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 im-
plemented by corecursive programs, and proofs are conducted us-
ing 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 reason-
ing 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 de-
development is constructive: streams and stream operators are im-
plemented 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 Tech-
niques]: 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 role, but often they can be avoided by
finding any index variables and not many subscripts though we deal
with recurrences, summations and power series. The common denomi-
nator of these examples is that they are, or rather, that they can be
based on streams, infinite sequences of elements. In a lazy func-
tional language, such as Haskell (Peyton Jones 2003), streams are
easy to define and many textbooks on Haskell reproduce the folk-
lore examples of Fibonacci or Hamming numbers defined by recur-
dition equations over streams. One has to be a bit careful in for-
mulating 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 general-
asied 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 intro-
duces 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 recur-
dences. 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 [α],
except that there is no base constructor so we cannot construct a
finite stream. The Stream type is not an inductive type, but a co-
inductive type, whose semantics is given by a final coalgebra
(Aczel and Mendler 1989).

data Stream α = Cons { head :: α, tail :: Stream α }

infixr 5 ≺
(≺) :: α → Stream α → Stream α
α ≺ s = Cons α s

Streams are constructed using ≺, which prepends an element to a
stream. They are destructed using head, which yields the first ele-
ment, 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.
2.1 Operations

Most definitions we encounter in the sequel make use of the following functions, which lift n-ary operations \( (\alpha \to \alpha) \to \alpha \) to streams.

\[
\begin{align*}
\text{repeat } & : \alpha \to \text{Stream } \alpha \\
\text{repeat } a & = s \quad \text{where } s = a \prec s \\
\text{map } & : (\alpha \to \beta) \to (\text{Stream } \alpha \to \text{Stream } \beta) \\
\text{map } f s & = f (\text{head } s) \prec \text{map } f (\text{tail } s) \\
\text{zip } & : (\alpha \to \beta \to \gamma) \to (\text{Stream } \alpha \to \text{Stream } \beta \to \text{Stream } \gamma) \\
\text{zip } f s t & = f (\text{head } s) (\text{head } t) \prec \text{zip } f (\text{tail } s) (\text{tail } t)
\end{align*}
\]

The call \( \text{repeat } 0 \) constructs a sequence of zeros (A000004\(^1\)). Clearly, a constant stream is of the form \( \text{repeat } k \) for some \( k \). We refer to \( \text{repeat} \) as a parametrised stream and to \text{map} and \text{zip} as stream operators.

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

\begin{verbatim}
instance (Num a) => Num (Stream a) where
  (+)      = zip (+) \\
  (-)      = zip (-) \\
  (*)      = zip (*) \\
  negate   = map negate -- unary minus

instance (Show a) => Show (Stream a) where
  show (Stream s) = show s
\end{verbatim}

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

Using this vocabulary we can already define the usual suspects:

- The three sequences are given by recursion, every stream is a solution of the second one. This situation is many solutions: every constant stream is a solution of the first equation.
- The time 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 \text{nat} = \text{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.

\[
\begin{align*}
\text{iterate } & : (\alpha \to \alpha) \to (\alpha \to \text{Stream } \alpha) \\
\text{iterate } f a & = a \prec \text{iterate } f (f a)
\end{align*}
\]

So, \text{iterate } \((+1)\) 0 is yet another definition of the naturals.

2.2 Definitions

Not every legal Haskell definition of type \text{Stream } \tau \text{ actually defines a stream. Two simple counterexamples are } s = \text{tail } s \text{ and } s = \text{head } s \prec \text{tail } s. \text{ 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 \( \text{Stream } \tau \), \( h \) is a constant expression of type \( \tau \), and \( t \) is an expression of type \( \text{Stream } \tau \) possibly referring to \( x \) or some other stream identifier in the case of mutual recursion. However, neither \( h \) nor \( t \) may involve \( \text{head } x \) or \( \text{tail } x \).

If \( x \) is a parametrised stream or a stream operator,

\[
x_1 \ldots x_n = h \prec t
\]

then \( h \) or \( t \) may use \( \text{head } x_1 \) or \( \text{tail } x_1 \) provided \( x_1 \) 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 \text{map} and \text{zip} and \( \gamma \), meet the requirements.

\(^1\) 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 \text{A?????} refer to entries in that database. Somewhat surprisingly, \text{repeat } 0 \) is not A000000. Just in case you were wondering, the first sequence (A000001) lists the number of groups of order \( n \).
As an aside, we could relax the conditions somewhat so that
\[
\text{fib} = 0 \prec 1 \prec \text{tail fib} + \text{fib}
\]
becomes admissible. However, the gain in expressivity is modest
as we can always eliminate such calls to \text{tail} by introducing a name
for the tail. In the example above, we simply replace \text{tail fib} by \text{fib'}
obtaining the two equations given in Sec. 2.1.

By the way, non-recursive definitions like
\[
nat' = \text{nat} + 1
\]
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 \text{arithmetic}.

Streams satisfy the following extensionality property.
\[
s = \text{head } s \prec \text{tail } s
\]
App. A provides a coinductive proof of this law.

Interleaving interacts nicely with lifted operations: let \oplus = \text{map} (\oplus) and \ominus = \text{zip} (\ominus), \oplus and \ominus arbitrary functions, then
\[
c \ Y \ c
\]
\[
\ominus s \ Y \ (\ominus t) = \ominus (s \ Y \ t)
\]
\[
(s_1 + s_2) \ Y \ (t_1 + t_2) = (s_1 \ Y \ t_1) + (s_2 \ Y \ t_2)
\]
A simple consequence is \((s \ Y \ t) + 1 = s + 1 \ Y \ t + 1\). The
last property is called \text{abide law} because of the following two-
dimensional way of writing the law, in which the two operators are
written either above or beside each other.
\[
s_1 + s_2 \ Y \ = \ s_1 \ Y \ + \ s_2 \ Y \ t_1 + t_2
\]
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 \(\text{fix } \phi\). (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.
\[
\text{fix } \phi s = s \quad \iff \quad \phi s = s
\]
Read from left to right it states that \(\text{fix } \phi\) is indeed a solution of
\(\phi = \phi x\). Read from right to left it asserts that any solution is equal
to \(\text{fix } \phi\). So, if we want to prove \(s = t\) where \(s = \text{fix } \phi\), 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 \text{repeat} \(k\) for some \(k\).
\[
c
\]
\[
\text{extensionality}\}
\]
\[
\text{head } c \prec \text{tail } c
\]
\[
\text{c is constant}\}
\]
\[
\text{head } c \prec c
\]
Consequently, \(c\) equals the unique solution of \(x = \text{head } c \prec x\),
which by definition is \text{repeat} \(\text{head } c\).

That was easy. The next proof is not much harder: we show that
\[
nat = 2 \ * \ nat \ Y \ 2 \ * \ nat + 1
\]
\[
\text{definition of } \nat
\]
\[
2 \ * (0 \prec 1 \prec \text{nat} + 1) \ Y \ 2 \ * \ nat + 1
\]
\[
\text{arithmetic}
\]
\[
(0 \prec 2 \ * \ nat + 2) \ Y \ 2 \ * \ nat + 1
\]
\[
\text{definition of } Y
\]
\[
0 \prec 2 \ * \ nat + 1 \ Y \ 2 \ * \ nat + 2
\]
\[
\text{arithmetic}
\]
\[
0 \prec (2 \ * \ nat \ Y \ 2 \ * \ nat + 1) + 1
\]
Inspecting the second but last term, we note that the result implies
\(\text{nat} = 0 \prec 2 \ * \text{nat} + 1 \ Y \ 2 \ * \text{nat} + 2\), which in turn proves \(\text{nat} = \text{bin}\).

Now, if both \(s\) and \(t\) are given as fixed points, \(s = \text{fix } \psi\) and
t = \text{fix } \chi, then there are at least four possibilities to prove \(s = t\):
\[
\forall (\psi s) = \psi s \implies \forall \psi s = s \implies s = t
\]
\[
\forall (\phi t) = \phi t \implies \forall \phi t = t \implies s = t
\]
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 \(\chi\) has a unique fixed
point, then \(s = t\). The point is that we discover the function \(\chi\) on
the fly during the calculation. Proofs in this style are laid out as follows.
\[
s
\]
\[
\{ \text{why?} \}
\]
\[
\chi s
\]
\[
\subseteq \{ x = \chi x \text{ has a unique solution} \}
\]
\[
\chi t
\]
\[
\{ \text{why?} \}
\]
\[
t
\]
The symbol \(\subseteq\) 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 \text{Cassini’s identity:}
\[
\text{fib}’’ \ * \ 2 - \text{fib} \ * \text{fib}’’ = (−1) \ * \ text{nat} \text{where fib}’’ = \text{tail fib’} = \text{fib’} + \text{fib}.
\]
\[
\text{fib}’’ \ * \ 2 - \text{fib} \ * \text{fib}’’
\]
\[
\{ \text{definition of fib’’ and arithmetic} \}
\]
\[
\text{fib}’’ \ * \ 2 - (\text{fib} \ * \text{fib’} \ * \text{fib}’’ \ * \ 2)
\]
\[
\{ \text{definition of fib and fib’} \}
\]
\[
1 \prec (\text{fib}’’ \ * \ 2 - (\text{fib} \ * \text{fib’} + \text{fib’’} \ * \ 2))
\]
\[
\{ \text{fib’’} - \text{fib}’ = \text{fib} \text{ and arithmetic} \}
\]
\[
1 \prec (-1) \ * (\text{fib}’’ \ * \ 2 - \text{fib} \ * \text{fib’’})
\]
\[
\subseteq \{ x = 1 \prec (-1) \ * x \text{ has a unique solution} \}
\]
\[
1 \prec (-1) \ * (-1) \ * \ text{nat}
\]
\[
\{ \text{arithmetic} \}
\]
\[
(-1) \ * 0 \prec (-1) \ * (\text{nat} + 1)
\]
\[
\{ \text{definition of nat and arithmetic} \}
\]
\[
(-1) \ * \ text{nat}
\]
When reading \(\subseteq\)-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
\text{Y-law} by showing \(f = g\) where
\[
f \ s \ t = \ominus s \ Y \ominus t \quad \text{and} \quad g \ s \ t = \ominus (s \ Y \ t)
\]
The proof is straightforward involving only bureaucratic steps.

\[
\begin{align*}
& f \ a \ b \\
& \equiv \{ \text{definition of } f \} \\
& \quad \Diamond \ a \ \Box \ b \\
& \equiv \{ \text{definition of } \Box \text{ and } \Diamond = \text{map} (\Box) \} \\
& \quad \Diamond \ \text{head} \ a \prec \Diamond \ b \ \Box \ \text{tail} \ a \\
& \equiv \{ \text{definition of } f \} \\
& \quad \Diamond \ \text{head} \ a \prec \Diamond \ (\text{tail} \ a) \\
& \subseteq \{ x \ s \ t = \Diamond \ \text{head} \ s \prec x \ t \ (\text{tail} \ s) \text{ has a unique solution} \} \\
& \quad \Diamond \ \text{head} \ a \prec g \ b \ (\text{tail} \ a) \\
& \equiv \{ \text{definition of } g \} \\
& \quad \Diamond \ \text{head} \ a \prec \Diamond \ (b \ \Box \ \text{tail} \ a) \\
& \equiv \{ \Diamond = \text{map} (\Box) \text{ and definition of } \Box \} \\
& \quad \Diamond (a \ \Box \ b) \\
& \equiv \{ \text{definition of } g \} \\
& \quad a \ b
\end{align*}
\]

In the sequel, we usually leave the two functions implicit sparing ourselves twirling 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 \textit{iterate} fusion law (Gibbons and Hutton 2005), which amounts to the free theorem of \((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \text{Stream} \ \alpha)\).

\[
\text{map} \ h \ \cdot \ \text{iterate} \ f_1 = \text{iterate} \ f_2 \cdot h \quad \Leftarrow \quad h \cdot f_1 = f_2 \cdot h
\]

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

\[
\begin{align*}
\text{map} \ h \ \cdot \ \text{iterate} \ f_1 \ a \\
& \equiv \{ \text{definition of \textit{iterate} and \textit{map}} \} \\
& h \ a \prec \text{map} \ h \ (\text{iterate} \ f_1 \ (f_1 \ a)) \\
& \subseteq \{ x \ a = h \ a \prec x \ (f_1 \ a) \text{ has a unique solution} \} \\
& h \ a \prec \text{iterate} \ f_2 \ (h \ (f_1 \ a)) \\
& \equiv \{ \text{assumption: } h \cdot f_1 = f_2 \cdot h \} \\
& h \ a \prec \text{iterate} \ f_2 \ (f_2 \ (h \ a)) \\
& \equiv \{ \text{definition of \textit{iterate}} \} \\
& \text{iterate} \ f_2 \ (h \ a)
\end{align*}
\]

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

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

\[
\text{iterate} \ (+1) \ 0 \\
= \{ \text{definition of \textit{iterate}} \} \\
0 \prec \text{iterate} \ (+1) \ 1 \\
= \{ \text{iterate fusion law: } h = f_1 = f_2 = (+1) \} \\
0 \prec \text{iterate} \ (+1) \ 0 + 1
\]

3. Recurrences (\( \prec, \Box \))

Using \( \prec \) and \( \Box \) we can easily capture recurrences: the sequence defined by \( a_0 = k \) and \( a_{n+1} = f(a_n) \) becomes the stream equation \( a = k \prec \text{map} \ f \ a \); likewise, the sequence given by \( a_0 = k, a_{2n+1} = f(a_n) \) and \( a_{2n+2} = g(a_n) \) becomes \( a = k \prec \text{map} \ f \ a \ \Box \ \text{map} \ g \ a \). 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.

\[
\text{pot} = \text{True} \prec \text{pot} \ \Box \ \text{repeat False}
\]

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

\[
\text{msb} = 1 \prec 2 \ast \text{msb} \ \Box \ 2 \ast \text{msb}
\]

Put differently, \( \text{msb} \) is the largest pot less than or equal to \( \text{nats} \). (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 \textit{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 \( \text{ones} = \text{ones} \ \Box \ \text{ones} + 1 \). Adding two initial values, we can turn the property into a definition.

\[
\text{ones} = 0 \prec \text{ones}' \\
\text{ones}' = 1 \prec \text{ones} \ \Box \ \text{ones}' + 1
\]

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

Let’s inspect the sequences.

\[
\Rightarrow \ast \text{msb} \\
<1, 2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 16, \ldots > \\
\Rightarrow \text{nat} \prec \ast \text{msb} \\
<0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, \ldots > \\
\Rightarrow \text{ones} \\
<0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 3, 3, 4, \ldots >
\]

The sequence \( \text{nat} \prec \ast \text{msb} \) (A053645) exhibits a nice pattern; it describes the distance to the largest pot less than or equal to \( \text{nats} \).

3.2 Binary carry sequence

Here is a sequence that every computer scientist should know: the \textit{binary carry sequence} or \textit{ruler function} (A0007814).

\[
\text{carry} = 0 \prec \text{carry} + 1
\]

(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: carry = 0 \( \prec \) carry + 1 \( \prec \) 0. The unfolding works as long as the the first argument of \( \Box \) is a sequence defined elsewhere.) The sequence gives the exponent of the largest pot dividing \( \text{nats} \), 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.

\[
\begin{align*}
\text{turn} \ 0 & = [] \\
\text{turn} \ (n + 1) & = \text{turn} \ n + [n] \ast \text{turn} \ n
\end{align*}
\]

The call \( \text{turn} \ n \) yields the heights of the nodes of a perfect binary tree of depth \( n \). Now, imagine traversing an infinite binary tree starting at the lefmost 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.

\[
\text{tree} \ n = n \ast \text{turn} \ n \ast \text{tree} \ (n + 1)
\]
where \( \leftarrow \) prepends a list to a sequence.

```plaintext
fix 5 \( \leftarrow \)
\[
\begin{align*}
\{ \leftarrow \} & \: \Rightarrow [\alpha] \rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha \\
\{ \} & \: \Rightarrow s = s \\
\{ \alpha : \text{as} \} & \: \Rightarrow s = a \leftarrow \{ \text{as} \leftarrow \{ s \} \}
\end{align*}
\]
```

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 \ Y x + 1`.

\[
\begin{align*}
0 \ Y \ \text{tree } 0 + 1 &= \{ \text{ definition of } \ Y \} \\
0 &\leftarrow \text{tree } 0 + 1 \ Y 0 \\
0 &= \{ \text{ proof obligation, see below } \} \\
0 &\leftarrow \text{tree } 1
\end{align*}
\]

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

\[
\{ x \leftarrow s \} \ Y \{ y \leftarrow t \} = \{ x \ Y y \} \leftarrow \{ s \ Y t \}
\]

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

\[
\begin{align*}
\text{tree } (k + 1) &= \{ \text{ definition of tree } \} \\
k + 1 &\leftarrow \text{turn } (k + 1) \leftarrow \text{tree } (k + 2) \\
\subset &\{ x n = n + 1 \leftarrow \text{turn } (n + 1) \leftarrow x \ (n + 1) \text{ has an u. s. } \} \\
k + 1 &\leftarrow \text{turn } (k + 1) \leftarrow \{ \text{tree } (k + 1) + 1 \ Y 0 \} \\
&\leftarrow \{ \text{ proof obligation, see below } \} \\
k + 1 &\leftarrow \{ \text{rep } 2^k 0 \ Y \text{turn } k + 1 \} \leftarrow \{ \text{tree } (k + 1) + 1 \ Y 0 \}
\end{align*}
\]

It remains to show the finite version of the proof obligation: `turn (k + 1) = replicate 2^k 0 \ Y \ 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`.

\[
\frac{\text{frac}}{\text{nat} \ Y \ \text{frac}}
\]

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:

\[
\text{god} = 2 \times \frac{\text{frac}}{+1}
\]

is the greatest odd divisor of `nat' = `2 \times \text{nat} + 1` \ Y `2 \times \text{nat} + 2`. The greatest odd divisor of an odd number, `2 \times \text{nat} + 1`, is the number itself; the greatest odd divisor of an even number, `2 \times \text{nat} + 2`, is the `god of nat'`.

Now, recall that `2 \times carry` is the largest power of two dividing `nat'`. Putting these observations together, we have

\[
2 \times \text{carry} \times \text{god} = \text{nat'}
\]

The proof is surprisingly straightforward.

\[
\begin{align*}
2 \times \text{carry} \times \text{god} &= \{ \text{ definition of carry and } \text{god } \} \\
2 \times (0 \ Y \text{carry} + 1) \times (2 \times \text{nat} + 1 \ Y \text{god}) &= \{ \text{ arithmetic and abide law } \} \\
2 \times \text{nat} + 1 \ Y 2 \times 2 \times \text{carry} \times \text{god} \subset \{ x = 2 \times \text{nat} + 1 \ Y 2 \times \text{has a unique solution } \} \\
2 \times \text{nat} + 1 \ Y 2 \times (\text{nat} + 1) &= \{ \text{ arithmetic } \} \\
2 \times \text{nat} + 1 \ Y 2 \times \text{nat} + 2 &= \{ \text{ property of } \text{nat}', \text{see above } \}
\end{align*}
\]

#### 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:

\[
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7
\]

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:

\[
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7
\]

since the number is odd, the first person is killed, as well. Renumbering \( 3 \ 5 \ 7 \) to \( 1 \ 2 \ 3 \ 4 \) we observe that if \( i \) is killed in the remaining sequence, then \( 2i + 1 \) is killed in the original sequence.

\[
\text{jos} = 1 \times 2 \times \text{jos} - 1 \ Y 2 \times \text{jos} + 1
\]

It’s quite revealing to inspect the sequence.

\[
\text{jos} \begin{cases}
<1, 1, 3, 1, 3, 5, 7, 1, 3, 5, 7, 9, 11, 13, 15, 1,.. > \\
</0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0,.. >
\end{cases}
\]

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

\[
\text{jos} = 2 \times (\text{nat'} - \text{msb}) + 1
\]

In terms of bit operations, `jos` implements a cyclic left shift: `nat’ - msb` removes the most significant bit, `2` shifts to the left and `+1` sets the least significant bit.

\[
2 \times (\text{nat’} - \text{msb}) + 1
\]

\[
= \{ \text{ definition of } \text{msb } \text{and property of } \text{nat’} \}
\]

\[
2 \times (\{1 \times 2 \times \text{nat’} \ Y \text{nat’} + 1\} - \{1 \times 2 \times \text{msb} \ Y 2 \times \text{msb}\}) + 1
\]

\[
= \{ \text{ definition of } - \text{ and abide law } \}
\]

\[
2 \times (0 \times 2 \times \text{nat’} - 2 \times \text{msb} \ Y 2 \times \text{nat’} + 1 - 2 \times \text{msb}) + 1
\]

\[
= \{ \text{ arithmetic } \}
\]

\[
1 - 2 \times (2 \times (\text{nat’} - \text{msb}) + 1) \ Y 2 \times (2 \times (\text{nat’} - \text{msb}) + 1) + 1
\]
4. Finite calculus ($\Delta$, $\Sigma$)

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 route 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.

$$\Delta \colon \{\text{Num } x\} \Rightarrow \text{Stream } x \Rightarrow \text{Stream } x$$

$$\Delta s = \text{tail } s - s$$

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

$$\gg \Delta (\text{nat} \prec 3) \gg$$

$$<1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, .. >$$

$$\gg \Delta (2 \prec \text{nat}) \gg$$

$$<1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, .. >$$

$$\gg \Delta \text{ carry} \gg$$

$$<1, -1, 2, -2, 1, -1, 3, -3, 1, -1, 2, -2, 1, -1, 4, -4, .. >$$

$$\gg \Delta \text{ jos} \gg$$

$$<0, 2, -2, 2, 2, 6, 2, 2, 2, 2, 2, -14, 2, .. >$$

Finite calculus has a simple rule for the derivative of a power:

$$(x^{n+1})' = (n+1)x^n.$$ Unforturately, the first example above shows that finite difference does not interact nicely with ordinary powers: $\Delta (\text{nat} \prec 3)$ is not $3 \ast \text{nat}^{\prec 2}$. Can we find a different notion that enjoys an analogous rule? Let’s try. Writing $x^n$ for the new power and its lifted variant, we calculate

$$\Delta (\text{nat}^{n+1}) \gg$$

$$= \{\text{definition of } \Delta\}$$

$$\gg tail (\text{nat}^{n+1}) = \text{nat}^{n+1} \gg$$

$$= \{ s^n = \text{map } (\lambda x \rightarrow x^n) \text{ s and definition of nat } \}$$

$$\gg (\text{nat} + 1)^{n+1} = \text{nat}^{n+1} \gg$$

Starting at the other end, we obtain

$$\gg \text{repeat } n + 1 \ast \text{nat}^{n} \gg$$

$$= \{ \text{arithmetic} \}$$

$$\gg (\text{nat} + 1) \ast \text{nat}^{n} = \text{nat}^{n} \ast (\text{nat} \ast \text{repeat } n) \gg$$

We can connect the loose ends if the new power satisfies both

$$x \ast (x - 1)^{n+1} = x^{n+1} = x^n \ast (x - n).$$

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.)

$$x^0 = x^\omega$$

$$x^1 = x^\omega$$

$$x^2 = x^\omega + x^\omega$$

$$x^3 = x^\omega + 3 * x^\omega + x^\omega$$

$$x^4 = x^\omega + 3 * x^\omega + 3 * x^\omega + x^\omega$$

Figure 1. Converting between powers and falling factorial powers.

$$\Delta (\text{tail } s) = \text{tail } (\Delta s)$$

$$\Delta (a < s) = \text{head } s - a < \Delta s$$

$$\Delta (s \gg t) = (t - s) \gg (tail \ s - t)$$

$$\Delta c = 0$$

$$\Delta (c * s) = c * \Delta s$$

$$\Delta (s + t) = \Delta s + \Delta t$$

$$\Delta (s \gg t) = s \gg \Delta s + \Delta s \gg tail \ t$$

$$\Delta (c \prec nat) = (c - 1) * c \prec nat$$

$$\Delta (\text{nat}^{n+1}) = (\text{repeat } n + 1) * \text{nat}^{n}$$

Figure 2. Laws for finite difference.

$$\Delta (c \prec nat)$$

$$= \{ \text{definition of } \Delta \}$$

$$\gg tail (c \prec nat) - c \prec nat$$

$$= \{ c \text{ is constant and definition of nat } \}$$

$$c \prec (\text{nat} + 1) - c \prec nat$$

$$= \{ \text{arithmetic} \}$$

$$(c - 1) * c \prec nat$$

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

$$\gg \Delta (s * t) \gg$$

$$= \{ \text{definition of } \Delta \text{ and } * \}$$

$$\gg tail s * tail t \ast s \ast t$$

$$= \{ \text{arithmetic} \}$$

$$s * tail t - s \ast t + tail s * tail t - s * tail t$$

$$= \{ \text{distributivity} \}$$

$$s * (tail t - t) + (tail s - s) * tail t$$

$$= \{ \text{definition of } \Delta \}$$

$$s * \Delta t + \Delta s * tail t$$

4.1.2 Examples

Let’s get back to the Josephus problem: the interactive session in Sec. 4.1 suggests that $\Delta \text{ jos}$ is almost always 2, except for pots. We can express this property using a stream conditional.

$$\Delta \text{ jos} = (\text{pot' } \rightarrow \neg \text{nat} ; 2)$$

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

$$(s_1 \ Y \ s_2 \rightarrow t_1 \ Y \ t_2 ; u_1 \ Y \ u_2) = (s_1 \rightarrow t_1 ; u_1) \ Y (s_2 \rightarrow t_2 ; u_2)$$

Both laws are used in the proof of the above property.
\[ \Delta jos = \{ \Delta \text{ law and arithmetic } \} \\
0 \prec 2 \; \forall \; 2 \ast (\text{tail } jos - jos) - 2 = \{ \text{definition of } \Delta \} \\
0 \prec 2 \; \forall \; 2 \ast \Delta jos - 2 \subseteq \{ x = 0 \prec 2 \; \forall \; 2 \ast x - 2 \text{ has a unique solution} \} \\
0 \prec 2 \; \forall \; 2 \ast (pot' \rightarrow -nat; \; 2) - 2 = \{ \text{arithmetic and definition of } nat' \} \\
0 \prec 2 \; \forall \; (pot' \rightarrow -(2 \ast nat'); \; 2) = \{ \text{definition of } nat, \; pot \text{ and } \forall \} \\
(pot' \rightarrow -(2 \ast nat); \; 2) \; \forall \; 2 = \{ \text{conditional and abide law} \} \\
(pot \forall \; repeat \; False \rightarrow -(2 \ast nat \; \forall \; 2 \ast nat + 1); \; 2 \; \forall \; 2) = \{ \text{definition of } pot' \text{ and characterisation of } nat \} \\
(pot' \rightarrow -nat; \; 2) \\
\]

4.2 Summation

Finite difference $\Delta$ has a right-inverse: the summation operator $\Sigma$. We can easily derive its definition.

\[
\Delta (\Sigma s) = s \\
\iff \{ \text{definition of } \Delta \} \\
\text{tail } (\Sigma s) - \Sigma s = s \\
\iff \{ \text{arithmetic} \} \\
\text{tail } (\Sigma s) = \Sigma s + s \\
\]

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

\[
\Sigma : (\text{Num } \alpha) \Rightarrow \text{Stream } \alpha \Rightarrow \text{Stream } \alpha \\
\Sigma s = t \text{ where } t = 0 \prec t + s \\
\]

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

\[
\gg, \Sigma \{ 0 \; \forall \; 1 \} \prec 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, \ldots \gg \gg \Sigma \{ 2 \ast nat + 1 \} \prec 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, \ldots \gg \gg \Sigma \text{ carry} \prec 0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, 11, \ldots \\
\]

The definition of $\Sigma$ 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, $\Sigma 1 = nat$ as $nat = 0 \prec nat + 1$. 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 $\Sigma \text{ fib } = \text{fib'} - 1$.

\[
\text{fib'} - 1 = \{ \text{definition of } \text{fib'} \} \\
(1 \prec \text{fib'} + \text{fib}) - 1 = \{ \text{arithmetic} \} \\
0 \prec (\text{fib'} - 1) + \text{fib} \\
\]

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 $\Delta$ and $\Sigma$.

\[
t = \Delta s \iff \Sigma t = s - \text{repeat } \text{head } s \\
\]

\[
\Sigma \{ \text{tail } s \} = \text{tail } (\Sigma s) - \text{repeat } (\text{head } s) \\
\Sigma \{ a \prec s \} = 0 \prec \text{repeat } a + \Sigma s \\
\Sigma \{ s \; \forall \; t \} = (\Sigma s + \Sigma t) \; \forall \; (s + \Sigma s + \Sigma t) \\
\Sigma \{ c \} = c \ast nat \\
\Sigma \{ c \ast s \} = c \ast \Sigma s \\
\Sigma \{ s + t \} = \Sigma s + \Sigma t \\
\Sigma \{ s \ast \Delta t \} = s \ast t - (\Delta s \ast \text{tail } t) - \text{head } (s \ast t) \\
\Sigma \{ c \ast nat \} = (c \ast \text{nat} - 1) / (c - 1) \\
\Sigma \{ \text{nat}^2 \} = \text{nat}^2 + 1 / (\text{repeat } n + 1) \\
\]

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

\[
\Sigma (\Delta s) = \{ \text{definition of } \Sigma \} \\
0 \prec \Sigma (\Delta s) + \Delta s \subseteq \{ x = 0 \prec x + \Delta s \text{ has a unique solution} \} \\
0 \prec s - \text{repeat } (\text{head } s) + \Delta s = \{ \text{definition of } \Delta \text{ and arithmetic} \} \\
(\text{head } s \ast \text{tail } s) - \text{repeat } (\text{head } s) = \{ \text{extensionality} \} \\
s - \text{repeat } (\text{head } s) \\
\]

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 $\Delta$. Let $c = \text{repeat } (\text{head } (s \ast t))$, then

\[
s \ast \Delta t + \Delta s \ast \text{tail } t = \Delta (s \ast t) \iff \{ \text{Fundamental Theorem} \} \\
\Sigma (s \ast \Delta t + \Delta s \ast \text{tail } t) = s \ast t - c = \{ \Sigma \text{ is linear} \} \\
\Sigma (s \ast \Delta t) + \Sigma (\Delta s \ast \text{tail } t) = s \ast t - c = \{ \text{arithmetic} \} \\
\Sigma (s \ast \Delta t) = s \ast t - \Sigma (\Delta s \ast \text{tail } t) - c \\
\]

Unlike the others, this law is not compositional: $\Sigma (s \ast t)$ 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.

\[
(t - s) \; \forall \; (\text{tail } s - t) = \Delta (s \; \forall \; t) = \{ \text{Fundamental Theorem and } \text{head } (s \; \forall \; t) = \text{head } s \} \\
\Sigma ((t - s) \; \forall \; (\text{tail } s - t)) = (s \; \forall \; t) - \text{repeat } (\text{head } s) \\
\]

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

\[
t - s = x \text{ and } \text{tail } s - t = y \\
\iff \{ \text{arithmetic} \} \\
x + y \text{ and } t = x + s \\
\iff \{ \text{definition of } \Delta \} \\
\Delta s = x + y \text{ and } t = x + s \\
\iff \{ \Delta (\Sigma s) = s \} \\
s = \Sigma x + \Sigma y \text{ and } t = x + \Sigma x + \Sigma y \\
\]

Since $\text{head } s = 0$, 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 \((0 \prec \text{A000330})\).

\[
\sum (\text{nat} \cdot \text{nat}) = \frac{1}{6} \sum (\text{nat} \cdot 3 - 3 \cdot \text{nat} \cdot 2 + 2 \cdot \text{nat}) + \frac{1}{2} \cdot (\text{nat} \cdot 2 - \text{nat})
\]

Voilà. We have found a closed form for \(\Sigma \text{carry}\).

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

\[
\Sigma \text{carry} = \begin{cases} \text{definition of carry} \\ \Sigma (0 \mathbin{\text{carry}} + 1) = \begin{cases} \text{summation law} \\ \Sigma (\text{carry} + 1) \mathbin{\text{carry}} \Sigma (\text{carry} + 1) = \begin{cases} \Sigma \text{is linear and } \Sigma 1 = \text{nat} \\ \Sigma \text{carry} + \text{nat} \mathbin{\text{carry}} \Sigma \text{carry} + \text{nat} = \begin{cases} \text{abide law} \\ \Sigma \text{carry} \mathbin{\text{carry}} \Sigma \text{carry} + \text{nat} \end{cases} \end{cases} \end{cases} \end{cases}
\]

That’s quite a weird property. Since we know where we are aiming at, let us determine \(\text{nat} \prec \Sigma \text{carry}\).

\[
\text{nat} \prec \Sigma \text{carry} = \begin{cases} \text{property of nat and } \Sigma \text{carry} \\ (2 \cdot \text{nat} \mathbin{\text{carry}} 2 \cdot \text{nat} + 1) - ([\Sigma \text{carry} \mathbin{\text{carry}} \Sigma \text{carry}] + (\text{nat} \mathbin{\text{carry}} \text{nat})) = \begin{cases} \text{arithmetical} \\ (\text{nat} \prec \Sigma \text{carry}) \mathbin{\text{carry}} (\text{nat} \prec \Sigma \text{carry}) + 1 \end{cases} \end{cases}
\]

Voilà again. The sequence \(\text{nat} \prec \Sigma \text{carry}\) satisfies \(x = x \mathbin{\text{carry}} x + 1\), which implies that \(\text{nat} \prec \Sigma \text{carry} = \text{ones}\). For the sake of completeness, we should also check that \(\text{head ones = head (nat \prec \Sigma \text{carry})}\), 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 \(t = \text{tail} \mathbin{\text{s}} \mathbin{\text{carry}} \mathbin{\text{s}}\) and applying the theorem from left to right we obtain

\[
\Sigma s = \Sigma (\text{tail} \mathbin{\text{s}} - \mathbin{\text{s}} + \text{repeat (head s))}
\]

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

\[
\Sigma (\text{nat} \cdot c \cdot \text{nat}) = \begin{cases} \text{perturbation, head (nat} \cdot c \cdot \text{nat}) = 0 \end{cases} \Sigma (\text{tail (nat} \cdot c \cdot \text{nat})) \mathbin{\text{carry}} \text{nat} \cdot c \cdot \text{nat} = \begin{cases} \text{definition of nat} \\ \Sigma ((\text{nat} + 1) \cdot c \cdot (\text{nat} + 1)) - (\text{nat} \cdot c \cdot \text{nat}) = \begin{cases} \text{summation law} \\ c \cdot \Sigma (\text{nat} \cdot c \cdot \text{nat}) + (c \cdot (\text{nat} \cdot 1)) - (c \cdot 1) - (\text{nat} \cdot c \cdot \text{nat}) = \begin{cases} \text{summation law} \\ c \cdot \Sigma (\text{nat} \cdot c \cdot \text{nat}) + (c \cdot (\text{nat} \cdot 1)) - (c \cdot 1) - (\text{nat} \cdot c \cdot \text{nat}) \end{cases} \end{cases} \end{cases} \end{cases}
\]

The sum \(\Sigma (\text{nat} \cdot c \cdot \text{nat})\) 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 \(\Sigma\), this time as a second-order fixed point.

\[
\Sigma s = \sum (\text{repeat (head s)} + \Sigma (\text{tail s})
\]

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, \(a_0, a_1, a_2, \ldots\), repre-
sents 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

\[
\begin{align*}
\text{const} & : (\text{Num } a) \Rightarrow a \rightarrow \text{Stream } a \\
\text{const } n & = n \times \text{repeat } 0 \\
z & : (\text{Num } a) \Rightarrow \text{Stream } a \\
z & = 0 < 1 \times \text{repeat } 0
\end{align*}
\]

The sum of two power series is implemented by \( + \). The successor function, for instance, is \( \text{const } 1 + z \). The product of two series, however, is not given by \( \ast \) since, for example, \( (\text{const } 1 + z) \ast (\text{const } 1 + 2z) = \text{const } 1 + 3z \). 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.

\[
s = \text{const } (\text{head } s) + z \times \text{tail } s
\]

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

\[
s \times t = \begin{cases} \text{Horner’s rule} & \\
\text{arithmetic} & \\
\text{Horner’s rule} & \\
\text{arithmetic} & \\
\text{Horner’s rule} & \\
\text{arithmetic} & \\
\text{Horner’s rule} & \\
\text{arithmetic} & \\
\text{Horner’s rule} & \\
\end{cases} \\
\begin{align*}
\text{repeat } a & = \text{repeat } a \\
\text{repeat } a & = \text{repeat } a
\end{align*}
\]

We reason

\[
\begin{align*}
\text{head } s \ast \text{head } (\text{recip } s) & = 1 \\
\iff & \{ \text{arithmetic} \} \\
\text{head } (\text{recip } s) & = \text{recip } (\text{head } s)
\end{align*}
\]

and

\[
\begin{align*}
\text{const } (\text{head } s) \ast \text{tail } (\text{recip } s) + \text{tail } s \ast \text{recip } s & = 0 \\
\iff & \{ \text{arithmetic} \} \\
\text{recip } s & = \text{recip } (\text{head } s)
\end{align*}
\]

Again replacing \( \text{const } k \times s \) by \( \text{repeat } k \ast s \), we obtain

\[
\text{recip } s = t \text{ where } a = \text{recip } (\text{head } s) \\
t = a \ast \text{repeat } (\neg a) \ast (\text{tail } s \ast t)
\]

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

5.2 Laws

The familiar arithmetic laws also hold for \( \text{const } n, +, -, \times \) 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 \( \text{const } n \) and \( \times \) with \( \text{repeat } n \) and \( \ast \).

As a start, let’s determine the generating function for \( \text{repeat } a \).

\[
\begin{align*}
\text{repeat } a & = a \ast \text{repeat } a \\
\iff & \{ \text{Horner’s rule} \} \\
\text{repeat } a & = \text{repeat } a \ast \text{repeat } a
\end{align*}
\]

The product of the resulting equation, \( s = u \ast (\text{const } 1 - v) \), is quite typical reflecting the shape of streams equations, \( s = h \ast t \).

Geometric sequences are not much harder.

\[
\begin{align*}
\text{repeat } a \ast \text{nat} & = \begin{cases} \text{definition of } \ast \text{ and nat} \} \\
1 \ast \text{repeat } a \ast \text{repeat } a \ast \text{nat} & = \{ \text{Horner’s rule and repeat } k \ast s = \text{const } k \ast s \} \\
\text{const } 1 \ast z \ast \text{const } a \ast \text{repeat } a \ast \text{nat} & = \{ \text{arithmetic} \} \\
\text{repeat } a \ast \text{const } 1 \ast (1 - \text{const } a \ast z) & = \{ \text{Horner’s rule} \} \\
\Sigma s = 0 < \Sigma s + s & = \{ \text{Horner’s rule} \} \\
\Sigma s = z \ast (\Sigma s + s) & = \{ \text{arithmetic} \} \\
\Sigma s = s \ast z \div (\text{const } 1 - z)
\end{align*}
\]

Consequently, \( \text{repeat } a \ast \text{nat} = \text{const } 1 \div (1 - \text{const } a \ast z) \).

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

\[
\begin{align*}
\Sigma s & = 0 < \Sigma s + s \\
\iff & \{ \text{Horner’s rule} \} \\
\Sigma s & = z \ast (\Sigma s + s) \iff \{ \text{arithmetic} \} \\
\Sigma s & = s \ast z \div (\text{const } 1 - z)
\end{align*}
\]
the coefficients works in general, see (Graham et al. 1994, p. 339).

So, we are left with the task of transforming the right-hand side:

\[ \text{repeat } a \not\propto \langle \text{const } 1 - z \rangle \]

\[ \Sigma s \prod s = s \times z \div (\text{const } 1 - z) \]

\[ \text{nat} = z \div (\text{const } 1 - z)^2 \]

**Figure 4.** Laws for generating functions.

This implies that the generating function of the natural numbers is

\[ \text{nat} = \sum (\text{repeat } 1) = z \div (\text{const } 1 - z)^2 \]

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, nat or repeat a \not\propto 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.

\[
\begin{align*}
\text{fib} &= 0 \times \text{fib} + (1 \times \text{fib}) \\
&= \{ \text{Horner’s rule} \} \\
&= z \times (\text{fib} + (\text{const } 1 + z \times \text{fib})) \\
&= \{ \text{arithmetic} \} \\
&= \text{fib} = z \div (\text{const } 1 - z - z^2)
\end{align*}
\]

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 (\(\alpha \not\propto \beta\)).

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

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

\[
\text{repeat } (1 / \alpha - \beta) \ast (\text{repeat } \alpha \not\propto \text{nat} - \text{repeat } \beta \not\propto \text{nat})
\]

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 \(z^2 - 2 \alpha z - 1\) 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, \(\frac{\alpha + \beta}{\alpha} = 2\), and \(\phi = \frac{1}{2} (1 - \sqrt{5})\) are the roots we seek. Consequently, \(1 - z - z^2 = (1 - \phi z)(1 - \phi^2 z)\). Since furthermore \(\phi - \phi^2 = \sqrt{5}\), we have inferred that

\[
\text{fib} = \text{repeat } (1 / \text{sqrt } 5) \ast (\text{repeat } \phi \not\propto \text{nat} - \text{repeat } \phi^2 \not\propto \text{nat})
\]

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

\[
s + t = \text{head } s + \text{head } t \prec \text{tail } s + \text{tail } t
\]

translates to

\[
(\text{nat } + t)(0) = \text{nat } (0) + t(0) \quad \text{and} \quad (s + t)' = s' + t'
\]

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 technique of unique fixed points in passing by: Karczmarczuk sketches a proof of iterate \(f \cdot f = \text{map } f \cdot \text{iterate } f\) 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,

\[ \text{nat} = 1 \prec \text{nat} + 1 \]

is not guarded as \(+\) is not a constructor. However, \(\text{nat}\) 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

\[
\begin{align*}
\text{class Coalgebra } \sigma \text{ where} \\
\text{head} :: \sigma \alpha \rightarrow \alpha \\
\text{tail} :: \sigma \alpha \rightarrow \sigma \alpha
\end{align*}
\]

with Stream an obvious instance of this class. We call an element of \(\sigma \tau\), where \(\sigma\) 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 \textit{behaviour}: do they yield the same head and are the tails related, as well?

$$a \mathrel{R} b \iff \text{head } a = \text{head } b \text{ and } (\text{tail } a) \mathrel{R} (\text{tail } b)$$

A relation \( R \) that satisfies this property is called a \textit{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 \( -\), is the union of all bisimulations.

\[ \sim = \bigcup \{ R \mid \text{R is a bisimulation} \} \]

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 \textit{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 \textit{the same}. This is captured by the

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

\[ s \sim t \iff s = t \]

**Proof.** \( \iff \) trivial as \( = \) is a bisimulation. \( \implies \) This direction can be shown with the Approximation Lemma (Gibbons and Hutton 2005) using the fact that \( \sim \) is a bisimulation. \( \Box \)

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

A.3 The operator \textit{unfold}

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

\[ \text{unfold } \alpha = (\text{Coalgebra } \sigma) \Rightarrow \sigma \alpha \rightarrow \text{Stream } \alpha \]

\[ \text{unfold } s = \text{head } s \prec \text{unfold } (\text{tail } s) \]

From the definition of \textit{unfold} we can derive the following laws.

\[ \text{head } \cdot \text{unfold} = \text{head} \]
\[ \text{tail } \cdot \text{unfold} = \text{unfold } \cdot \text{tail} \]

In fact, \textit{unfold} is the unique solution of these equations.

**Lemma 1** \textit{unfold} is a functional bisimulation.

\[ a \sim \text{unfold } a \]

**Proof.** Using the properties of \textit{unfold} it is straightforward to show that \( R = \{ (\alpha, \text{unfold } a) \mid a \in \sigma \tau \} \) is a bisimulation. \( \Box \)

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

\[ a_1 \sim a_2 \iff \text{unfold } a_1 = \text{unfold } a_2 \]

**Proof.** \( \implies \) We reason

\[ a_1 \sim a_2 \]
\[ \implies \{ \text{Lemma 1 and } \sim \text{ is symmetric and transitive} \} \]
\[ \text{unfold } a_1 = \text{unfold } a_2 \]

\( \iff \{ \text{Coinduction} \} \]
\[ \text{unfold } a_1 = \text{unfold } a_2 \]

\( \iff \): We show that \( R = \{ (a_1, a_2) \mid \text{unfold } a_1 = \text{unfold } a_2 \} \) is a bisimulation. This follows from the properties of \textit{unfold}. \( \Box \)

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).

\[ \text{data } \text{Expr} :: * \rightarrow * \text{ where} \]

\[ \text{Var} :: \text{Stream } \alpha \rightarrow \text{Expr } \alpha \]
\[ \text{Repeat } :: \alpha \rightarrow \text{Expr } \alpha \]
\[ \text{Plus } :: [\text{Num } \alpha] \Rightarrow \text{Expr } \alpha \rightarrow \text{Expr } \alpha \rightarrow \text{Expr } \alpha \]
\[ \text{Nat } :: \text{Expr Integer} \]

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

We turn \textit{Expr} into a coalgebra by transforming the stream equations into definitions for \textit{head} and \textit{tail}: \( s = h \succ t \) becomes \( \text{head } s = h \) and \( \text{tail } s = t \) where \( h \) is \textit{t} with \textit{repeat}, \textit{+} and \textit{nat} replaced by the corresponding constructors \textit{Repeat}, \textit{Plus} and \textit{Nat}.

\[ \text{instance } \text{Coalgebra } \text{Expr where} \]
\[ \text{head } (\text{Var } s) = \text{head } s \]
\[ \text{head } (\text{Repeat } a) = a \]
\[ \text{head } (\text{Plus } e_1 e_2) = \text{head } e_1 + \text{head } e_2 \]
\[ \text{head } \text{Nat } = 0 \]
\[ \text{tail } (\text{Var } s) = \text{Var } (\text{tail } s) \]
\[ \text{tail } (\text{Repeat } a) = \text{Repeat } a \]
\[ \text{tail } (\text{Plus } e_1 e_2) = \text{Plus } (\text{tail } e_1) (\text{tail } e_2) \]
\[ \text{tail } \text{Nat } = \text{Nat } (\text{Repeat } 1) \]

Both \textit{head} and \textit{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, \textit{head} and \textit{tail} may only be invoked on the arguments of a stream operator.

Using \textit{unfold} we can evaluate a stream expression into a stream.

\[ \text{eval} :: \text{Expr } \alpha \rightarrow \text{Stream } \alpha \]
\[ \text{eval} = \text{unfold} \]

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

\[ \text{repeat } k = \text{eval } (\text{Repeat } k) \]
\[ \text{plus } s_1 s_2 = \text{eval } (\text{Plus } (\text{Var } s_1) (\text{Var } s_2)) \]
\[ \text{nat } = \text{eval } \text{Nat} \]

For \textit{plus}, we embed the argument streams using \textit{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 \textit{head} and \textit{tail}, we replaced functions by constructors. In order to prove that the stream equations are satisfied, we have to show that \textit{eval} undoes this conversion step replacing constructors by functions. In other words, we have to show that \textit{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.

\[ t_1 \sim u_1 \text{ and } t_2 \sim u_2 \implies \text{Plus } t_1 t_2 \sim \text{Plus } u_1 u_2 \]
PROOF. Let $R$ be given by the following inductive definition.

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

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 $\{\text{Plus } t_1 t_2\} R \{\text{Plus } u_1 u_2\}$: The definition of $R$ implies that $t_1 R u_1$ and $t_2 R u_2$. Ex hypothesis, $\text{head } t_1 = \text{head } u_1$ and $\{\text{tail } t_1\} R \{\text{tail } u_1\}$, and likewise for $t_2$ and $u_2$.

$$\begin{align*}
\text{head } (\text{Plus } t_1 t_2) &= \{ \text{definition of head} \} \quad \text{tail } (\text{Plus } t_1 t_2) \\
\text{head } t_1 + \text{head } t_2 &= \text{Plus } (\text{tail } t_1) (\text{tail } t_2) \\
\{ \text{ex hypothesis} \} &= \text{R } \{ \text{R is a congruence} \} \\
\text{head } u_1 + \text{head } u_2 &= \text{Plus } (\text{tail } u_1) (\text{tail } u_2) \\
\{ \text{definition of head} \} &= \{ \text{definition of tail} \} \\
\text{head } (\text{Plus } u_1 u_2) &= \text{tail } (\text{Plus } u_1 u_2)
\end{align*}$$

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

Lemma 4 eval is an interpreter.

$$\begin{align*}
\text{eval } (\text{Var } s) &= s \\
\text{eval } (\text{Repeat } k) &= \text{repeat } k \\
\text{eval } (\text{Plus } e_1 e_2) &= \text{plus } (\text{eval } e_1) (\text{eval } e_2) \\
\text{eval } (\text{Nat}) &= \text{nat}
\end{align*}$$

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

$$\begin{align*}
\text{Var } s &\sim s \\
\implies \{ \text{Lemma 3: } \sim \text{ is a congruence} \} \\
\text{Plus } e_1 e_2 &\sim \text{Plus } (\text{eval } e_1) (\text{eval } e_2)
\end{align*}$$

We proceed

$$\begin{align*}
e_1 &\sim \text{Var } (\text{eval } e_1) \text{ and } e_2 \sim \text{Var } (\text{eval } e_2) \\
\implies \{ \text{Lemma 2 } \} \\
\text{eval } (\text{Plus } e_1 e_2) &= \text{eval } (\text{Plus } (\text{eval } e_1) (\text{eval } e_2))
\end{align*}$$

$$\begin{align*}
\{ \text{Definition of plus } \} \\
\text{eval } (\text{Plus } e_1 e_2) &= \text{plus } (\text{eval } e_1) (\text{eval } e_2)
\end{align*}$$

Case $\text{Nat}$: By definition. □

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

$$\begin{align*}
\text{nat} &= \{ \text{definition of nat and eval} \} \\
\text{head Nat} &\prec \text{eval } (\text{tail Nat}) \\
0 &\prec \text{eval } (\text{Plus Nat } (\text{Repeat } 1)) \\
0 &\prec \text{plus nat } (\text{repeat } 1)
\end{align*}$$

A.6 Uniqueness of solutions

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

$$R = \sim \cup \{ (\text{repeat } k, \text{repeat } k) \mid k \in \tau \}$$

$$\cup \{ (\text{plus } s_1 s_2, \text{plus } t_1 t_2) \mid s_1 R t_1 \text{ and } s_2 R t_2 \}$$

$$\cup \{ (\text{nat, nat}) \}$$

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

$$\begin{align*}
\text{head } (\text{plus } s_1 s_2) &= \{ \text{plus satisfies the eqn} \} \\
\text{tail } (\text{plus } s_1 s_2) &= \{ \text{plus satisfies the eqn} \} \\
\text{head } s_1 + \text{head } s_2 &= \text{plus } (\text{tail } s_1) (\text{tail } s_2) \\
\text{head } t_1 + \text{head } t_2 &= \text{plus } (\text{tail } t_1) (\text{tail } t_2) \\
\{ \text{plus satisfies the eqn} \} &= \{ \text{plus satisfies the eqn} \} \\
\text{head } (\text{plus } t_1 t_2) &= \text{tail } (\text{plus } s_1 s_2)
\end{align*}$$

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

References


