x \end{array}$$

Using this approach, the definitions for the streams *ones*, *nats* and *fibs* in terms of generating functions now take linear time, with constant factors similar to the original recursive definitions. It is natural at this point to wonder why we didn't just use a reversed history from the outset? The answer is that using the time ordered history gives a simpler proof of our representation theorem for contractive functions.

While we have now regained the desired time performance, the space problem that we raised at the start of this section still remains, namely that the entire input history is not usually required to produce the next output value. However, it turns out that both efficiency problems can be solved together in a simple manner using another representation of contractive functions, this time in terms of infinite trees.

8 Generating trees

Let us write *Tree a b* for the type of *a*-branching infinite trees with *b*-labels in the nodes, which are built using a prefix constructor *Node* that forms a tree from a label of type *b* and a branching function of type $a \to \mathit{Tree}\ a\ b$, and are destructed using functions *label* and *branches* that select the first and second components of a node:

$$\begin{array}{lcl} \mathit{label} & :: & \mathit{Tree}\ a\ b \to b \\ \mathit{label}\ (\mathit{Node}\ y\ f) & = & y \\ \mathit{branches} & :: & \mathit{Tree}\ a\ b \to (a \to \mathit{Tree}\ a\ b) \\ \mathit{branches}\ (\mathit{Node}\ y\ f) & = & f \end{array}$$

More formally, in set-theoretic terms the coinductive type *Tree A B* can be defined as the greatest set X for which there is a bijection $X \cong B \times (A \to X)$. We will refer to values of type *Tree a b* as *generating trees*, because every such tree uniquely represents a generating function. In order to formalise this idea, we first define a function that converts a generating tree into the corresponding generating function:

$$\begin{array}{lcl} gen' & :: & Tree\ a\ b \to Gen\ a\ b \\ gen'\ (Node\ y\ f)\ [\,] & = & y \\ gen'\ (Node\ y\ f)\ (x{:}xs) & = & gen'\ (f\ x)\ xs \end{array}$$

That is, the first output produced by the generating function (when its input history is empty) is given by the label y of the generating tree, and subsequent outputs are given by applying the branching function f to the first value x in the input history to obtain a new tree that is used to process the remainder of the history xs in the same manner.

Conversely, every generating function can be represented as a generating tree:

$$\begin{array}{lcl} rep' & :: & Gen\ a\ b \to Tree\ a\ b \\ rep'\ g & = & Node\ (g\ [\,])\ (\lambda x \to rep'\ (g \circ (x{:}))) \end{array}$$

This definition expresses that the label at each level in the resulting tree is given by applying the generating function g to the finite list of all previous branching values in the tree. For example, the first few levels of the tree $rep'\ g$ are as follows:

$$\begin{array}{ll} Node\ (g\ [\,]) & (\lambda x \to \\ Node\ (g\ [x]) & (\lambda y \to \\ Node\ (g\ [x,y]) & (\lambda z \to \\ Node\ (g\ [x,y,z]) & \cdots)\,)\,) \end{array}$$

Using the two conversion functions, we can now formalise that generating trees and generating functions are in one-to-one correspondence, which is an instance of a general result concerning the representation of functions with an inductive argument type — in this case finite lists — using coinductive types (Altenkirch, 2001).

Theorem 3 (*representation theorem*)
The functions gen' and rep' form an isomorphism $Tree\ a\ b \cong Gen\ a\ b$.

Proof: we verify the two parts of the isomorphism separately. For the first part, $gen' \circ rep' = id$, we are required to show that $gen'\ (rep'\ g)\ xs = g\ xs$ for any generating function g and finite list xs, which can be verified by induction on xs.

Base case:

$$\begin{array}{ll} & gen'\ (rep'\ g)\ [\,] \\ = & \quad \{\ \text{applying } rep'\ \} \\ & gen'\ (Node\ (g\ [\,])\ (\lambda x \to rep'\ (g \circ (x{:}))))\ [\,] \\ = & \quad \{\ \text{applying } gen'\ \} \\ & g\ [\,] \end{array}$$

Inductive case:

$$
\begin{array}{ll}
 & gen'\ (rep'\ g)\ (x : xs) \\
= & \quad \{ \text{ applying } rep' \ \} \\
 & gen'\ (Node\ (g\ [\,])\ (\lambda x \rightarrow rep'\ (g \circ (x:))))\ (x : xs) \\
= & \quad \{ \text{ applying } gen' \ \} \\
 & gen'\ (rep'\ (g \circ (x:)))\ xs \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & (g \circ (x:))\ xs \\
= & \quad \{ \text{ applying } \circ \ \} \\
 & g\ (x : xs)
\end{array}
$$

For the second part, $rep' \circ gen' = id$, we are required to show that $rep'\ (gen'\ t) = t$ for any generating tree $t = Node\ f\ y$, which can be verified by (guarded) coinduction on t:

$$
\begin{array}{ll}
 & rep'\ (gen'\ (Node\ y\ f)) \\
= & \quad \{ \text{ applying } gen' \ \} \\
 & rep'\ (\lambda as \rightarrow \textbf{case}\ as\ \textbf{of}\ \{[\,] \rightarrow y;\ (x : xs) \rightarrow gen'\ (f\ x)\ xs\}) \\
= & \quad \{ \text{ applying } rep',\ \text{simplification } \} \\
 & Node\ y\ (\lambda x \rightarrow rep'\ (gen'\ (f\ x))) \\
= & \quad \{ \text{ coinduction hypothesis } \} \\
 & Node\ y\ (\lambda x \rightarrow f\ x) \\
= & \quad \{ \text{ eta reduction } \} \\
 & Node\ y\ f
\end{array}
$$

□

Because the type *Tree a b* is coinductively defined, it comes equipped with a canonical means of producing values of this type, in the form of an *unfold* operator (Meijer *et al.*, 1991; Gibbons & Jones, 1998). In order to define this operator, we first introduce the notion of a *co-algebra* (Jacobs & Rutten, 1997) for generating trees:

$$\textbf{type}\ Coalg\ c\ a\ b \quad = \quad (c \rightarrow b,\ c \rightarrow a \rightarrow c)$$

That is, a co-algebra for the type *Tree a b* comprises two functions that respectively turn a value of type c into a value of type b and a function of type $a \rightarrow c$. Using this notion, the *unfold* operator for producing trees is then defined as follows:

$$
\begin{array}{lcl}
unfold & :: & Coalg\ c\ a\ b \rightarrow c \rightarrow Tree\ a\ b \\
unfold\ (h,t)\ z & = & Node\ (h\ z)\ (\lambda x \rightarrow unfold\ (h,t)\ (t\ z\ x))
\end{array}
$$

That is, given a co-algebra $(h :: c \rightarrow b,\ t :: c \rightarrow a \rightarrow c)$ and a *seed* value $z :: c$, the label of the resulting tree 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.

9 Practical applications

Combining our two representation theorems with the use of the *unfold* operator provides another means of producing contractive functions on streams. In particular, given a co-algebra and a seed value, we first apply *unfold* to produce a generating tree, then apply

gen' to convert this into a generating function, and finally apply *gen* to convert this into a contractive function. We encapsulate this idea as follows:

$$\begin{array}{lcl} \mathit{generate} & :: & \mathit{Coalg}\ c\ a\ b \to c \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{generate}\ (h,t)\ z & = & \mathit{gen}\ (\mathit{gen}'\ (\mathit{unfold}\ (h,t)\ z)) \end{array}$$

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

$$\begin{array}{lcl} \mathit{generate} & :: & \mathit{Coalg}\ c\ a\ b \to c \to (\mathit{Stream}\ a \to_c \mathit{Stream}\ b) \\ \mathit{generate}\ (h,t)\ z\ (x \lhd xs) & = & h\ z \lhd \mathit{generate}\ (h,t)\ (t\ z\ x)\ xs \end{array}$$

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.

We encapsulate the idea of defining a stream as the unique fixed point of a contractive function produced using *generate* by defining a new fixed point operator:

$$\begin{array}{lcl} \mathit{cfix} & :: & \mathit{Coalg}\ c\ a\ a \to c \to \mathit{Stream}\ a \\ \mathit{cfix}\ (h,t)\ z & = & \mathit{fix}\ (\mathit{generate}\ (h,t)\ z) \end{array}$$

To define a stream using *cfix*, we must first choose an appropriate state type to represent the history of previous values in the stream, and then define a suitable starting value and co-algebra for this type. Ideally, the state should be compact in terms of space, and the co-algebra should be efficient in terms of time. For example, the natural numbers can be defined using a state that comprises a single natural number that represents the next output value, starting value zero, and a co-algebra $(\mathit{hnats}, \mathit{tnats})$ which expresses that the next output and state are given by simply copying and incrementing the current state:

$$\begin{array}{lcl} \mathit{nats} & :: & \mathit{Stream}\ \mathit{Nat} \\ \mathit{nats} & = & \mathit{cfix}\ (\mathit{hnats}, \mathit{tnats})\ 0 \\ \mathit{hnats} & :: & \mathit{Nat} \to \mathit{Nat} \\ \mathit{hnats}\ x & = & x \\ \mathit{tnats} & :: & \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\ \mathit{tnats}\ x\ \_ & = & x+1 \end{array}$$

Similarly, the Fibonacci numbers can be defined using a state that comprises the next two output values, starting value (0,1), and a simple co-algebra on this state:

$$\begin{array}{lcl} \mathit{fibs} & :: & \mathit{Stream}\ \mathit{Nat} \\ \mathit{fibs} & = & \mathit{cfix}\ (\mathit{hfibs}, \mathit{tfibs})\ (0,1) \\ \mathit{hfibs} & :: & (\mathit{Nat}, \mathit{Nat}) \to \mathit{Nat} \\ \mathit{hfibs}\ (x,y) & = & x \\ \mathit{tfibs} & :: & (\mathit{Nat}, \mathit{Nat}) \to \mathit{Nat} \to (\mathit{Nat}, \mathit{Nat}) \\ \mathit{tfibs}\ (x,y)\ \_ & = & (y, x+y) \end{array}$$

In terms of performance, the above definitions for *nats* and *fibs* once again take linear time, but have the space advantage of using a small, finite state to produce the next output value, rather than maintaining the entire output history for this purpose.

We conclude this section by showing how *cfix* can also be used to define general purpose functions to convert different representations of contractive functions into streams. First of all, it is easy to convert a generating tree into a stream by exploiting the fact that two projection functions on nodes naturally form a (terminal) co-algebra for such trees:

$$\begin{array}{lcl} \mathit{fromtree} & :: & \mathit{Tree}\ a\ a \to \mathit{Stream}\ a \\ \mathit{fromtree} & = & \mathit{cfix}\ (\mathit{label}, \mathit{branches}) \end{array}$$

It is also straightforward to convert a generating function into a stream, by means of a co-algebra that applies the function to the empty history to produce the first output value, and prepends this value to the current history to give a new generating function that is then used to produce the remainder of the stream:

$$\begin{array}{lcl} \mathit{fromgen} & :: & \mathit{Gen}\ a\ a \to \mathit{Stream}\ a \\ \mathit{fromgen} & = & \mathit{cfix}\ (\mathit{hgen}, \mathit{tgen}) \\ \mathit{hgen} & :: & \mathit{Gen}\ a\ b \to b \\ \mathit{hgen}\ g & = & g\ [\,] \\ \mathit{tgen} & :: & \mathit{Gen}\ a\ b \to a \to \mathit{Gen}\ a\ b \\ \mathit{tgen}\ g\ x & = & g \circ (x{:}) \end{array}$$

Finally, we can produce a stream using a generating function that maintains its history in reverse time order, by using a state that combines these two components:

$$\begin{array}{lcl} \mathit{fromrgen} & :: & (\mathit{Gen}\ a\ a, [a]) \to \mathit{Stream}\ a \\ \mathit{fromrgen} & = & \mathit{cfix}\ (\mathit{hrgen}, \mathit{trgen}) \\ \mathit{hrgen} & :: & (\mathit{Gen}\ a\ b, [a]) \to b \\ \mathit{hrgen}\ (g, xs) & = & g\ xs \\ \mathit{trgen} & :: & (\mathit{Gen}\ a\ b, [a]) \to a \to (\mathit{Gen}\ a\ b, [a]) \\ \mathit{trgen}\ (g, xs)\ x & = & (g, x : xs) \end{array}$$

10 Related work

In this section, we briefly survey a selection of other work that is most closely related to our use of contractive functions as a means of ensuring that streams are well-defined.

Productivity. The notion of streams being defined in a productive manner first appears in (Dijkstra, 1980). The first semantic treatment of productivity was given by Wadge (1981), and later generalised by Sijtsma (1989). A number of techniques for ensuring productivity have since been developed, including the use of syntactic guardedness (Coquand, 1994; Telford & Turner, 1997), special type systems (Hughes *et al.*, 1996; Sculthorpe & Nilsson, 2009; Sculthorpe, 2011; McBride, 2011), custom algorithms (Endrullis *et al.*, 2007; Endrullis *et al.*, 2008), and transformation techniques (Danielsson, 2010).

Unique fixed points. The idea of exploiting the fact that recursive stream equations may have unique solutions dates back to early work on recursion operators (Hagino, 1987;

Malcolm, 1990), and has recently been promoted as a simple but powerful proof technique in its own right by Hinze (2008; 2010; 2011). In general, determining whether a stream equation has a unique solution is undecidable (Endrullis *et al.*, 2009), but a number of partial algorithms have been developed, for example based upon the construction of bisimulations (Capretta, 2010) and term rewriting systems (Zantema, 2010).

Contractive functions. The idea of defining the semantics of recursive stream equations using contractive functions on metric spaces first appears in the work of de Bakker and Kok (1985), and has since been applied to give semantics to many different kinds of languages and language features, including real-time systems (Müller & Scholz, 1997), PCF (Escardo, 1999), guarded recursion (Birkedal *et al.*, 2010a), mutable state (Birkedal *et al.*, 2010b), and reactive programs (Krishnaswami & Benton, 2011). A syntactic approach to ensuring contractivity is developed in (Buchholz, 2005), and a type based approach in (Krishnaswami & Benton, 2011). Generalisations of the underlying topological concepts have also been considered (Matthews, 1999; Gianantonio & Miculan, 2003).

Representing functions on streams. Uustalu and Vene (2006) show how arbitrary and causal functions on streams can be represented in terms of the zipper type for streams, and explore the co-monadic nature of this representation. Hancock et al. (2009) show how continuous functions on streams can be represented using infinite trees defined by mixed induction/coinduction, and have generalised their result from streams to a large class of coinductive types (Ghani *et al.*, 2009). However, none of these articles consider the stronger notion of contractive functions on streams, which as we have seen is well-suited to the problem of ensuring that recursive stream equations are well-defined.

11 Conclusion and further work

In this article we showed how Banach's fixed point theorem for contractive functions on streams can be formulated and proved correct using simple functional programming techniques, presented representation theorems that formalise the temporal nature of contractive functions and improve their performance, and showed how these theorems provide a practical means of producing streams that are guaranteed to be well-defined.

There are many interesting topics for further work that can be considered, including a range of generalisations (e.g. from streams to other terminal co-algebras, from SET to other categories, from single to mutual recursion, from first-order to higher-order stream functions, and from contractive functions to causal and continuous functions), the algebraic structure of generating functions (e.g. how such functions can be composed and how this can be done efficiently), the connection with the *unfold* operator for streams (e.g. our new fixed point operators for producing streams can also be expressed using *unfold*), more precise typings for functions on streams (e.g. tracking how far they may look into the past, or future, of their input stream), and the application of our techniques to implementing stream programming languages such as FRP (Elliott & Hudak, 1997).

Acknowledgements

We would like to thank the Functional Programming Lab in Nottingham (in particular, Thorsten Altenkirch, Florent Balestrieri, Venanzio Capretta, Henrik Nilsson and Christian

Sattler) for many useful comments and suggestions, Christian Sattler for assistance with Banach's theorem, and Peter Hancock for a number of pointers to related work.

References

Altenkirch, Thorsten. (2001). Representations of First-order Function Types as Terminal Coalgebras. *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, no. 2044.

Banach, Stefan. (1922). Sur Les Opérations Dans Les Ensembles Abstraits Et Leur Application Aux Équations Intégrales. *Fundamenta Mathematicae*, **2**, 133–181.

Bird, Richard, & Wadler, Philip. (1988). *An Introduction to Functional Programming*. Prentice Hall.

Birkedal, Lars, Schwinghammer, Jan, & Støvring, Kristian. (2010a). A Metric Model of Guarded Recursion. *Proceedings of 7th Workshop on Fixed Points in Computer Science*.

Birkedal, Lars, Støvring, Kristian, & Thamsborg, Jacob. (2010b). The Category-Theoretic Solution of Recursive Metric-Space Equations. *Theoretical Computer Science*, **411**(47), 4102–4122.

Buchholz, Wilfried. (2005). A Term Calculus for (Co-)Recursive Definitions on Streamlike Data Structures. *Annals of Pure and Applied Logic*, **136**(1-2), 75–90.

Capretta, Venanzio. (2010). Bisimulations Generated from Corecursive Equations. *Electronic Notes in Theoretical Computer Science*, **265**, 245–258. Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics.

Coquand, Thierry. (1994). Infinite Objects in Type Theory. *Pages 62–78 of:* Barendregt, Henk, & Nipkow, Tobias (eds), *Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 806. Berlin: Springer-Verlag.

Danielsson, Nils Anders. (2010). Beating the Productivity Checker Using Embedded Languages. *Pages 29–48 of: Proceedings of the Workshop on Partiality and Recursion in Interactive Theorem Provers*. Electronic Proceedings in Theoretical Computer Science, vol. 43.

de Bakker, Jaco, & Kok, Joost. (1985). Towards a Uniform Topological Treatment of Streams and Functions on Streams. *Pages 140–148 of: Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 194. Springer Berlin / Heidelberg.

Dijkstra, Edsger. (1980). *On the Productivity of Recursive Definitions*. Personal Note EWD 749, University of Texas at Austin.

Elliott, Conal, & Hudak, Paul. (1997). Functional Reactive Animation. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.

Endrullis, Jörg, Grabmayer, Clemens, Hendriks, Dimitri, Isihara, Ariya, & Klop, Jan Willem. (2007). Productivity of Stream Definitions. *Pages 274–287 of: Fundamentals of Computation Theory*. Lecture Notes in Computer Science, vol. 4639. Springer Berlin / Heidelberg.

Endrullis, Jörg, Grabmayer, Clemens, & Hendriks, Dimitri. (2008). Data-Oblivious Stream Productivity. *Pages 79–96 of: Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science, vol. 5330. Springer Berlin / Heidelberg.

Endrullis, Jörg, Grabmayer, Clemens, & Hendriks, Dimitri. (2009). Complexity of Fractran and Productivity. *Pages 371–387 of: Automated Deduction - CADE-22*. Lecture Notes in Computer Science, vol. 5663. Springer Berlin / Heidelberg.

Escardo, Martin. (1999). A Metric Model of PCF. *Proceedings of the Workshop on Realizability Semantics and Applications*.

Ghani, Neil, Hancock, Peter, & Pattinson, Dirk. (2009). Continuous Functions on Final Coalgebras. *Electronic Notes in Theoretical Computer Science*, **249**, 3–18. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics.

Gianantonio, Pietro Di, & Miculan, Marino. (2003). A Unifying Approach to Recursive and Co-recursive Definitions. *Pages 618–618 of: Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 2646. Springer Berlin / Heidelberg.

Gibbons, Jeremy, & Jones, Geraint. (1998). The Under-Appreciated Unfold. *Pages 273–279 of: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*.

Hagino, Tatsuya. (1987). *Category Theoretic Approach to Data Types*. Ph.D. thesis, University of Edinburgh.

Hancock, Peter, Pattinson, Dirk, & Ghani, Neil. (2009). Representations of Stream Processors Using Nested Fixed Points. *Logical Methods in Computer Science*, **5**(3).

Hinze, Ralf. (2008). Functional Pearl: Streams and Unique Fixed Points. *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*.

Hinze, Ralf. (2010). Concrete Stream Calculus: An Extended Study. *Journal of Functional Programming*, **20(5&6)**, 463–535.

Hinze, Ralf, & James, Daniel. (2011). Proving The Unique Fixed-Point Principle Correct. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.

Hughes, John, Pareto, Lars, & Sabry, Amr. (1996). Proving the Correctness of Reactive Systems Using Sized Types. *Proceedings of the Symposium on Principles of Programming Languages*.

Hutton, Graham. (2007). *Programming in Haskell*. Cambridge University Press.

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

Jacobs, Bart, Niqui, Milad, Rutten, Jan, & Silva, Alexandra (eds). (2010). *Proceedings of the 10th International Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science. Electronic Notes in Theoretical Computer Science Volume 264.2.

Krishnaswami, Neelakantan R., & Benton, Nick. (2011). Ultrametric Semantics of Reactive Programs. *Pages 257–266 of: Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science*.

Malcolm, Grant. (1990). Algebraic Data Types and Program Transformation. *Science of Computer Programming*, **14**(2-3), 255–280.

Matthews, John. (1999). Recursive Function Definition over Coinductive Types. *Pages 839–839 of: Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 1690. Springer Berlin / Heidelberg.

McBride, Conor. (2011). *Tomorrow is Another Day*. Talk given at the IFIP Working Group 2.1 meeting on Algorithmic Languages and Calculi in Reykjavik, Iceland.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. Hughes, John (ed), *Proceedings of the Conference on Functional Programming and Computer Architecture*. LNCS, no. 523. Springer-Verlag.

Milner, Robin. (1989). *Communication and Concurrency*. Prentice Hall.

Müller, Olaf, & Scholz, Peter. (1997). Functional Specification of Real-Time and Hybrid Systems. *Hybrid and Real-Time Systems*. Lecture Notes in Computer Science, vol. 1201. Springer Berlin / Heidelberg.

Sculthorpe, Neil. 2011 (July). *Towards Safe and Efficient Functional Reactive Programming*. Ph.D. thesis, School of Computer Science, University of Nottingham.

Sculthorpe, Neil, & Nilsson, Henrik. (2009). Safe Functional Reactive Programming Through Dependent Types. *Pages 23–34 of: Proceedings of the 14th International Conference on Functional Programming*. ACM Press.

Sijtsma, Ben. (1989). On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, **11**(4), 633–649.

Telford, Alastair, & Turner, David. (1997). Ensuring Streams Flow. *Pages 509–523 of: Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science, vol. 1349. Springer Berlin / Heidelberg.

Turner, David A. (1995). Elementary Strong Functional Programming. *Pages 1–13 of: Proceedings of the First International Symposium on Functional Programming Languages in Education*. Lecture Notes in Computer Science, vol. 1022. Springer-Verlag.

Uustalu, Tarmo, & Vene, Varmo. (2006). The Essence of Dataflow Programming. *Pages 135–167 of: Central European Functional Programming School*. Lecture Notes in Computer Science, vol. 4164. Springer Berlin / Heidelberg.

Wadge, William. (1981). An Extensional Treatment of Dataflow Deadlock. *Theoretical Computer Science*, **13**(1), 3–15.

Zantema, Hans. (2010). Well-definedness of Streams by Transformation and Termination. *Logical Methods in Computer Science*, **6**(3).