Functional Pearl: Streams and Unique Fixed Points

Ralf Hinze

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@comlab.ox.ac.uk

Abstract

Streams, infinite sequences of elements, live in a coworld: they are given by a coinductive data type, operations on streams are implemented by corecursive programs, and proofs are conducted using coinduction. But there is more to it: suitably restricted, stream equations possess *unique solutions*, a fact that is not very widely appreciated. We show that this property gives rise to a simple and attractive proof technique essentially bringing equational reasoning to the coworld. In fact, we redevelop the theory of recurrences, finite calculus and generating functions using streams and stream operators building on the cornerstone of unique solutions. The development is constructive: streams and stream operators are implemented in Haskell, usually by one-liners. The resulting calculus or library, if you wish, is elegant and fun to use. Finally, we rephrase the proof of uniqueness using generalised algebraic data types.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software/ Program Verification*]: correctness proofs, formal methods; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—specification techniques

General Terms Design, Languages, Theory, Verification

Keywords streams, unique fixed points, coinduction, recurrences, finite calculus, generating functions

1. Introduction

The cover of my favourite maths book displays a large, lonesome Σ imprinted in concrete (Graham et al. 1994). There is a certain beauty to it, but sure enough, when the letter first appears in the text, it is decorated with formulas. Sigma denotes summation and, traditionally, summation is a binder introducing an index variable that ranges over some set. More often than not, the index variable then appears as a subscript referring to an element of some other set or sequence. If you turn the pages of this paper, you won't find any index variables and not many subscripts though we deal with recurrences, summations and power series. Index variables and subscripts have their rôle, but often they can be avoided by treating the set or the sequence they refer to as a single entity. Manipulating a single entity is almost always simpler and more elegant than manipulating a cohort of singletons.

However, this paper is not about style, notation or even discrete mathematics. Rather, the paper sets out to popularise a certain proof technique, and it just so happens that recurrences, summations and power series serve admirably as illustrations. The common denominator of these examples is that they are, or rather, that they can be based on *streams*, infinite sequences of elements. In a lazy functional language, such as Haskell (Peyton Jones 2003), streams are easy to define and many textbooks on Haskell reproduce the folklore examples of Fibonacci or Hamming numbers defined by recursion equations over streams. One has to be a bit careful in formulating a recursion equation basically avoiding that the sequence defined swallows its own tail. However, if this care is exercised, the equation even possesses a *unique solution*, a fact that is not very widely appreciated. Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are! We will use this technique to infer some intriguing facts about particular streams and to develop the basics of finite calculus and generating functions. Quite attractively, the resulting proofs have a strong equational flavour. Whenever applicable, we derive programs from their specifications. We also reproduce the proof of uniqueness, which, perhaps surprisingly, involves generalised algebraic data types (Hinze 2003; Peyton Jones et al. 2006) and interpreters. But we are getting ahead of the story.

The rest of the paper is structured as follows. Sec. 2 introduces the basic definitions, laws and proof techniques. Sec. 3 shows how to capture recurrences as streams and solves some recreational puzzles. Sec. 4 applies the techniques to finite calculus. Sec. 5 introduces generating functions and explains how to solve recurrences. Finally, Sec. 6 reviews related work and Sec. 7 concludes. The proof of existence and uniqueness of solutions is relegated to App. A in order not to disturb the flow.

2. Streams

The type of streams, *Stream* α, is like Haskell's list data type $[\alpha]$, except that there is no base constructor so we cannot construct a finite stream. The *Stream* type is not an inductive type, but a *coinductive type*, whose semantics is given by a *final coalgebra* (Aczel and Mendler 1989).

$$\begin{array}{l}\textbf{data}\ \mathit{Stream}\ \alpha = \mathit{Cons}\ \{\mathit{head} :: \alpha, \mathit{tail} :: \mathit{Stream}\ \alpha\}\\ \textbf{infixr}\ 5\ \prec\\ (\prec)\quad :: \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha\\ a \prec s = \mathit{Cons}\ a\ s\end{array}$$

Streams are constructed using $\prec$, which prepends an element to a stream. They are destructed using *head*, which yields the first element, and *tail*, which returns the stream without the first element.

We say s is a *constant* stream iff *tail* $s = s$. We let s, t and u range over streams and c over constant streams.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2.1 Operations

Most definitions we encounter in the sequel make use of the following functions, which lift n-ary operations ($n = 0, 1, 2$) to streams.

$$
\begin{aligned}
&\mathit{repeat} :: \alpha \to \mathit{Stream}\ \alpha \\
&\mathit{repeat}\ a = s\ \textbf{where}\ s = a \prec s \\
&\mathit{map} :: (\alpha \to \beta) \to (\mathit{Stream}\ \alpha \to \mathit{Stream}\ \beta) \\
&\mathit{map}\ f\ s = f\ (\mathit{head}\ s) \prec \mathit{map}\ f\ (\mathit{tail}\ s) \\
&\mathit{zip} :: (\alpha \to \beta \to \gamma) \to (\mathit{Stream}\ \alpha \to \mathit{Stream}\ \beta \to \mathit{Stream}\ \gamma) \\
&\mathit{zip}\ f\ s\ t = f\ (\mathit{head}\ s)\ (\mathit{head}\ t) \prec \mathit{zip}\ f\ (\mathit{tail}\ s)\ (\mathit{tail}\ t)
\end{aligned}
$$

The call *repeat* 0 constructs a sequence of zeros ($A000004$[1]). Clearly, a constant stream is of the form *repeat* k for some k. We refer to *repeat* as a *parametrised stream* and to *map* and *zip* as *stream operators*.

For convenience and conciseness of notation, let us lift the arithmetic operations to streams. In Haskell, this is easily accomplished using type classes. Here is an excerpt of the necessary code.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Num}\ a) \Rightarrow \mathit{Num}\ (\mathit{Stream}\ a)\ \textbf{where} \\
&\quad (+) \quad = \mathit{zip}\ (+) \\
&\quad (-) \quad = \mathit{zip}\ (-) \\
&\quad (*) \quad = \mathit{zip}\ (*) \\
&\quad \mathit{negate} \quad = \mathit{map}\ \mathit{negate} \quad \text{-- unary minus} \\
&\quad \mathit{fromInteger}\ i = \mathit{repeat}\ (\mathit{fromInteger}\ i)
\end{aligned}
$$

This instance declaration allows us, in particular, to use integer constants as streams — in Haskell, unqualified 3 abbreviates *fromInteger* $(3 :: \mathit{Integer})$.

Using this vocabulary we can already define the usual suspects: the natural numbers ($A001477$), the factorial numbers ($A000142$), and the Fibonacci numbers ($A000045$).

$$
\begin{aligned}
\mathit{nat} &= 0 \prec \mathit{nat} + 1 \\
\mathit{fac} &= 1 \prec (\mathit{nat} + 1) * \mathit{fac} \\
\mathit{fib} &= 0 \prec \mathit{fib}' \\
\mathit{fib}' &= 1 \prec \mathit{fib}' + \mathit{fib}
\end{aligned}
$$

Note that $\prec$ binds less tightly than $+$. For instance, $0 \prec \mathit{nat} + 1$ is grouped $0 \prec (\mathit{nat} + 1)$. The three sequences are given by recursion equations adhering to a strict scheme: each equation defines the head and the tail of the sequence, the latter possibly in terms of the entire sequence. The Fibonacci numbers provide an example of mutual recursion: fib', which denotes the tail of the sequence, refers to *fib* and vice versa. Actually, in this case mutual recursion is not necessary as a quick calculation shows: $\mathit{fib}' = 1 \prec \mathit{fib}' + \mathit{fib} = (0 \prec \mathit{fib}') + (1 \prec \mathit{fib}) = \mathit{fib} + (1 \prec \mathit{fib})$. So, an alternative definition is

$$\mathit{fib} = 0 \prec \mathit{fib} + (1 \prec \mathit{fib})$$

As an aside, we will use the convention that the identifier x' denotes the tail of x.

It's fun to play with the sequences. Here is a short interactive session.

$\gg$ *fib*
$<0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ..>$
$\gg$ $\mathit{nat} * \mathit{nat}$
$<0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ..>$
$\gg$ $\mathit{tail}\ \mathit{fib}\ \hat{}\ 2 - \mathit{fib} * \mathit{tail}\ (\mathit{tail}\ \mathit{fib})$
$<1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, ..>$
$\gg$ $\mathit{tail}\ \mathit{fib}\ \hat{}\ 2 - \mathit{fib} * \mathit{tail}\ (\mathit{tail}\ \mathit{fib}) == (-1)\ \hat{}\ \mathit{nat}$
True

[1] Most if not all integer sequences defined in this paper are recorded in Sloane's On-Line Encyclopedia of Integer Sequences (Sloane). Keys of the form $Annnnnn$ refer to entries in that database. Somewhat surprisingly, *repeat* 0 is not $A000000$. Just in case you were wondering, the first sequence ($A000001$) lists the number of groups of order n.

The part after the prompt, $\gg$, is the user's input. The result of each submission is shown in the subsequent line. This is the actual output of the Haskell interpreter; the session has been generated automatically using lhs2TeX's active features (Hinze and Löh 2008).

Obviously, we can't print out a sequence in full. The *Show* instance for *Stream* only displays the first n elements. Likewise, we can't test two streams for equality: $==$ only checks whether the first n elements are equal. So, 'equality' is most useful for falsifying conjectures. For the purposes of this paper, n equals 16.

Another important operator is *interleaving* of two streams.

$$
\begin{aligned}
&\textbf{infixr}\ 5\ \curlyvee \\
&(\curlyvee) :: \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \\
&s \curlyvee t = \mathit{head}\ s \prec t \curlyvee \mathit{tail}\ s
\end{aligned}
$$

Though the symbol is symmetric, $\curlyvee$ is not commutative. Neither it is associative. Let's look at an example application. The above definition of the naturals is based on the unary number system. Using interleaving, we can alternatively base the sequence on the binary number system.

$$\mathit{bin} = 0 \prec 2 * \mathit{bin} + 1 \curlyvee 2 * \mathit{bin} + 2$$

Note that $\curlyvee$ has lower precedence than the arithmetic operators. For instance, the right-hand side of the equation above is grouped $0 \prec ((2 * \mathit{bin} + 1) \curlyvee (2 * \mathit{bin} + 2))$.

Now that we have two definitions of the natural numbers, the question naturally arises as to whether they are actually equal. Reassuringly, the answer is in the affirmative. Proving the equality of streams or of stream operators is one of our main activities in the sequel. However, we postpone a proof of $\mathit{nat} = \mathit{bin}$ until we have the necessary prerequisites at hand.

Finally, we can build a stream by repeatedly applying a given function to a given value.

$$
\begin{aligned}
&\mathit{iterate} :: (\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\ \alpha) \\
&\mathit{iterate}\ f\ a = a \prec \mathit{iterate}\ f\ (f\ a)
\end{aligned}
$$

So, *iterate* $(+1)\ 0$ is yet another definition of the naturals.

2.2 Definitions

Not every legal Haskell definition of type *Stream* τ actually defines a stream. Two simple counterexamples are $s = \mathit{tail}\ s$ and $s = \mathit{head}\ s \prec \mathit{tail}\ s$. Both of them loop in Haskell; viewed as stream equations they are ambiguous. In fact, they admit infinitely many solutions: every constant stream is a solution of the first equation, every stream is a solution of the second one. This situation is undesirable from both a practical and a theoretical standpoint. Fortunately, it is not hard to restrict the syntactic form of equations so that they possess *unique solutions*. We insist that equations adhere to the following form.

$$x = h \prec t$$

where x is an identifier of type *Stream* τ, h is a constant expression of type τ, and t is an expression of type *Stream* τ possibly referring to x or some other stream identifier in the case of mutual recursion. However, neither h nor t may involve *head* x or *tail* x.

If x is a parametrised stream or a stream operator,

$$x\ x_1\ \ldots\ x_n = h \prec t$$

then h or t may use *head* x_i or *tail* x_i provided x_i is of the right type. Furthermore, t may contain recursive calls to x, but we are not allowed to take the head or the tail of a recursive call. However, there are no further restrictions on the form of the arguments.

For a formal account of these requirements, we refer the interested reader to App. A, which contains a constructive proof that equations of this form indeed have unique solutions. Looking back, we find that the definitions we have encountered so far, including *map*, *zip* and $\curlyvee$, meet the requirements.

As an aside, we could relax the conditions somewhat so that

$$\mathit{fib} = 0 \prec 1 \prec \mathit{tail}\ \mathit{fib} + \mathit{fib}$$

becomes admissible. However, the gain in expressivity is modest as we can always eliminate such calls to *tail* by introducing a name for the tail. In the example above, we simply replace *tail fib* by fib' obtaining the two equations given in Sec. 2.1.

By the way, non-recursive definitions like

$$\mathit{nat}' = \mathit{nat} + 1$$

are unproblematic and unrestricted as they can always be inlined.

2.3 Laws

Since the arithmetic operations are defined point-wise, the familiar arithmetic laws also hold for streams. In proofs we will signal their use by the hint *arithmetic*.

Streams satisfy the following *extensionality* property.

$$s \quad = \quad \mathit{head}\ s \prec \mathit{tail}\ s$$

App. A provides a coinductive proof of this law.

Interleaving interacts nicely with lifted operations: let $\ominus = \mathit{map}\ (\boxminus)$ and $\oplus = \mathit{zip}\ (\boxplus)$, $\boxminus$ and $\boxplus$ arbitrary functions, then

$$\begin{aligned} c \curlyvee c \quad &= \quad c \\ (\ominus s) \curlyvee (\ominus t) \quad &= \quad \ominus (s \curlyvee t) \\ (s_1 \oplus s_2) \curlyvee (t_1 \oplus t_2) \quad &= \quad (s_1 \curlyvee t_1) \oplus (s_2 \curlyvee t_2) \end{aligned}$$

A simple consequence is $(s \curlyvee t) + 1 = s + 1 \curlyvee t + 1$. The last property is called *abide law* because of the following two-dimensional way of writing the law, in which the two operators are written either *ab*ove or bes*ide* each other.

$$\begin{array}{ccc} s_1 & \oplus & s_2 \\ & \curlyvee & \\ t_1 & \oplus & t_2 \end{array} \quad = \quad \begin{array}{ccc} s_1 & & s_2 \\ \curlyvee & \oplus & \curlyvee \\ t_1 & & t_2 \end{array}$$

The two-dimensional arrangement is originally due to Hoare, the catchy name is due to Bird.

2.4 Proofs

In Sec. 2.2 we have planted the seeds by restricting the syntactic form of equations so that they possess unique solutions. It is now time to reap the harvest. If $s = \phi\ s$ is an admissible equation, we denote its unique solution by $\mathit{fix}\ \phi$. (The equation implicitly defines a function in s. A solution of the equation is a fixed point of this function and vice versa.) The fact that the solution is unique is captured by the following universal property.

$$\mathit{fix}\ \phi = s \quad \Longleftrightarrow \quad \phi\ s = s$$

Read from left to right it states that $\mathit{fix}\ \phi$ is indeed a solution of $x = \phi\ x$. Read from right to left it asserts that any solution is equal to $\mathit{fix}\ \phi$. So, if we want to prove $s = t$ where $s = \mathit{fix}\ \phi$, then it suffices to show that $\phi\ t = t$.

As a first example, let us prove an earlier claim, namely, that a constant stream is of the form *repeat* k for some k.

$$\begin{aligned} & c \\ = \quad & \{\ \text{extensionality}\ \} \\ & \mathit{head}\ c \prec \mathit{tail}\ c \\ = \quad & \{\ c \text{ is constant}\ \} \\ & \mathit{head}\ c \prec c \end{aligned}$$

Consequently, c equals the unique solution of $x = \mathit{head}\ c \prec x$, which by definition is $\mathit{repeat}\ (\mathit{head}\ c)$.

That was easy. The next proof is not much harder: we show that $\mathit{nat} = 2 * \mathit{nat} \curlyvee 2 * \mathit{nat} + 1$.

$$\begin{aligned} & 2 * \mathit{nat} \curlyvee 2 * \mathit{nat} + 1 \\ = \quad & \{\ \text{definition of } \mathit{nat}\ \} \\ & 2 * (0 \prec \mathit{nat} + 1) \curlyvee 2 * \mathit{nat} + 1 \\ = \quad & \{\ \text{arithmetic}\ \} \\ & (0 \prec 2 * \mathit{nat} + 2) \curlyvee 2 * \mathit{nat} + 1 \\ = \quad & \{\ \text{definition of } \curlyvee\ \} \\ & 0 \prec 2 * \mathit{nat} + 1 \curlyvee 2 * \mathit{nat} + 2 \\ = \quad & \{\ \text{arithmetic}\ \} \\ & 0 \prec (2 * \mathit{nat} \curlyvee 2 * \mathit{nat} + 1) + 1 \end{aligned}$$

Inspecting the second but last term, we note that the result implies $\mathit{nat} = 0 \prec 2{*}\mathit{nat}{+}1 \curlyvee 2{*}\mathit{nat}{+}2$, which in turn proves $\mathit{nat} = \mathit{bin}$.

Now, if both s and t are given as fixed points, $s = \mathit{fix}\ \phi$ and $t = \mathit{fix}\ \psi$, then there are at least four possibilities to prove $s = t$:

$$\begin{array}{lllll} \phi\ (\psi\ s) = \psi\ s & \Longrightarrow & \psi\ s = s & \Longrightarrow & s = t \\ \psi\ (\phi\ t) = \phi\ t & \Longrightarrow & \phi\ t = t & \Longrightarrow & s = t \end{array}$$

We may be lucky and establish one of the equations. Unfortunately, there is no success guarantee. The following approach is often more promising. We show $s = \chi\ s$ and $\chi\ t = t$. If χ has a unique fixed point, then $s = t$. The point is that we discover the function χ on the fly during the calculation. Proofs in this style are laid out as follows.

$$\begin{aligned} & s \\ = \quad & \{\ \text{why?}\ \} \\ & \chi\ s \\ \subset \quad & \{\ x = \chi\ x \text{ has a unique solution}\ \} \\ & \chi\ t \\ = \quad & \{\ \text{why?}\ \} \\ & t \end{aligned}$$

The symbol $\subset$ is meant to suggest a link connecting the upper and the lower part. Overall, the proof establishes that $s = t$.

Let us illustrate the technique by proving *Cassini's identity*: $\mathit{fib}' \mathbin{\hat{}} 2 - \mathit{fib} * \mathit{fib}'' = (-1) \mathbin{\hat{}} \mathit{nat}$ where $\mathit{fib}'' = \mathit{tail}\ \mathit{fib}' = \mathit{fib}' + \mathit{fib}$.

$$\begin{aligned} & \mathit{fib}' \mathbin{\hat{}} 2 - \mathit{fib} * \mathit{fib}'' \\ = \quad & \{\ \text{definition of } \mathit{fib}'' \text{ and arithmetic}\ \} \\ & \mathit{fib}' \mathbin{\hat{}} 2 - (\mathit{fib} * \mathit{fib}' + \mathit{fib} \mathbin{\hat{}} 2) \\ = \quad & \{\ \text{definition of } \mathit{fib} \text{ and } \mathit{fib}'\ \} \\ & 1 \prec (\mathit{fib}'' \mathbin{\hat{}} 2 - (\mathit{fib}' * \mathit{fib}'' + \mathit{fib}' \mathbin{\hat{}} 2)) \\ = \quad & \{\ \mathit{fib}'' - \mathit{fib}' = \mathit{fib} \text{ and arithmetic}\ \} \\ & 1 \prec (-1) * (\mathit{fib}' \mathbin{\hat{}} 2 - \mathit{fib} * \mathit{fib}'') \\ \subset \quad & \{\ x = 1 \prec (-1) * x \text{ has a unique solution}\ \} \\ & 1 \prec (-1) * (-1) \mathbin{\hat{}} \mathit{nat} \\ = \quad & \{\ \text{arithmetic}\ \} \\ & (-1) \mathbin{\hat{}} 0 \prec (-1) \mathbin{\hat{}} (\mathit{nat} + 1) \\ = \quad & \{\ \text{definition of } \mathit{nat} \text{ and arithmetic}\ \} \\ & (-1) \mathbin{\hat{}} \mathit{nat} \end{aligned}$$

When reading $\subset$-proofs, it is easiest to start at both ends working towards the link. Each part follows a typical pattern, which we will see time and time again: starting with e we unfold the definitions obtaining $e_1 \prec e_2$; then we try to express e_2 in terms of e.

So far, we have been concerned with proofs about streams. However, the proof techniques apply equally well to parametric streams or stream operators! As an example, let us prove the second $\curlyvee$-law by showing $f = g$ where

$$f\ s\ t = \ominus s \curlyvee \ominus t \quad \text{and} \quad g\ s\ t = \ominus (s \curlyvee t)$$

The proof is straightforward involving only bureaucratic steps.

$$
\begin{array}{ll}
& f\ a\ b \\
= & \quad \{ \text{definition of } f \} \\
& \ominus a \curlyvee \ominus b \\
= & \quad \{ \text{definition of } \curlyvee \text{ and } \ominus = \mathit{map}\ (\boxminus) \} \\
& \boxminus \mathit{head}\ a \prec \ominus b \curlyvee \ominus \mathit{tail}\ a \\
= & \quad \{ \text{definition of } f \} \\
& \boxminus \mathit{head}\ a \prec f\ b\ (\mathit{tail}\ a) \\
\subset & \quad \{ x\ s\ t = \boxminus \mathit{head}\ s \prec x\ t\ (\mathit{tail}\ s) \text{ has a unique solution} \} \\
& \boxminus \mathit{head}\ a \prec g\ b\ (\mathit{tail}\ a) \\
= & \quad \{ \text{definition of } g \} \\
& \boxminus \mathit{head}\ a \prec \ominus (b \curlyvee \mathit{tail}\ a) \\
= & \quad \{ \ominus = \mathit{map}\ (\boxminus) \text{ and definition of } \curlyvee \} \\
& \ominus (a \curlyvee b) \\
= & \quad \{ \text{definition of } g \} \\
& g\ a\ b
\end{array}
$$

In the sequel, we usually leave the two functions implicit sparing ourselves two rolling and two unrolling steps. On the downside, this makes the common pattern around the link more difficult to spot.

A popular benchmark for the effectiveness of proof methods for corecursive programs is the *iterate* fusion law (Gibbons and Hutton 2005), which amounts to the free theorem of $(\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\ \alpha)$.

$$\mathit{map}\ h \cdot \mathit{iterate}\ f_1 = \mathit{iterate}\ f_2 \cdot h \quad \Longleftarrow \quad h \cdot f_1 = f_2 \cdot h$$

The 'unique fixed-point proof' is short and sweet; it compares favourably to the ones given by Gibbons and Hutton (2005).

$$
\begin{array}{ll}
& \mathit{map}\ h\ (\mathit{iterate}\ f_1\ a) \\
= & \quad \{ \text{definition of } \mathit{iterate} \text{ and } \mathit{map} \} \\
& h\ a \prec \mathit{map}\ h\ (\mathit{iterate}\ f_1\ (f_1\ a)) \\
\subset & \quad \{ x\ a = h\ a \prec x\ (f_1\ a) \text{ has a unique solution} \} \\
& h\ a \prec \mathit{iterate}\ f_2\ (h\ (f_1\ a)) \\
= & \quad \{ \text{assumption: } h \cdot f_1 = f_2 \cdot h \} \\
& h\ a \prec \mathit{iterate}\ f_2\ (f_2\ (h\ a)) \\
= & \quad \{ \text{definition of } \mathit{iterate} \} \\
& \mathit{iterate}\ f_2\ (h\ a)
\end{array}
$$

Interestingly, the linking equation $g\ a = h\ a \prec g\ (f_1\ a)$ corresponds to the *unfold* operator, which captures a common recursion pattern of stream-producing functions, see App. A.3.

The fusion law implies $\mathit{map}\ f \cdot \mathit{iterate}\ f = \mathit{iterate}\ f \cdot f$, which is the key for proving $\mathit{nat} = \mathit{iterate}\ (+1)\ 0$.

$$
\begin{array}{ll}
& \mathit{iterate}\ (+1)\ 0 \\
= & \quad \{ \text{definition of } \mathit{iterate} \} \\
& 0 \prec \mathit{iterate}\ (+1)\ 1 \\
= & \quad \{ \text{iterate fusion law: } h = f_1 = f_2 = (+1) \} \\
& 0 \prec \mathit{iterate}\ (+1)\ 0 + 1
\end{array}
$$

3. Recurrences ($\prec$, $\curlyvee$)

Using $\prec$ and $\curlyvee$ we can easily capture recurrences: the sequence defined by $a_0 = k$ and $a_{n+1} = f(a_n)$ becomes the stream equation $a = k \prec \mathit{map}\ f\ a$; likewise, the sequence given by $a_0 = k$, $a_{2n+1} = f(a_n)$ and $a_{2n+2} = g(a_n)$ becomes $a = k \prec \mathit{map}\ f\ a \curlyvee \mathit{map}\ g\ a$. The point of this paper is that a stream is easier to manipulate than a recurrence because a stream is a single entity, often defined by a single equation. Nonetheless, you may want to keep the correspondence in mind when studying the following examples.

3.1 Bit fiddling

To familiarise ourselves with the notation, let us tackle some easy problems first. How can we characterise the pots, the powers of two (A036987)? Clearly, 1 is a pot (we only consider positive numbers); the even number $2n$ is a pot, if n is; an odd number greater than 1 is not one.

$$\mathit{pot} = \mathit{True} \prec \mathit{pot} \curlyvee \mathit{repeat}\ \mathit{False}$$

Using a similar approach we can characterise the most significant bit of a positive number ($0 \prec \mathit{msb}$ is A053644).

$$\mathit{msb} = 1 \prec 2 * \mathit{msb} \curlyvee 2 * \mathit{msb}$$

Put differently, *msb* is the largest pot less than or equal to nat'. (Here we lift relations, "x and y are related by R", to streams.)

Another example along these lines is the 1s-counting sequence (A000120), also known as the *binary weight*. The binary representation of the even number $2n$ has the same number of 1s as n; the odd number $2n + 1$ has one 1 more. Hence, the sequence satisfies $\mathit{ones} = \mathit{ones} \curlyvee \mathit{ones} + 1$. Adding two initial values, we can turn the property into a definition.

$$
\begin{aligned}
\mathit{ones} &= 0 \prec \mathit{ones}' \\
\mathit{ones}' &= 1 \prec \mathit{ones}' \curlyvee \mathit{ones}' + 1
\end{aligned}
$$

It is important to note that $x = x \curlyvee x + 1$ does not have a unique solution. However, all solutions are of the form $\mathit{ones} + c$.

Let's inspect the sequences.

$$
\begin{array}{l}
\gg\ \mathit{msb} \\
\langle 1, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, 16, ..\rangle \\
\gg\ \mathit{nat}' - \mathit{msb} \\
\langle 0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, ..\rangle \\
\gg\ \mathit{ones} \\
\langle 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, ..\rangle
\end{array}
$$

The sequence $\mathit{nat}' - \mathit{msb}$ (A053645) exhibits a nice pattern; it describes the distance to the largest pot less than or equal to nat'.

3.2 Binary carry sequence

Here is a sequence that every computer scientist should know: the *binary carry sequence* or *ruler function* (A007814).

$$\mathit{carry} = 0 \curlyvee \mathit{carry} + 1$$

(The form of the equation does not quite meet the requirements. We allow ourselves some liberty as a simple unfolding turns it into an admissible form: $\mathit{carry} = 0 \prec \mathit{carry} + 1 \curlyvee 0$. The unfolding works as long as the the first argument of $\curlyvee$ is a sequence defined elsewhere.) The sequence gives the exponent of the largest pot dividing nat', that is, the number of trailing zeros in the binary representation. In other words, it specifies the running time of the binary increment.

There is also an intriguing connection to infinite binary trees. Consider the following definition.

$$
\begin{aligned}
\mathit{turn}\ 0 &= [\,] \\
\mathit{turn}\ (n + 1) &= \mathit{turn}\ n +\!\!+ [n] +\!\!+ \mathit{turn}\ n
\end{aligned}
$$

The call $\mathit{turn}\ n$ yields the heights of the nodes of a perfect binary tree of depth n. Now, imagine traversing an infinite binary tree starting at the leftmost leaf: visit the current node, visit its finite right subtree and then continue with its parent — the tree has no root, it extends infinitely upwards. The following parametrised stream captures the traversal.

$$\mathit{tree}\ n = n \prec \mathit{turn}\ n \prec\!\!\prec \mathit{tree}\ (n + 1)$$

where $\prec\!\!\prec$ prepends a list to a sequence.

$$\begin{array}{lll}
\textbf{infixr}\ 5\ \prec\!\!\prec & & \\
(\prec\!\!\prec) & :: & [\alpha] \rightarrow \mathit{Stream}\ \alpha \rightarrow \mathit{Stream}\ \alpha \\
[\,] & \prec\!\!\prec s = s & \\
(a : \mathit{as}) & \prec\!\!\prec s = a \prec (\mathit{as} \prec\!\!\prec s) &
\end{array}$$

Here is the punch line: *tree* 0 also yields the binary carry sequence!

Turning to the proof, let us try the obvious: we show that *tree* 0 satisfies the equation $x = 0 \curlyvee x + 1$.

$$\begin{array}{ll}
 & 0 \curlyvee \mathit{tree}\ 0 + 1 \\
= & \quad \{ \text{ definition of } \curlyvee \ \} \\
 & 0 \prec \mathit{tree}\ 0 + 1 \curlyvee 0 \\
= & \quad \{ \text{ proof obligation, see below } \} \\
 & 0 \prec \mathit{tree}\ 1 \\
= & \quad \{ \text{ definition of } \mathit{tree} \text{ and } \mathit{turn} \ \} \\
 & \mathit{tree}\ 0
\end{array}$$

We are left with the proof obligation *tree* $1 =$ *tree* $0 + 1 \curlyvee 0$. With some foresight, we generalise to *tree* $(k + 1) =$ *tree* $k + 1 \curlyvee 0$. The $\subset$-proof below makes essential use of the *mixed abide law*: if *length* $x =$ *length* y, then

$$(x \prec\!\!\prec s) \curlyvee (y \prec\!\!\prec t) \quad = \quad (x \curlyvee y) \prec\!\!\prec (s \curlyvee t)$$

where $\curlyvee$ in $x \curlyvee y$ denotes interleaving of two *lists* of the same length. Noting that *length* $(\mathit{turn}\ n) = 2^n - 1$, we reason (*replicate* is abbreviated by *rep*)

$$\begin{array}{ll}
 & \mathit{tree}\ (k+1) \\
= & \quad \{ \text{ definition of } \mathit{tree} \ \} \\
 & k + 1 \prec \mathit{turn}\ (k+1) \prec\!\!\prec \mathit{tree}\ (k+2) \\
\subset & \quad \{ \ x\ n = n + 1 \prec \mathit{turn}\ (n+1) \prec\!\!\prec x\ (n+1) \text{ has a u. s. } \} \\
 & k + 1 \prec \mathit{turn}\ (k+1) \prec\!\!\prec (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\
= & \quad \{ \text{ proof obligation, see below } \} \\
 & k + 1 \prec (\mathit{rep}\ 2^k\ 0 \curlyvee \mathit{turn}\ k + 1) \prec\!\!\prec (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\
= & \quad \{ \text{ definition of } \curlyvee \text{ and definition of } \prec\!\!\prec \ \} \\
 & ((k + 1 : \mathit{turn}\ k + 1) \curlyvee \mathit{rep}\ 2^k\ 0) \prec\!\!\prec (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\
= & \quad \{ \text{ mixed abide law } \} \\
 & ((k + 1 : \mathit{turn}\ k + 1) \prec\!\!\prec \mathit{tree}\ (k+1) + 1) \curlyvee (\mathit{rep}\ 2^k\ 0 \prec\!\!\prec 0) \\
= & \quad \{ \text{ arithmetic and definition of } \prec\!\!\prec \ \} \\
 & (k \prec \mathit{turn}\ k \prec\!\!\prec \mathit{tree}\ (k+1)) + 1 \curlyvee 0 \\
= & \quad \{ \text{ definition of } \mathit{tree} \ \} \\
 & \mathit{tree}\ k + 1 \curlyvee 0
\end{array}$$

It remains to show the finite version of the proof obligation: *turn* $(k + 1) =$ *replicate* $2^k\ 0 \curlyvee$ *turn* $k + 1$. We omit the straightforward induction, which relies on an abide law for lists.

3.3 Fractal sequences

The sequence *pot* and the 1s-counting sequence are examples of *fractal* or *self-similar* sequences: a subsequence is identical to the entire sequence. Another fractal sequence is A025480.

$$\mathit{frac} = \mathit{nat} \curlyvee \mathit{frac}$$

This sequence contains infinitely many copies of the natural numbers. The distance between equal numbers grows exponentially, 2^n, as we progress to the right. Like *carry*, *frac* is related to divisibility:

$$\mathit{god} = 2 * \mathit{frac} + 1$$

is the *greatest odd divisor* of $\mathit{nat}' = 2 * \mathit{nat} + 1 \curlyvee 2 * \mathit{nat} + 2$: The greatest odd divisor of an odd number, $2 * \mathit{nat} + 1$, is the number itself; the greatest odd divisor of an even number, $2 * \mathit{nat} + 2$, is the *god* of nat'.

Now, recall that $2\ \hat{}\ \mathit{carry}$ is the largest power of two dividing nat'. Putting these observations together, we have

$$2\ \hat{}\ \mathit{carry} * \mathit{god} \quad = \quad \mathit{nat}'$$

The proof is surprisingly straightforward.

$$\begin{array}{ll}
 & 2\ \hat{}\ \mathit{carry} * \mathit{god} \\
= & \quad \{ \text{ definition of } \mathit{carry} \text{ and } \mathit{god} \ \} \\
 & 2\ \hat{}\ (0 \curlyvee \mathit{carry} + 1) * (2 * \mathit{nat} + 1 \curlyvee \mathit{god}) \\
= & \quad \{ \text{ arithmetic and abide law } \} \\
 & 2 * \mathit{nat} + 1 \curlyvee 2 * 2\ \hat{}\ \mathit{carry} * \mathit{god} \\
\subset & \quad \{ \ x = 2 * \mathit{nat} + 1 \curlyvee 2 * x \text{ has a unique solution } \} \\
 & 2 * \mathit{nat} + 1 \curlyvee 2 * (\mathit{nat} + 1) \\
= & \quad \{ \text{ arithmetic } \} \\
 & 2 * \mathit{nat} + 1 \curlyvee 2 * \mathit{nat} + 2 \\
= & \quad \{ \text{ property of } \mathit{nat}' \text{, see above } \} \\
 & \mathit{nat}'
\end{array}$$

3.4 Josephus problem

Our final example is a variant of the *Josephus problem* (Graham et al. 1994, Sec. 1.3). Imagine n people numbered 1 to n forming a circle. Every second person is killed until only one survives. Our task is to determine the survivor's number.

Now, if there is only one person, then this person survives. For an even number of persons the martial process starts as follows: $1\ 2\ 3\ 4\ 5\ 6$ becomes $1\ \cancel{2}\ 3\ \cancel{4}\ 5\ \cancel{6}$. Renumbering $1\ 3\ 5$ to $1\ 2\ 3$, we observe that if i is killed in the sequence of first-round survivors, then $2i - 1$ is killed in the original sequence. Likewise for odd numbers: $1\ 2\ 3\ 4\ 5\ 6\ 7$ becomes $\cancel{1}\ \cancel{2}\ 3\ \cancel{4}\ 5\ \cancel{6}\ 7$ — since the number is odd, the first person is killed, as well. Renumbering $3\ 5\ 7$ to $1\ 2\ 3$, we observe that if i is killed in the remaining sequence, then $2i + 1$ is killed in the original sequence.

$$\mathit{jos} = 1 \prec 2 * \mathit{jos} - 1 \curlyvee 2 * \mathit{jos} + 1$$

It's quite revealing to inspect the sequence.

$$\begin{array}{l}
\gg\ \ \mathit{jos} \\
<1, 1, 3, 1, 3, 5, 7, 1, 3, 5, 7, 9, 11, 13, 15, 1, ..\, > \\
\gg\ \ (\mathit{jos} - 1)\ /\ 2 \\
<0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, ..\, >
\end{array}$$

Since the even numbers are eliminated in the first round, *jos* only contains odd numbers. If we divide $\mathit{jos} - 1$ by two, we obtain a sequence we have encountered before: $\mathit{nat}' - \mathit{msb}$. Indeed,

$$\mathit{jos} \quad = \quad 2 * (\mathit{nat}' - \mathit{msb}) + 1$$

In terms of bit operations, *jos* implements a *cyclic left shift*: $\mathit{nat}' - \mathit{msb}$ removes the most significant bit, $2*$ shifts to the left and $+1$ sets the least significant bit.

$$\begin{array}{ll}
 & 2 * (\mathit{nat}' - \mathit{msb}) + 1 \\
= & \quad \{ \text{ definition of } \mathit{msb} \text{ and property of } \mathit{nat}' \ \} \\
 & 2 * ((1 \prec 2 * \mathit{nat}' \curlyvee 2 * \mathit{nat}' + 1) - \\
 & \qquad (1 \prec 2 * \mathit{msb} \curlyvee 2 * \mathit{msb})) + 1 \\
= & \quad \{ \text{ definition of } - \text{ and abide law } \} \\
 & 2 * (0 \prec 2 * \mathit{nat}' - 2 * \mathit{msb} \curlyvee 2 * \mathit{nat}' + 1 - 2 * \mathit{msb}) + 1 \\
= & \quad \{ \text{ arithmetic } \} \\
 & 1 \prec 2 * (2 * (\mathit{nat}' - \mathit{msb}) + 1) - 1 \curlyvee \\
 & \qquad 2 * (2 * (\mathit{nat}' - \mathit{msb}) + 1) + 1
\end{array}$$

4. Finite calculus (Δ, Σ)

Let's move on to another application of streams: *finite calculus*. Finite calculus is the discrete counterpart of infinite calculus, where finite difference replaces the derivative and summation replaces integration. We shall see that difference and summation can be easily recast as stream operators.

4.1 Finite difference

A common type of puzzle asks the reader to continue a given sequence of numbers. A first routine step towards solving the puzzle is to calculate the difference of subsequent elements. This stream operator, *finite difference* or *forward difference*, enjoys a simple, non-recursive definition.

$$\begin{aligned}&\Delta \; :: \; (\mathit{Num}\ \alpha) \Rightarrow \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha\\ &\Delta\ s = \mathit{tail}\ s - s\end{aligned}$$

Here are some examples (A003215, A000079, A094267, not listed).

$\gg$ $\Delta\ (\mathit{nat}\ \hat{}\ 3)$
<1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, .. >
$\gg$ $\Delta\ (2\ \hat{}\ \mathit{nat})$
<1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, .. >
$\gg$ Δ *carry*
<1, −1, 2, −2, 1, −1, 3, −3, 1, −1, 2, −2, 1, −1, 4, −4, .. >
$\gg$ Δ *jos*
<0, 2, −2, 2, 2, 2, −6, 2, 2, 2, 2, 2, 2, 2, −14, 2, .. >

Infinite calculus has a simple rule for the derivative of a power: $(x^{n+1})' = (n+1)x^n$. Unfortunately, the first example above shows that finite difference does not interact nicely with ordinary powers: $\Delta\ (\mathit{nat}\ \hat{}\ 3)$ is not $3 * \mathit{nat}\ \hat{}\ 2$. Can we find a different notion that enjoys an analogous rule? Let's try. Writing $x^{\underline{n}}$ for the new power and its lifted variant, we calculate

$$\begin{aligned}&\Delta\ (\mathit{nat}^{\underline{n+1}})\\ =\;&\quad \{\ \text{definition of } \Delta\ \}\\ &\mathit{tail}\ (\mathit{nat}^{\underline{n+1}}) - \mathit{nat}^{\underline{n+1}}\\ =\;&\quad \{\ s^{\underline{n}} = \mathit{map}\ (\lambda x \to x^{\underline{n}})\ s \text{ and definition of } \mathit{nat}\ \}\\ &(\mathit{nat}+1)^{\underline{n+1}} - \mathit{nat}^{\underline{n+1}}\end{aligned}$$

Starting at the other end, we obtain

$$\begin{aligned}&(\mathit{repeat}\ n+1) * \mathit{nat}^{\underline{n}}\\ =\;&\quad \{\ \text{arithmetic}\ \}\\ &(\mathit{nat}+1) * \mathit{nat}^{\underline{n}} - \mathit{nat}^{\underline{n}} * (\mathit{nat} - \mathit{repeat}\ n)\end{aligned}$$

We can connect the loose ends if the new power satisfies both $x * (x-1)^{\underline{n}} = x^{\underline{n+1}} = x^{\underline{n}} * (x-n)$. That's easy to arrange, we use the first equation as a definition. (It is not hard to see that the definition then also satisfies the second equation.)

$$\begin{aligned}x^{\underline{0}} &= 1\\ x^{\underline{n+1}} &= x * (x-1)^{\underline{n}}\end{aligned}$$

The new powers are, of course, well-known: they are called *falling factorial powers*.

One can convert mechanically between powers and falling factorial powers using Stirling numbers (Graham et al. 1994, Sec. 6.1). The details are beyond the scope of this paper. For reference, Fig. 1 displays the correspondence up to the third power.

$$\begin{aligned}x^0 &= x^{\underline{0}} & x^{\underline{0}} &= x^0\\ x^1 &= x^{\underline{1}} & x^{\underline{1}} &= x^1\\ x^2 &= x^{\underline{2}} + x^{\underline{1}} & x^{\underline{2}} &= x^2 - x^1\\ x^3 &= x^{\underline{3}} + 3 * x^{\underline{2}} + x^{\underline{1}} & x^{\underline{3}} &= x^3 - 3 * x^2 + 2 * x^1\end{aligned}$$

Figure 1. Converting between powers and falling factorial powers.

4.1.1 Laws

Fig. 2 lists the rules for finite differences. First of all, Δ is a *linear operator*: it distributes over sums. The stream $2\ \hat{}\ \mathit{nat}$ is the discrete analogue of e^x as $\Delta\ (2\ \hat{}\ \mathit{nat}) = 2\ \hat{}\ \mathit{nat}$. In general, we have

$$\begin{aligned}\Delta\ (c\ \hat{}\ \mathit{nat})&\\ =\;&\quad \{\ \text{definition of } \Delta\ \}\\ &\mathit{tail}\ (c\ \hat{}\ \mathit{nat}) - c\ \hat{}\ \mathit{nat}\\ =\;&\quad \{\ c \text{ is constant and definition of } \mathit{nat}\ \}\\ &c\ \hat{}\ (\mathit{nat}+1) - c\ \hat{}\ \mathit{nat}\\ =\;&\quad \{\ \text{arithmetic}\ \}\\ &(c-1) * c\ \hat{}\ \mathit{nat}\end{aligned}$$

$$\begin{aligned}\Delta\ (\mathit{tail}\ s) &= \mathit{tail}\ (\Delta\ s)\\ \Delta\ (a \prec s) &= \mathit{head}\ s - a \prec \Delta\ s\\ \Delta\ (s \curlyvee t) &= (t - s) \curlyvee (\mathit{tail}\ s - t)\\ \Delta\ c &= 0\\ \Delta\ (c * s) &= c * \Delta\ s\\ \Delta\ (s + t) &= \Delta\ s + \Delta\ t\\ \Delta\ (s * t) &= s * \Delta\ t + \Delta\ s * \mathit{tail}\ t\\ \Delta\ (c\ \hat{}\ \mathit{nat}) &= (c-1) * c\ \hat{}\ \mathit{nat}\\ \Delta\ (\mathit{nat}^{\underline{n+1}}) &= (\mathit{repeat}\ n+1) * \mathit{nat}^{\underline{n}}\end{aligned}$$

Figure 2. Laws for finite difference.

The product rule is similar to the product rule of infinite calculus except for an occurrence of *tail* on the right-hand side.

$$\begin{aligned}&\Delta\ (s * t)\\ =\;&\quad \{\ \text{definition of } \Delta \text{ and } *\ \}\\ &\mathit{tail}\ s * \mathit{tail}\ t - s * t\\ =\;&\quad \{\ \text{arithmetic}\ \}\\ &s * \mathit{tail}\ t - s * t + \mathit{tail}\ s * \mathit{tail}\ t - s * \mathit{tail}\ t\\ =\;&\quad \{\ \text{distributivity}\ \}\\ &s * (\mathit{tail}\ t - t) + (\mathit{tail}\ s - s) * \mathit{tail}\ t\\ =\;&\quad \{\ \text{definition of } \Delta\ \}\\ &s * \Delta\ t + \Delta\ s * \mathit{tail}\ t\end{aligned}$$

4.1.2 Examples

Let's get back to the Josephus problem: the interactive session in Sec. 4.1 suggests that Δ *jos* is almost always 2, except for pots. We can express this property using a stream conditional.

$$\Delta\ \mathit{jos} = (\mathit{pot}' \to -\mathit{nat}\,;\, 2)$$

where $(\_ \to \_\,;\, \_)$ is **if** _ **then** _ **else** _ lifted to streams (using a ternary version of *zip*). The stream conditional enjoys the standard laws, such as $(\mathit{repeat}\ \mathit{True} \to s\,;\, t) = s$, and a ternary version of the abide law.

$$\begin{aligned}&(s_1 \curlyvee s_2 \to t_1 \curlyvee t_2\,;\, u_1 \curlyvee u_2) =\\ &\quad (s_1 \to t_1\,;\, u_1) \curlyvee (s_2 \to t_2\,;\, u_2)\end{aligned}$$

Both laws are used in the proof of the above property.

$$
\begin{aligned}
& \Delta\ \mathit{jos} \\
=\ & \quad \{\ \Delta \text{ law and arithmetic }\} \\
& 0 \prec 2 \curlyvee 2 * (\mathit{tail}\ \mathit{jos} - \mathit{jos}) - 2 \\
=\ & \quad \{\ \text{definition of } \Delta\ \} \\
& 0 \prec 2 \curlyvee 2 * \Delta\ \mathit{jos} - 2 \\
\subset\ & \quad \{\ x = 0 \prec 2 \curlyvee 2 * x - 2 \text{ has a unique solution }\} \\
& 0 \prec 2 \curlyvee 2 * (\mathit{pot}' \rightarrow -\mathit{nat}\,;\, 2) - 2 \\
=\ & \quad \{\ \text{arithmetic and definition of } \mathit{nat}'\ \} \\
& 0 \prec 2 \curlyvee (\mathit{pot}' \rightarrow -(2 * \mathit{nat}')\,;\, 2) \\
=\ & \quad \{\ \text{definition of } \mathit{nat}, \mathit{pot} \text{ and } \curlyvee\ \} \\
& (\mathit{pot} \rightarrow -(2 * \mathit{nat})\,;\, 2) \curlyvee 2 \\
=\ & \quad \{\ \text{conditional and abide law }\} \\
& (\mathit{pot} \curlyvee \mathit{repeat}\ \mathit{False} \rightarrow -(2 * \mathit{nat} \curlyvee 2 * \mathit{nat} + 1)\,;\, 2 \curlyvee 2) \\
=\ & \quad \{\ \text{definition of } \mathit{pot}' \text{ and characterisation of } \mathit{nat}\ \} \\
& (\mathit{pot}' \rightarrow -\mathit{nat}\,;\, 2)
\end{aligned}
$$

4.2 Summation

Finite difference Δ has a right-inverse: the *summation* operator Σ. We can easily derive its definition.

$$
\begin{aligned}
& \Delta\ (\Sigma\ s) = s \\
\Longleftrightarrow\ & \quad \{\ \text{definition of } \Delta\ \} \\
& \mathit{tail}\ (\Sigma\ s) - \Sigma\ s = s \\
\Longleftrightarrow\ & \quad \{\ \text{arithmetic }\} \\
& \mathit{tail}\ (\Sigma\ s) = \Sigma\ s + s
\end{aligned}
$$

Setting $\mathit{head}\ (\Sigma\ s) = 0$, we obtain

$$
\begin{aligned}
& \Sigma \quad :: (\mathit{Num}\ \alpha) \Rightarrow \mathit{Stream}\ \alpha \rightarrow \mathit{Stream}\ \alpha \\
& \Sigma\ s = t \ \textbf{where}\ t = 0 \prec t + s
\end{aligned}
$$

Here are some examples (A004520, A000290, A011371).

$\gg\ \ \Sigma\ (0 \curlyvee 1)$
$<0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, .. >$
$\gg\ \ \Sigma\ (2 * \mathit{nat} + 1)$
$<0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, .. >$
$\gg\ \ \Sigma\ \mathit{carry}$
$<0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, 11, .. >$

The definition of Σ suggests an unusual approach for determining the sum of a sequence: if we observe that a stream satisfies $t = 0 \prec t + s$, then we may conclude that $\Sigma\ s = t$. For example, $\Sigma\ 1 = \mathit{nat}$ as $\mathit{nat} = 0 \prec \mathit{nat} + 1$. This is *summation by happenstance*. Of course, if we already know the sum, we can use the definition to verify our conjecture. As an example, let us prove $\Sigma\ \mathit{fib} = \mathit{fib}' - 1$.

$$
\begin{aligned}
& \mathit{fib}' - 1 \\
=\ & \quad \{\ \text{definition of } \mathit{fib}'\ \} \\
& (1 \prec \mathit{fib}' + \mathit{fib}) - 1 \\
=\ & \quad \{\ \text{arithmetic }\} \\
& 0 \prec (\mathit{fib}' - 1) + \mathit{fib}
\end{aligned}
$$

The unique fixed-point proof avoids the inelegant case analysis of an inductive proof.

4.2.1 Laws

The *Fundamental Theorem of finite calculus* relates Δ and Σ.

$$t = \Delta\ s \quad \Longleftrightarrow \quad \Sigma\ t = s - \mathit{repeat}\ (\mathit{head}\ s)$$

$$
\begin{aligned}
\Sigma\ (\mathit{tail}\ s) &= \mathit{tail}\ (\Sigma\ s) - \mathit{repeat}\ (\mathit{head}\ s) \\
\Sigma\ (a \prec s) &= 0 \prec \mathit{repeat}\ a + \Sigma\ s \\
\Sigma\ (s \curlyvee t) &= (\Sigma\ s + \Sigma\ t) \curlyvee (s + \Sigma\ s + \Sigma\ t) \\
\Sigma\ c &= c * \mathit{nat} \\
\Sigma\ (c * s) &= c * \Sigma\ s \\
\Sigma\ (s + t) &= \Sigma\ s + \Sigma\ t \\
\Sigma\ (s * \Delta\ t) &= s * t - \Sigma\ (\Delta\ s * \mathit{tail}\ t) - \mathit{head}\ (s * t) \\
\Sigma\ (c\ \hat{}\ \mathit{nat}) &= (c\ \hat{}\ \mathit{nat} - 1) \mathbin{/} (c - 1) \\
\Sigma\ (\mathit{nat}^{\underline{n}}) &= \mathit{nat}^{\underline{n+1}} \mathbin{/} (\mathit{repeat}\ n + 1)
\end{aligned}
$$

Figure 3. Laws for summation.

The implication from right to left is easy to show using $\Delta\ (\Sigma\ t) = t$ and $\Delta\ c = 0$. For the reverse direction, we reason

$$
\begin{aligned}
& \Sigma\ (\Delta\ s) \\
=\ & \quad \{\ \text{definition of } \Sigma\ \} \\
& 0 \prec \Sigma\ (\Delta\ s) + \Delta\ s \\
\subset\ & \quad \{\ x = 0 \prec x + \Delta\ s \text{ has a unique solution }\} \\
& 0 \prec s - \mathit{repeat}\ (\mathit{head}\ s) + \Delta\ s \\
=\ & \quad \{\ \text{definition of } \Delta \text{ and arithmetic }\} \\
& (\mathit{head}\ s \prec \mathit{tail}\ s) - \mathit{repeat}\ (\mathit{head}\ s) \\
=\ & \quad \{\ \text{extensionality }\} \\
& s - \mathit{repeat}\ (\mathit{head}\ s)
\end{aligned}
$$

Using the Fundamental Theorem we can transform the rules in Fig. 2 into rules for summation, see Fig. 3. As an example, the rule for products, *summation by parts*, can be derived from the product rule of Δ. Let $c = \mathit{repeat}\ (\mathit{head}\ (s * t))$, then

$$
\begin{aligned}
& s * \Delta\ t + \Delta\ s * \mathit{tail}\ t = \Delta\ (s * t) \\
\Longleftrightarrow\ & \quad \{\ \text{Fundamental Theorem }\} \\
& \Sigma\ (s * \Delta\ t + \Delta\ s * \mathit{tail}\ t) = s * t - c \\
\Longleftrightarrow\ & \quad \{\ \Sigma \text{ is linear }\} \\
& \Sigma\ (s * \Delta\ t) + \Sigma\ (\Delta\ s * \mathit{tail}\ t) = s * t - c \\
\Longleftrightarrow\ & \quad \{\ \text{arithmetic }\} \\
& \Sigma\ (s * \Delta\ t) = s * t - \Sigma\ (\Delta\ s * \mathit{tail}\ t) - c
\end{aligned}
$$

Unlike the others, this law is not compositional: $\Sigma\ (s * t)$ is not given in terms of $\Sigma\ s$ and $\Sigma\ t$, a situation familiar from calculus.

The only slightly tricky derivation is the one for interleaving.

$$
\begin{aligned}
& (t - s) \curlyvee (\mathit{tail}\ s - t) = \Delta\ (s \curlyvee t) \\
=\ & \quad \{\ \text{Fundamental Theorem and } \mathit{head}\ (s \curlyvee t) = \mathit{head}\ s\ \} \\
& \Sigma\ ((t - s) \curlyvee (\mathit{tail}\ s - t)) = (s \curlyvee t) - \mathit{repeat}\ (\mathit{head}\ s)
\end{aligned}
$$

At first glance, we are stuck. To make progress, let's introduce two fresh variables: $x = t - s$ and $y = \mathit{tail}\ s - t$. If we can express s and t in terms of x and y, then we have found the desired formula.

$$
\begin{aligned}
& t - s = x \ \text{ and }\ \mathit{tail}\ s - t = y \\
\Longleftrightarrow\ & \quad \{\ \text{arithmetic }\} \\
& \mathit{tail}\ s - s = x + y \ \text{ and }\ t = x + s \\
\Longleftrightarrow\ & \quad \{\ \text{definition of } \Delta\ \} \\
& \Delta\ s = x + y \ \text{ and }\ t = x + s \\
\Longleftarrow\ & \quad \{\ \Delta\ (\Sigma\ s) = s\ \} \\
& s = \Sigma\ x + \Sigma\ y \ \text{ and }\ t = x + \Sigma\ x + \Sigma\ y
\end{aligned}
$$

Since $\mathit{head}\ s = 0$, the interleaving rule follows.

4.2.2 Examples

Using the rules in Fig. 3 we can mechanically calculate summations of polynomials. The main effort goes into converting between ordinary and falling factorial powers. Here is a formula for the sum of the first n squares, the *square pyramidal numbers* ($0 \prec \mathsf{A000330}$).

$$
\begin{aligned}
& \Sigma\ (nat \mathbin{\hat{}} 2) \\
= \quad & \{ \text{ converting to falling factorial powers } \} \\
& \Sigma\ (nat^{\underline{2}} + nat^{\underline{1}}) \\
= \quad & \{ \text{ summation laws } \} \\
& \tfrac{1}{3} * nat^{\underline{3}} + \tfrac{1}{2} * nat^{\underline{2}} \\
= \quad & \{ \text{ converting to ordinary powers } \} \\
& \tfrac{1}{3} * (nat \mathbin{\hat{}} 3 - 3 * nat \mathbin{\hat{}} 2 + 2 * nat) + \tfrac{1}{2} * (nat \mathbin{\hat{}} 2 - nat) \\
= \quad & \{ \text{ arithmetic } \} \\
& \tfrac{1}{6} * (nat - 1) * nat * (2 * nat - 1)
\end{aligned}
$$

Calculating the summation of a product, say, $\Sigma\ (nat * c \mathbin{\hat{}} nat)$ is often more involved. Recall that the rule for products, *summation by parts*, is imperfect: to be able to apply it, we have to spot a difference among the factors. In the example above, there is an obvious candidate: $c \mathbin{\hat{}} nat$. Let's see how it goes.

$$
\begin{aligned}
& \Sigma\ (nat * c \mathbin{\hat{}} nat) \\
= \quad & \{\ \Delta\ (c \mathbin{\hat{}} nat) = (c - 1) * c \mathbin{\hat{}} nat\ \} \\
& \Sigma\ (nat * \Delta\ (c \mathbin{\hat{}} nat)\ /\ (c - 1)) \\
= \quad & \{\ \Sigma \text{ is linear } \} \\
& \Sigma\ (nat * \Delta\ (c \mathbin{\hat{}} nat))\ /\ (c - 1) \\
= \quad & \{ \text{ summation by parts } \} \\
& (nat * c \mathbin{\hat{}} nat - \Sigma\ (\Delta\ nat * tail\ (c \mathbin{\hat{}} nat)))\ /\ (c - 1) \\
= \quad & \{\ \Delta\ nat = 1, c \text{ constant, and definition of } nat\ \} \\
& (nat * c \mathbin{\hat{}} nat - c * \Sigma\ (c \mathbin{\hat{}} nat))\ /\ (c - 1) \\
= \quad & \{ \text{ summation law } \} \\
& (nat * c \mathbin{\hat{}} nat - c * (c \mathbin{\hat{}} nat - 1)\ /\ (c - 1))\ /\ (c - 1) \\
= \quad & \{ \text{ arithmetic } \} \\
& (((c - 1) * nat - c) * c \mathbin{\hat{}} nat + c)\ /\ (c - 1) \mathbin{\hat{}} 2
\end{aligned}
$$

That wasn't too hard.

As a final example, let us tackle a sum that involves interleaving: $\Sigma\ carry$ ($\mathsf{A011371}$). The sum is important as it determines the amortised running time of the binary increment. Going back to the interactive session, Sec. 4.2, we observe that the sum is always less than or equal to *nat*, which would imply that the amortised running time is constant. That's nice, but can we actually quantify the difference? Let's approach the problem from a different angle. The binary increment changes the number of 1s, so we might hope to relate *carry* to *ones*. The increment flips the trailing 1s to 0s and flips the first 0 to 1. Now, since *carry* defines the number of trailing 0s, we obtain the following alternative definition of *ones*.

$$ones = 0 \prec ones + 1 - carry$$

We omit the proof that both definitions are indeed equal. (If you want to try, use a $\subset$-proof.) Now, we can invoke the *summation by happenstance* rule.

$$
\begin{aligned}
& ones = 0 \prec ones + (1 - carry) \\
\iff \quad & \{ \text{ summation by happenstance } \} \\
& \Sigma\ (1 - carry) = ones \\
\iff \quad & \{ \text{ arithmetic } \} \\
& \Sigma\ carry = nat - ones
\end{aligned}
$$

Voilà. We have found a closed form for $\Sigma\ carry$.

That was fun. But surely, the interleaving rule in Fig. 3 would yield the result directly, wouldn't it? Let's try.

$$
\begin{aligned}
& \Sigma\ carry \\
= \quad & \{ \text{ definition of } carry\ \} \\
& \Sigma\ (0 \curlyvee carry + 1) \\
= \quad & \{ \text{ summation law } \} \\
& \Sigma\ (carry + 1) \curlyvee \Sigma\ (carry + 1) \\
= \quad & \{\ \Sigma \text{ is linear and } \Sigma\ 1 = nat\ \} \\
& (\Sigma\ carry + nat) \curlyvee (\Sigma\ carry + nat) \\
= \quad & \{ \text{ abide law } \} \\
& (\Sigma\ carry \curlyvee \Sigma\ carry) + (nat \curlyvee nat)
\end{aligned}
$$

That's quite a weird property. Since we know where we are aiming at, let us determine $nat - \Sigma\ carry$.

$$
\begin{aligned}
& nat - \Sigma\ carry \\
= \quad & \{ \text{ property of nat and } \Sigma\ carry\ \} \\
& (2 * nat \curlyvee 2 * nat + 1) - ((\Sigma\ carry \curlyvee \Sigma\ carry) + (nat \curlyvee nat)) \\
= \quad & \{ \text{ arithmetic } \} \\
& (nat - \Sigma\ carry) \curlyvee (nat - \Sigma\ carry) + 1
\end{aligned}
$$

Voilà again. The sequence $nat - \Sigma\ carry$ satisfies $x = x \curlyvee x + 1$, which implies that $nat - \Sigma\ carry = ones$. For the sake of completeness, we should also check that $head\ ones = head\ (nat - \Sigma\ carry)$, which is indeed the case.

4.2.3 Perturbation method

The Fundamental Theorem has another easy consequence, which is the basis of the *perturbation method*. Setting $t = tail\ s - s$ and applying the theorem from left to right we obtain

$$\Sigma\ s \quad = \quad \Sigma\ (tail\ s) - s + repeat\ (head\ s)$$

The idea of the method is to try to express $\Sigma\ (tail\ s)$ in terms of $\Sigma\ s$. Then we obtain an equation whose solution is the sum we seek. Let's try the method on a sum we have done before.

$$
\begin{aligned}
& \Sigma\ (nat * c \mathbin{\hat{}} nat) \\
= \quad & \{ \text{ perturbation, } head\ (nat * c \mathbin{\hat{}} nat) = 0\ \} \\
& \Sigma\ (tail\ (nat * c \mathbin{\hat{}} nat)) - nat * c \mathbin{\hat{}} nat \\
= \quad & \{ \text{ definition of } nat\ \} \\
& \Sigma\ ((nat + 1) * c \mathbin{\hat{}} (nat + 1)) - nat * c \mathbin{\hat{}} nat \\
= \quad & \{ \text{ summation law } \} \\
& c * \Sigma\ (nat * c \mathbin{\hat{}} nat) + c * \Sigma\ (c \mathbin{\hat{}} nat) - nat * c \mathbin{\hat{}} nat \\
= \quad & \{ \text{ summation law } \} \\
& c * \Sigma\ (nat * c \mathbin{\hat{}} nat) + c * (c \mathbin{\hat{}} nat - 1)\ /\ (c - 1) - \\
& \quad nat * c \mathbin{\hat{}} nat
\end{aligned}
$$

The sum $\Sigma\ (nat * c \mathbin{\hat{}} nat)$ appears again on the right-hand side. All that is left to do is to solve the resulting equation, which yields the result we have seen in Sec. 4.2.2.

As an aside, the perturbation method also suggests an alternative definition of Σ, this time as a second-order fixed point.

$$\Sigma\ s = 0 \prec repeat\ (head\ s) + \Sigma\ (tail\ s)$$

The code implements the naïve way of summing: the i-th element is computed using i additions not reusing any previous results.

5. Generating functions ($\times$, $\div$)

In this section, we look at number sequences from a different perspective: we take the view that a sequence, $a_0, a_1, a_2 \ldots$, represents a power series, $a_0 + a_1 z + a_2 z^2 + \cdots$, in some formal variable z. It's an alternative view and we shall see that it provides us with additional operators and techniques for manipulating streams.

5.1 Power series

Let's put on the 'power series' glasses. The simplest series, the constant function a_0 and the identity z (A063524), are given by

$$\begin{array}{l} \mathit{const} \;\; :: (\mathit{Num}\ \alpha) \Rightarrow \alpha \to \mathit{Stream}\ \alpha \\ \mathit{const}\ n = n \prec \mathit{repeat}\ 0 \\ z :: (\mathit{Num}\ \alpha) \Rightarrow \mathit{Stream}\ \alpha \\ z = 0 \prec 1 \prec \mathit{repeat}\ 0 \end{array}$$

The sum of two power series is implemented by $+$. The successor function, for instance, is $\mathit{const}\ 1 + z$. The product of two series, however, is not given by $*$ since, for example, $(\mathit{const}\ 1 + z) * (\mathit{const}\ 1 + z) = \mathit{const}\ 1 + z$. So, let us introduce a new operator for the product of two series, say, $\times$ and derive its implementation. The point of departure is *Horner's rule* for evaluating a polynomial, rephrased as an identity on streams.

$$s \;=\; \mathit{const}\ (\mathit{head}\ s) + z \times \mathit{tail}\ s$$

The rule implies $z \times s = 0 \prec s$. In other words, multiplying by z amounts to prepending 0. The derivation of $\times$ proceeds as follows (we abbreviate *head*, *tail* and *const*).

$$\begin{array}{ll} & s \times t \\ = & \quad \{ \text{Horner's rule} \} \\ & (\mathit{con}\ (\mathit{hd}\ s) + z \times \mathit{tl}\ s) \times t \\ = & \quad \{ \text{arithmetic} \} \\ & \mathit{con}\ (\mathit{hd}\ s) \times t + z \times \mathit{tl}\ s \times t \\ = & \quad \{ \text{Horner's rule} \} \\ & \mathit{con}\ (\mathit{hd}\ s) \times (\mathit{con}\ (\mathit{hd}\ t) + z \times \mathit{tl}\ t) + z \times \mathit{tl}\ s \times t \\ = & \quad \{ \text{arithmetic} \} \\ & \mathit{con}\ (\mathit{hd}\ s) \times \mathit{con}\ (\mathit{hd}\ t) + \mathit{con}\ (\mathit{hd}\ s) \times z \times \mathit{tl}\ t + z \times \mathit{tl}\ s \times t \\ = & \quad \{ \mathit{con}\ a \times \mathit{con}\ b = \mathit{con}\ (a * b) \text{ and arithmetic} \} \\ & \mathit{con}\ (\mathit{hd}\ s * \mathit{hd}\ t) + z \times (\mathit{con}\ (\mathit{hd}\ s) \times \mathit{tl}\ t + \mathit{tl}\ s \times t) \\ = & \quad \{ \text{Horner's rule} \} \\ & \mathit{hd}\ s * \mathit{hd}\ t \prec \mathit{con}\ (\mathit{hd}\ s) \times \mathit{tl}\ t + \mathit{tl}\ s \times t \end{array}$$

The first line jointly with the last one serves as a perfectly valid implementation. However, $\times$ is a costly operation; we can improve the efficiency somewhat if we replace $\mathit{const}\ k \times s$ by $\mathit{repeat}\ k * s$ (this law also follows from Horner's rule).

$$\begin{array}{l} \textbf{infixl}\ 7\ \times \\ (\times) \;\; :: (\mathit{Num}\ \alpha) \Rightarrow \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \\ s \times t = \mathit{head}\ s * \mathit{head}\ t \prec \mathit{repeat}\ (\mathit{head}\ s) * \mathit{tail}\ t + \mathit{tail}\ s \times t \end{array}$$

Here are some examples (A014824, $0 \prec$ A099670, *tail* A002275).

$$\begin{array}{l} \gg\ \ \mathit{nat} \times 10\ \hat{}\ \mathit{nat} \\ <0, 1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, ..> \\ \gg\ \ 9 * (\mathit{nat} \times 10\ \hat{}\ \mathit{nat}) \\ <0, 9, 108, 1107, 11106, 111105, 1111104, 11111103, ..> \\ \gg\ \ 9 * (\mathit{nat} \times 10\ \hat{}\ \mathit{nat}) + \mathit{nat}' \\ <1, 11, 111, 1111, 11111, 111111, 1111111, 11111111, ..> \end{array}$$

The operator $\times$ is also called *convolution product*. The first example suggests how it works: the product of $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ is $a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + a_1 b_1 + a_2 b_0, \ldots$.

Let us complete our repertoire of arithmetic operators with reciprocal and division. Convolution was a little more complicated than the other operations, so it is wise to derive reciprocal from a specification (note that *recip* is yet another class method).

$$s \times \mathit{recip}\ s \;=\; \mathit{const}\ 1$$

We reason

$$\begin{array}{ll} & \mathit{head}\ s * \mathit{head}\ (\mathit{recip}\ s) = 1 \\ \iff & \quad \{ \text{arithmetic} \} \\ & \mathit{head}\ (\mathit{recip}\ s) = \mathit{recip}\ (\mathit{head}\ s) \end{array}$$

and

$$\begin{array}{ll} & \mathit{const}\ (\mathit{head}\ s) \times \mathit{tail}\ (\mathit{recip}\ s) + \mathit{tail}\ s \times \mathit{recip}\ s = 0 \\ \iff & \quad \{ \text{arithmetic} \} \\ & -\mathit{const}\ (\mathit{head}\ s) \times \mathit{tail}\ (\mathit{recip}\ s) = \mathit{tail}\ s \times \mathit{recip}\ s \\ \iff & \quad \{ \text{arithmetic} \} \\ & \mathit{tail}\ (\mathit{recip}\ s) = \mathit{const}\ (-\mathit{recip}\ (\mathit{head}\ s)) \times \mathit{tail}\ s \times \mathit{recip}\ s \end{array}$$

Again replacing $\mathit{const}\ k \times s$ by $\mathit{repeat}\ k * s$, we obtain

$$\begin{array}{l} \mathit{recip}\ s = t\ \textbf{where}\ a = \mathit{recip}\ (\mathit{head}\ s) \\ \qquad\qquad\qquad\quad\;\, t = a \prec \mathit{repeat}\ (-a) * (\mathit{tail}\ s \times t) \\ \textbf{infixl}\ 7\ \div \\ s \div t = s \times \mathit{recip}\ t \end{array}$$

Finally, we use s^n, where n is a natural number, for iterated convolution and set $s^{-n} = (\mathit{recip}\ s)^n$.

5.2 Laws

The familiar arithmetic laws also hold for $\mathit{const}\ n$, $+$, $-$, $\times$ and $\div$. Perhaps surprisingly, we can reformulate the streams we introduced so far in terms of these operators. In other words, we view them with our new 'power series' glasses. Mathematically speaking, this conversion corresponds to finding the *generating function* of a sequence. The good news is that we need not leave our stamping ground: everything can be accomplished within the world of streams. The only caveat is that we have to be careful not to confuse $\mathit{const}\ n$ and $\times$ with $\mathit{repeat}\ n$ and $*$.

As a start, let's determine the generating function for $\mathit{repeat}\ a$.

$$\begin{array}{ll} & \mathit{repeat}\ a = a \prec \mathit{repeat}\ a \\ \iff & \quad \{ \text{Horner's rule} \} \\ & \mathit{repeat}\ a = \mathit{const}\ a + z \times \mathit{repeat}\ a \\ \iff & \quad \{ \text{arithmetic} \} \\ & \mathit{const}\ 1 \times \mathit{repeat}\ a - z \times \mathit{repeat}\ a = \mathit{const}\ a \\ \iff & \quad \{ \text{arithmetic} \} \\ & (\mathit{const}\ 1 - z) \times \mathit{repeat}\ a = \mathit{const}\ a \\ \iff & \quad \{ \text{arithmetic} \} \\ & \mathit{repeat}\ a = \mathit{const}\ a \div (\mathit{const}\ 1 - z) \end{array}$$

The form of the resulting equation, $s = u \div (\mathit{const}\ 1 - v)$, is quite typical reflecting the shape of streams equations, $s = h \prec t$.

Geometric sequences are not much harder.

$$\begin{array}{ll} & \mathit{repeat}\ a\ \hat{}\ \mathit{nat} \\ = & \quad \{ \text{definition of } \hat{}\ \text{and } \mathit{nat} \} \\ & 1 \prec \mathit{repeat}\ a * \mathit{repeat}\ a\ \hat{}\ \mathit{nat} \\ = & \quad \{ \text{Horner's rule and } \mathit{repeat}\ k * s = \mathit{const}\ k \times s \} \\ & \mathit{const}\ 1 + z \times \mathit{const}\ a \times \mathit{repeat}\ a\ \hat{}\ \mathit{nat} \end{array}$$

Consequently, $\mathit{repeat}\ a\ \hat{}\ \mathit{nat} = \mathit{const}\ 1 \div (1 - \mathit{const}\ a \times z)$.

We can even derive a formula for the sum of a sequence.

$$\begin{array}{ll} & \Sigma\ s = 0 \prec \Sigma\ s + s \\ \iff & \quad \{ \text{Horner's rule} \} \\ & \Sigma\ s = z \times (\Sigma\ s + s) \\ \iff & \quad \{ \text{arithmetic} \} \\ & \Sigma\ s = s \times z \div (\mathit{const}\ 1 - z) \end{array}$$

$$\begin{array}{lcl} \textit{const}\ k \times s & = & \textit{repeat}\ k * s \\ \textit{repeat}\ a & = & \textit{const}\ a \div (\textit{const}\ 1 - z) \\ \textit{repeat}\ a\ \hat{}\ \textit{nat} & = & \textit{const}\ 1 \div (1 - \textit{const}\ a \times z) \\ \Sigma\ s & = & s \times z \div (\textit{const}\ 1 - z) \\ \textit{nat} & = & z \div (\textit{const}\ 1 - z)^2 \end{array}$$

Figure 4. Laws for generating functions.

This implies that the generating function of the natural numbers is $\textit{nat} = \Sigma\ (\textit{repeat}\ 1) = z \div (\textit{const}\ 1 - z)^2$. Fig. 4 summarises our findings.

Of course, there is no reason for jubilation: the formula for the sum does not immediately provide us with a closed form for the *coefficients* of the generating function. In fact, to be able to read off the coefficients, we have to reduce the generating function to a known stream, for instance, *repeat* a, *nat* or *repeat* $a\ \hat{}\ \textit{nat}$. This is what we do next.

5.3 Solving recurrences

Let's try to find a closed form for our all-time favourite, the Fibonacci sequence. As a first step, we determine the generating function of *fib*, that is, we express *fib* in terms of $\times$ and friends.

$$\begin{array}{ll} & \textit{fib} = 0 \prec \textit{fib} + (1 \prec \textit{fib}) \\ = & \quad \{\ \text{Horner's rule}\ \} \\ & \textit{fib} = z \times (\textit{fib} + (\textit{const}\ 1 + z \times \textit{fib})) \\ = & \quad \{\ \text{arithmetic}\ \} \\ & \textit{fib} = z \div (\textit{const}\ 1 - z - z^2) \end{array}$$

Now, to find a closed form for *fib* we have to turn the right-hand side into a generating function or a sum of generating functions whose coefficients we know. The following algebraic identity points us into the right direction ($\alpha \neq \beta$).

$$\frac{x}{(1-\alpha x)(1-\beta x)} = \frac{1}{\alpha - \beta}\left(\frac{1}{1-\alpha x} - \frac{1}{1-\beta x}\right)$$

Inspecting Fig. 4 we realize that we know the stream expression for the right-hand side:

$$\textit{repeat}\ (1\ /\ \alpha - \beta) * (\textit{repeat}\ \alpha\ \hat{}\ \textit{nat} - \textit{repeat}\ \beta\ \hat{}\ \textit{nat})$$

So, we are left with the task of transforming $1 - z - z^2$ into the form $(1 - \alpha z)(1 - \beta z)$. It turns out that the roots of $z^2 - z - 1$ are the reciprocals of the roots of $1 - z - z^2$. (The trick of reversing the coefficients works in general, see (Graham et al. 1994, p. 339).) A quick calculation shows that $\phi = \frac{1}{2}(1 + \sqrt{5})$, the golden ratio $\frac{a+b}{a} = \frac{a}{b}$, and $\hat{\phi} = \frac{1}{2}(1 - \sqrt{5})$ are the roots we seek. Consequently, $1 - z - z^2 = (1 - \phi z)(1 - \hat{\phi} z)$. Since furthermore $\phi - \hat{\phi} = \sqrt{5}$, we have inferred that

$$\textit{fib} = \textit{repeat}\ (1\ /\ \textit{sqrt}\ 5) * (\textit{repeat}\ \phi\ \hat{}\ \textit{nat} - \textit{repeat}\ \hat{\phi}\ \hat{}\ \textit{nat})$$

A noteworthy feature of the derivation is that is stays within the world of streams. For the general theory of solving recurrences, we refer the interested reader to Graham et al. (1994, Sec. 7.3).

6. Related work

The two major sources of inspiration were Rutten's work on stream calculus (Rutten 2003, 2005) and the text book on concrete mathematics (Graham et al. 1994). Rutten introduces streams and stream operators using coinductive definitions. which he calls *behavioural differential equations*. As an example, the Haskell definition of sum

$$s + t = \textit{head}\ s + \textit{head}\ t \prec \textit{tail}\ s + \textit{tail}\ t$$

translates to

$$(s + t)(0) = s(0) + t(0) \quad \text{and} \quad (s + t)' = s' + t'$$

where $s(0)$ denotes the head of s, its initial value, and s' the tail of s, its stream derivative. (The notation goes back to Hoare.) Rutten relies on coinduction as the main proof technique and emphasises the 'power series' view of streams. In fact, we have given power series and generating functions only a cursory treatment as there are already a number of papers on that subject, most notably, (Karczmarczuk 1997; McIlroy 1999, 2001). Both Karczmarczuk and McIlroy mention the proof technique of unique fixed points in passing by: Karczmarczuk sketches a proof of $\textit{iterate}\ f \cdot f = \textit{map}\ f \cdot \textit{iterate}\ f$ and McIlroy shows $1/e^x = e^{-x}$.

Various proof methods for corecursive programs are discussed by Gibbons and Hutton (2005). Interestingly, the technique of unique fixed points is not among them.[2] Unique fixed-point proofs are closely related to the principle of *guarded induction* (Coquand 1994). Loosely speaking, the guarded condition ensures that functions are productive by restricting the context of a recursive call to one ore more constructors. For instance,

$$\textit{nat} = 1 \prec \textit{nat} + 1$$

is not guarded as $+$ is not a constructor. However, *nat* can be defined by *iterate* $(+1)\ 0$ as *iterate* is guarded. The proof method then allows us to show that *iterate* $(+1)\ 0$ is the unique solution of $x = x \prec x + 1$ by constructing a suitable proof transformer using guarded equations. Indeed, the central idea underlying guarded induction is to express proofs as lazy functional programs.

7. Conclusion

I hope you enjoyed the journey. Lazy functional programming has proven its worth: with a couple of one-liners we have hacked, eerh, built a small domain-specific language for manipulating infinite sequences. Suitably restricted, stream equations possess unique fixed points, a property that can be exploited to redevelop the theory of recurrences, finite calculus and generating functions.

Acknowledgments

A big thank you to Jeremy Gibbons for improving my English. Thanks are also due to Nils Anders Danielsson and the anonymous referees for pointing out several typos.

A. Proof of existence and uniqueness of solutions

This appendix reproduces the proof of existence and uniqueness of solutions (Rutten 2003). It has been rephrased in familiar programming language terms to make it accessible to a wider audience.

A.1 Coalgebras

There are many data types that support *head* and *tail* operations. So, let's turn the two functions into class methods

$$\begin{array}{l} \textbf{class}\ \textit{Coalgebra}\ \sigma\ \textbf{where} \\ \quad \textit{head} :: \sigma\ \alpha \to \alpha \\ \quad \textit{tail}\ \ :: \sigma\ \alpha \to \sigma\ \alpha \end{array}$$

with *Stream* an obvious instance of this class. We call an element of $\sigma\ \tau$, where σ is an instance of *Coalgebra*, a *stream-like value*.

[2] The minutes of the 2003 Meetings of the Algebra of Programming Research Group, 21st November, seem to suggest that the authors were aware of the technique, but were not sure of constraints on applicability, see `http://www.comlab.ox.ac.uk/research/pdt/ap/minutes/minutes2003.html#21nov`.

A.2 Coinduction

If we are given two stream-like values, not necessarily of the same type, then we can relate them by studying their *behaviour*: do they yield the same head and are the tails related, as well?

$$a\ R\ b \quad \Longrightarrow \quad \mathit{head}\ a = \mathit{head}\ b \ \text{ and } \ (\mathit{tail}\ a)\ R\ (\mathit{tail}\ b)$$

A relation R that satisfies this property is called a *bisimulation*. (In Haskell, products are lifted so the definition of a bisimulation is actually more involved. We simply ignore this complication here.) Bisimulations are closed under union. The greatest bisimulation, written $\sim$, is the union of all bisimulations.

$$\sim \quad = \quad \bigcup \{ R \mid R \text{ is a bisimulation} \}$$

Bisimulations are also closed under relational converse and relational composition. In particular, $a \sim b$ implies $b \sim a$; furthermore, $a \sim b$ and $b \sim c$ imply $a \sim c$.

If $\sim$ relates elements of the same type, it is called the *bisimilarity* relation. In this case, because $=$ is a bisimulation, $\sim$ is an equivalence relation. Streams are a special coalgebra: if two streams behave the same, then they *are* the same. This is captured by the

Theorem 1 (Coinduction proof principle) *Let $s, t \in \mathit{Stream}\ \tau$, then s and t are bisimilar iff they are equal.*

$$s \sim t \quad \Longleftrightarrow \quad s = t$$

PROOF. $\Longleftarrow$: trivial as $=$ is a bisimulation. $\Longrightarrow$: This direction can be shown with the Approximation Lemma (Gibbons and Hutton 2005) using the fact that $\sim$ is a bisimulation. □

Let us illustrate the coinduction proof principle with a simple example: $s = \mathit{head}\ s \prec \mathit{tail}\ s$. Let $R = (=) \cup \{ (s, \mathit{head}\ s \prec \mathit{tail}\ s) \mid s \in \mathit{Stream}\ \tau \}$. We show that R is a bisimulation. **Case** $s\ R\ s$: trivial since $=$ is a bisimulation. **Case** $s\ R\ (\mathit{head}\ s \prec \mathit{tail}\ s)$: the *head* and the *tail* of both streams are, in fact, identical. Since $= \subseteq R$, this implies $(\mathit{tail}\ s)\ R\ (\mathit{tail}\ (\mathit{head}\ s \prec \mathit{tail}\ s))$, as desired.

A.3 The operator *unfold*

A stream-like value can be converted into a real stream using

$$\begin{aligned} &\mathit{unfold} \ :: \ (\mathit{Coalgebra}\ \sigma) \Rightarrow \sigma\ \alpha \to \mathit{Stream}\ \alpha \\ &\mathit{unfold}\ s = \mathit{head}\ s \prec \mathit{unfold}\ (\mathit{tail}\ s) \end{aligned}$$

From the definition of *unfold* we can derive the following laws.

$$\begin{aligned} \mathit{head} \cdot \mathit{unfold} \quad &= \quad \mathit{head} \\ \mathit{tail} \cdot \mathit{unfold} \quad &= \quad \mathit{unfold} \cdot \mathit{tail} \end{aligned}$$

In fact, *unfold* is the *unique* solution of these equations.

Lemma 1 *unfold is a functional bisimulation.*

$$a \quad \sim \quad \mathit{unfold}\ a$$

PROOF. Using the properties of *unfold* it is straightforward to show that $R = \{ (a, \mathit{unfold}\ a) \mid a \in \sigma\ \tau \}$ is a bisimulation. □

Lemma 2 *Two elements are related by $\sim$ iff they evaluate to the same stream.*

$$a_1 \sim a_2 \quad \Longleftrightarrow \quad \mathit{unfold}\ a_1 = \mathit{unfold}\ a_2$$

PROOF. $\Longrightarrow$: We reason

$$\begin{aligned} & a_1 \sim a_2 \\ \Longrightarrow \quad & \{ \text{Lemma 1 and} \sim \text{is symmetric and transitive} \} \\ & \mathit{unfold}\ a_1 \sim \mathit{unfold}\ a_2 \\ \Longleftrightarrow \quad & \{ \text{Coinduction} \} \\ & \mathit{unfold}\ a_1 = \mathit{unfold}\ a_2 \end{aligned}$$

$\Longleftarrow$: We show that $R = \{ (a_1, a_2) \mid \mathit{unfold}\ a_1 = \mathit{unfold}\ a_2 \}$ is a bisimulation. This follows from the properties of *unfold*. □

A.4 Syntactic streams

The central idea underlying the proof is to recast streams and stream operators as interpreters that operate on *syntactic representations* of streams. As a first step, let us define a data type of stream expressions (we list only a few representative examples).

```
data Expr :: * → * where
  Var    :: Stream α → Expr α
  Repeat :: α → Expr α
  Plus   :: (Num α) ⇒ Expr α → Expr α → Expr α
  Nat    :: Expr Integer
```

The definition makes use of a recent extension of Haskell, called *generalised algebraic data types*. The type argument of *Expr* specifies the type of the elements of the stream represented. If we replace *Expr* by *Stream* in the signatures above, we obtain the original types of *repeat*, $+$ and *nat*. The only extra constructor is *Var*, which allows us to embed a stream into a stream expression.

We turn *Expr* into a coalgebra by transforming the stream equations into definitions for *head* and *tail*: $s = h \prec t$ becomes $\mathit{head}\ s = h$ and $\mathit{tail}\ s = \hat{t}$ where $\hat{t}$ is t with *repeat*, $+$ and *nat* replaced by the corresponding constructors *Repeat*, *Plus* and *Nat*.

```
instance Coalgebra Expr where
  head (Var s)         = head s
  head (Repeat a)      = a
  head (Plus e1 e2)    = head e1 + head e2
  head Nat             = 0
  tail (Var s)         = Var (tail s)
  tail (Repeat a)      = Repeat a
  tail (Plus e1 e2)    = Plus (tail e1) (tail e2)
  tail Nat             = Plus Nat (Repeat 1)
```

Both *head* and *tail* are given by simple inductive definitions. In fact, the restrictions on stream equations, detailed in Sec. 2.2, are chosen in order to guarantee this property! In particular, *head* and *tail* may only be invoked on the arguments of a stream operator.

Using *unfold* we can evaluate a stream expression into a stream.

```
eval :: Expr α → Stream α
eval = unfold
```

Furthermore, using *eval* alias *unfold* we can define the streams and stream operators in terms of their syntactic counterparts.

```
repeat k    = eval (Repeat k)
plus s1 s2  = eval (Plus (Var s1) (Var s2))
nat         = eval Nat
```

For *plus*, we embed the argument streams using *Var* and then evaluate the resulting expression. We claim that these definitions satisfy the original stream equations (App. A.5) and furthermore that they are the unique solutions (App. A.6).

A.5 Existence of solutions

When we turned the stream equations into definitions for *head* and *tail*, we replaced functions by constructors. In order to prove that the stream equations are satisfied, we have to show that *eval* undoes this conversion step replacing constructors by functions. In other words, we have to show that *eval* is an interpreter. Working towards this goal we first prove that $\sim$ is a congruence relation.

Lemma 3 *$\sim$ is a congruence relation on expressions.*

$$t_1 \sim u_1 \ \text{ and } \ t_2 \sim u_2 \quad \Longrightarrow \quad \mathit{Plus}\ t_1\ t_2 \sim \mathit{Plus}\ u_1\ u_2$$

PROOF. Let R be given by the following inductive definition.

$$\mathsf{R} \quad = \quad \sim \cup \{ (\mathit{Plus}\ t_1\ t_2, \mathit{Plus}\ u_1\ u_2) \mid t_1\ \mathsf{R}\ u_1 \text{ and } t_2\ \mathsf{R}\ u_2 \}$$

Note that R is a congruence relation by construction, indeed, the smallest congruence containing $\sim$. We show that R is a bisimulation by induction over its definition. **Case** $t \sim u$: trivial. **Case** $(\mathit{Plus}\ t_1\ t_2)\ \mathsf{R}\ (\mathit{Plus}\ u_1\ u_2)$: The definition of R implies that $t_1\ \mathsf{R}\ u_1$ and $t_2\ \mathsf{R}\ u_2$. Ex hypothesi, $\mathit{head}\ t_1 = \mathit{head}\ u_1$ and $(\mathit{tail}\ t_1)\ \mathsf{R}\ (\mathit{tail}\ u_1)$, and likewise for t_2 and u_2.

$$\begin{array}{llcll}
& \mathit{head}\ (\mathit{Plus}\ t_1\ t_2) & & & \mathit{tail}\ (\mathit{Plus}\ t_1\ t_2) \\
= & \quad \{ \text{definition of } \mathit{head} \} & & = & \quad \{ \text{definition of } \mathit{tail} \} \\
& \mathit{head}\ t_1 + \mathit{head}\ t_2 & & & \mathit{Plus}\ (\mathit{tail}\ t_1)\ (\mathit{tail}\ t_2) \\
= & \quad \{ \text{ex hypothesi} \} & & \mathsf{R} & \quad \{ \mathsf{R} \text{ is a congruence} \} \\
& \mathit{head}\ u_1 + \mathit{head}\ u_2 & & & \mathit{Plus}\ (\mathit{tail}\ u_1)\ (\mathit{tail}\ u_2) \\
= & \quad \{ \text{definition of } \mathit{head} \} & & = & \quad \{ \text{definition of } \mathit{tail} \} \\
& \mathit{head}\ (\mathit{Plus}\ u_1\ u_2) & & & \mathit{tail}\ (\mathit{Plus}\ u_1\ u_2)
\end{array}$$

Consequently, $\mathsf{R} \subseteq \sim$ and furthermore $\mathsf{R} = \sim$. □

Lemma 4 *eval is an interpreter.*

$$\begin{array}{ll}
\mathit{eval}\ (\mathit{Var}\ s) & = s \\
\mathit{eval}\ (\mathit{Repeat}\ k) & = \mathit{repeat}\ k \\
\mathit{eval}\ (\mathit{Plus}\ e_1\ e_2) & = \mathit{plus}\ (\mathit{eval}\ e_1)\ (\mathit{eval}\ e_2) \\
\mathit{eval}\ (\mathit{Nat}) & = \mathit{nat}
\end{array}$$

PROOF. **Case** *Var* s: First of all, $\{ (\mathit{Var}\ s, s) \mid s \in \mathit{Stream}\ \tau \}$ is a bisimulation, consequently $\mathit{Var}\ s \sim s$. Lemma 1 furthermore implies $\mathit{Var}\ s \sim \mathit{eval}\ (\mathit{Var}\ s)$. Transitivity gives $\mathit{eval}\ (\mathit{Var}\ s) \sim s$, which in turn implies $\mathit{eval}\ (\mathit{Var}\ s) = s$. **Case** *Repeat* k: By definition. **Case** *Plus* $e_1\ e_2$: We first show that $\mathit{Var}\ (\mathit{eval}\ e) \sim e$.

$$\begin{array}{ll}
& \mathit{Var}\ s \sim s \\
\Longrightarrow & \quad \{ \text{substitute } s = \mathit{eval}\ e \} \\
& \mathit{Var}\ (\mathit{eval}\ e) \sim \mathit{eval}\ e \\
\Longrightarrow & \quad \{ \mathit{eval}\ e \sim e \} \\
& \mathit{Var}\ (\mathit{eval}\ e) \sim e
\end{array}$$

We proceed

$$\begin{array}{ll}
& e_1 \sim \mathit{Var}\ (\mathit{eval}\ e_1) \text{ and } e_2 \sim \mathit{Var}\ (\mathit{eval}\ e_2) \\
\Longrightarrow & \quad \{ \text{Lemma 3: } \sim \text{ is a congruence} \} \\
& \mathit{Plus}\ e_1\ e_2 \sim \mathit{Plus}\ (\mathit{Var}\ (\mathit{eval}\ e_1))\ (\mathit{Var}\ (\mathit{eval}\ e_2)) \\
\Longleftrightarrow & \quad \{ \text{Lemma 2} \} \\
& \mathit{eval}\ (\mathit{Plus}\ e_1\ e_2) = \\
& \quad \mathit{eval}\ (\mathit{Plus}\ (\mathit{Var}\ (\mathit{eval}\ e_1))\ (\mathit{Var}\ (\mathit{eval}\ e_2))) \\
\Longleftrightarrow & \quad \{ \text{Definition of } \mathit{plus} \} \\
& \mathit{eval}\ (\mathit{Plus}\ e_1\ e_2) = \mathit{plus}\ (\mathit{eval}\ e_1)\ (\mathit{eval}\ e_2)
\end{array}$$

Case *Nat*: By definition. □

Equipped with this lemma we can now show that *repeat*, *nat* and *plus* satisfy the recursion equations. We only give the proof for *nat* as the others follow exactly the same scheme.

$$\begin{array}{ll}
& \mathit{nat} \\
= & \quad \{ \text{definition of } \mathit{nat} \text{ and } \mathit{eval} \} \\
& \mathit{head}\ \mathit{Nat} \prec \mathit{eval}\ (\mathit{tail}\ \mathit{Nat}) \\
= & \quad \{ \text{definition of } \mathit{head} \text{ and } \mathit{tail} \} \\
& 0 \prec \mathit{eval}\ (\mathit{Plus}\ \mathit{Nat}\ (\mathit{Repeat}\ 1)) \\
= & \quad \{ \text{Lemma 4: } \mathit{eval} \text{ is an interpreter} \} \\
& 0 \prec \mathit{plus}\ \mathit{nat}\ (\mathit{repeat}\ 1)
\end{array}$$

A.6 Uniqueness of solutions

Assume that $\underline{\mathit{repeat}}$, $\underline{\mathit{plus}}$ and $\underline{\mathit{nat}}$ also satisfy the stream equations. We show that they must be equal to *repeat*, *plus* and *nat*. Let R be given by the following inductive definition.

$$\begin{array}{lll}
\mathsf{R} & = & \sim \cup \{ (\mathit{repeat}\ k, \underline{\mathit{repeat}}\ k) \mid k \in \tau \} \\
& & \cup \{ (\mathit{plus}\ s_1\ s_2, \underline{\mathit{plus}}\ t_1\ t_2) \mid s_1\ \mathsf{R}\ t_1 \text{ and } s_2\ \mathsf{R}\ t_2 \} \\
& & \cup \{ (\mathit{nat}, \underline{\mathit{nat}}) \}
\end{array}$$

We show that R is a bisimulation by induction on its definition. Hence, $\mathsf{R} \subseteq \sim$ and consequently $\mathsf{R} = \sim$. **Case** $s \sim t$: trivial. **Case** $(\mathit{repeat}\ k)\ \mathsf{R}\ (\underline{\mathit{repeat}}\ k)$: Omitted. **Case** $\mathit{nat}\ \mathsf{R}\ \underline{\mathit{nat}}$: Omitted. **Case** $(\mathit{plus}\ s_1\ s_2)\ \mathsf{R}\ (\underline{\mathit{plus}}\ t_1\ t_2)$: The definition of R implies that $s_1\ \mathsf{R}\ t_1$ and $s_2\ \mathsf{R}\ t_2$. Ex hypothesi, $\mathit{head}\ s_1 = \mathit{head}\ t_1$ and $(\mathit{tail}\ s_1)\ \mathsf{R}\ (\mathit{tail}\ t_1)$, and likewise for s_2 and t_2.

$$\begin{array}{llcll}
& \mathit{head}\ (\mathit{plus}\ s_1\ s_2) & & & \mathit{tail}\ (\mathit{plus}\ s_1\ s_2) \\
= & \quad \{ \mathit{plus} \text{ satisfies the eqn} \} & & = & \quad \{ \mathit{plus} \text{ satisfies the eqn} \} \\
& \mathit{head}\ s_1 + \mathit{head}\ s_2 & & & \mathit{plus}\ (\mathit{tail}\ s_1)\ (\mathit{tail}\ s_2) \\
= & \quad \{ \text{ex hypothesi} \} & & \mathsf{R} & \quad \{ \text{definition of } \mathsf{R} \} \\
& \mathit{head}\ t_1 + \mathit{head}\ t_2 & & & \underline{\mathit{plus}}\ (\mathit{tail}\ t_1)\ (\mathit{tail}\ t_2) \\
= & \quad \{ \underline{\mathit{plus}} \text{ satisfies the eqn} \} & & = & \quad \{ \underline{\mathit{plus}} \text{ satisfies the eqn} \} \\
& \mathit{head}\ (\underline{\mathit{plus}}\ t_1\ t_2) & & & \mathit{tail}\ (\underline{\mathit{plus}}\ s_1\ s_2)
\end{array}$$

Since $\mathsf{R} = \sim$, it follows that $\mathit{nat} \sim \underline{\mathit{nat}}$ and by coinduction $\mathit{nat} = \underline{\mathit{nat}}$, and likewise for the other operations. □

References

P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, and A. Poigné, editors, *Category Theory and Computer Science (Manchester)*, LNCS 389, pages 357–365, 1989. Springer.

Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-–28, 1993, Selected Papers*, LNCS 806, pages 62–78, 1994. Springer.

Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, (XX):1–14, 2005.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics*. Addison-Wesley, 2nd edition, 1994.

Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

Ralf Hinze and Andres Löh. Guide2lhs2tex (for version 1.13), February 2008. `http://people.cs.uu.nl/andres/lhs2tex/`.

Jerzy Karczmarczuk. Generating power of lazy semantics. *Theoretical Computer Science*, (187):203–219, 1997.

M. Douglas McIlroy. The music of streams. *Information Processing Letters*, (77):189–195, 2001.

M. Douglas McIlroy. Power series, power serious. *J. Functional Programming*, 3(9):325–337, May 1999.

Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. Lawall, editor, *Proceedings of the 11th ACM SIGPLAN international conference on Functional programming, Portland, 2006*, pages 50–61. ACM Press, 2006.

J.J.M.M. Rutten. Fundamental study — Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, (308):1–53, 2003.

J.J.M.M. Rutten. A coinductive calculus of streams. *Math. Struct. in Comp. Science*, (15):93–147, 2005.

Neil J.A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/`.