JFP: page 1 of 73. © Cambridge University Press 2011
doi:10.1017/S0956796810000213

Concrete stream calculus: An extended study

RALF HINZE
Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
(*e-mail:* ralf.hinze@comlab.ox.ac.uk)

Abstract

This paper shows how to reason about streams concisely and precisely. Streams, infinite sequences of elements, live in a coworld: they are given by a coinductive datatype, operations on streams are implemented by corecursive programs, and proofs are typically concocted using coinduction. This paper offers an alternative to coinduction. Suitably restricted, stream equations possess *unique solutions*. This property gives rise to a simple and attractive proof technique, essentially bringing equational reasoning to the coworld. We redevelop the theory of recurrences, finite calculus and generating functions using streams and stream operators, building on the cornerstone of unique solutions. The paper contains a smörgåsbord of examples: we study recursion elimination, investigate the binary carry sequence, explore Sprague-Grundy numbers and present two proofs of Moessner's Theorem. The calculations benefit from the rich structure of streams. As the type of streams is an applicative functor we can effortlessly lift operations and their properties to streams. In combination with Haskell's facilities for overloading, this greatly contributes to conciseness of notation. The development is indeed 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.

1 Introduction

The cover of my favourite maths book displays a large 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 will not find many index variables or subscripts though we deal with recurrences, summations and power series. Index variables have their rôle, but often they can be avoided by taking a holistic view, 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.

Adopting this holistic view, this paper shows how to reason effectively about streams, combining concision with precision. Streams, infinite sequences of elements, are a simple example of codata: they are given by a coinductive datatype, operations on streams are implemented by corecursive programs. 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 ensuring that the sequence defined does not swallow its own tail. However, if this care is exercised, the equation possesses a *unique solution*, a fact that is not widely appreciated. Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are! This technique makes an interesting alternative to coinduction, the prevalent proof technique for codata. Building on the cornerstone of unique solutions, we redevelop the theory of recurrences, finite calculus and generating functions. We hope that this will contribute to popularising this attractive proof technique.

The paper contains a smörgåsbord of examples: we study recursion elimination, investigate the binary carry sequence, explore Sprague-Grundy numbers and present two proofs of Moessner's Theorem. Rather attractively, the calculations have a strong equational flavour. Furthermore, they benefit greatly from the rich structure of streams: the type of streams is an applicative functor or idiom; streams can be seen as memo-tables or tabulations. The idiomatic structure allows us to effortlessly *lift operations* and their properties to streams, a device we use extensively in combination with Haskell's facilities for overloading. The development is indeed constructive: streams and stream operators are implemented in Haskell, usually by one-liners. Technically, we view Haskell as a meta-language for Set, the category of sets and total functions, avoiding the need to consider $\bot$.

The paper draws heavily from previous work. The two major sources of inspiration were Rutten's work on stream calculus (2003, 2005) and the aforementioned textbook on concrete mathematics (Graham *et al.*, 1994)—hence the title of the paper. Rutten shows that stream equations, which he calls behavioural differential equations, have indeed unique solutions. A detailed discussion of related work can be found in Section 6.

The rest of the paper is structured as follows. Section 2 introduces the basic stream operations and their properties, organised around the different structural views of streams: idioms, tabulations and final coalgebras. Furthermore, it introduces the fundamental proof technique, which exploits uniqueness of solutions. Section 3 shows how to capture recurrences as streams and solves a multitude of recreational puzzles. We study recursion elimination, investigate the binary carry sequence and explore Sprague-Grundy numbers. Section 4 redevelops the theory of finite calculus using streams and stream operators. As a major application, we present a proof of Moessner's Theorem. Section 5 introduces generating functions and explains how to solve recurrences. Moessner's Theorem is reformulated in terms of generating functions, and an application to combinatorial counting is presented. Finally, Section 6 reviews related work and Section 7 concludes.

2 Streams

A stream is an infinite sequence of elements. The type of streams, called *Stream* α, is like Haskell's list datatype $[\alpha]$, except that there is no base constructor so we cannot

Table 1. *Precedences and fixities of operators (including prelude operators)*

Precedence	Left associative	Non-associative	Right-associative
9	$\diamond$		$\cdot$ $\circ$
7	$*$ $/$ **div** **mod** $\backslash$ $\times$ $\div$ $\otimes$		
6	$+$ $-$ $\cup$		
5			$:$ $+\!\!+$ $\prec$ $\curlyvee$ $\lll$
4		$\in$ $==$ $\neq$ $<$ $\leqslant$ $>$ $\geqslant$ $\dot{=}$ etc	
0			$\star$

construct a finite stream. The *Stream* type is not an inductive type, but a *coinductive type*.

$$\textbf{data}\ \mathit{Stream}\ \alpha = \mathit{Cons}\ \{\mathit{head} :: \alpha, \mathit{tail} :: \mathit{Stream}\ \alpha\}$$

$$\begin{aligned} &(\prec) \quad :: \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \\ &a \prec s = \mathit{Cons}\ a\ s \end{aligned}$$

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 let s, t and u range over streams.

Though the type of streams enjoys a deceptively simple definition, it is an infinite product, it exhibits a rich structure: *Stream* is an idiom, a tabulation, a monad, a final coalgebra, and a comonad. The following section, which introduces the most important generic operations on streams, is organised around these different views. Table 1 summarises the operators, listing their precedences and fixities.

2.1 Operations

Idiom Most definitions we encounter later on make use of operations lifted to streams. We obtain these liftings almost for free, as *Stream* is a so-called *applicative functor* or *idiom* (McBride & Paterson, 2008).

$$\begin{aligned} &\textbf{class}\ \mathit{Idiom}\ \iota\ \textbf{where} \\ &\quad \mathit{pure} :: \alpha \to \iota\ \alpha \\ &\quad (\diamond) \quad :: \iota\ (\alpha \to \beta) \to (\iota\ \alpha \to \iota\ \beta) \end{aligned}$$

The constructor class introduces an operation for embedding a value into an idiomatic structure, and an application operator that takes a structure of functions to a function that maps a structure of arguments to a structure of results. The identity type constructor, $\mathit{Id}\ \alpha = \alpha$, is an idiom. Idioms are closed both under type composition, $(\iota \circ \kappa)\ \alpha = \iota\ (\kappa\ \alpha)$, and type pairing, $(\iota \dot{\times} \kappa)\ \alpha = (\iota\ \alpha, \kappa\ \alpha)$. Here are the definitions that turn *Stream* into an idiom.

$$\begin{aligned} &\textbf{instance}\ \mathit{Idiom}\ \mathit{Stream}\ \textbf{where} \\ &\quad \mathit{pure}\ a = s\ \textbf{where}\ s = a \prec s \\ &\quad s \diamond t \quad = (\mathit{head}\ s)\ (\mathit{head}\ t) \prec (\mathit{tail}\ s) \diamond (\mathit{tail}\ t) \end{aligned}$$

The function *pure* constructs a *constant stream*; *pure* 0, for example, yields the infinite sequence of zeros (*A*000004[1]). The method is given by a first-order equation:

[1] Most if not all integer sequences defined in this paper are recorded in Sloane's On-Line Encyclopedia of Integer Sequences (Sloane, 2009). Keys of the form *Annnnnn* refer to entries in that database.

$s = a \prec s$. Alternatively, it can be defined by a second-order equation: $pure\ a = a \prec pure\ a$, which amounts to the λ-lifted version of the actual definition (Danvy, 1999). The first-order equation is preferable, because it constructs a cyclic structure, using only a constant amount of space (Bird, 1998). We let c range over constant streams. Idiomatic application lifts function application pointwise to streams. We refer to *pure* as a *parametrised stream* and to $\diamond$ as a *stream operator*.

Using nested idiomatic applications, we can lift an arbitrary function pointwise to an idiomatic structure. Here are generic combinators for lifting *n*ary operations ($n = 0, 1, 2$).

$$\begin{aligned}
&repeat :: (Idiom\ \iota) \Rightarrow \alpha \to \iota\ \alpha \\
&repeat\ a = pure\ a \\
&map :: (Idiom\ \iota) \Rightarrow (\alpha \to \beta) \to (\iota\ \alpha \to \iota\ \beta) \\
&map\ f\ s = pure\ f \diamond s \\
&zip :: (Idiom\ \iota) \Rightarrow (\alpha \to \beta \to \gamma) \to (\iota\ \alpha \to \iota\ \beta \to \iota\ \gamma) \\
&zip\ g\ s\ t = pure\ g \diamond s \diamond t
\end{aligned}$$

The context $(Idiom\ \iota) \Rightarrow$ constrains the type variable ι to instances of the class *Idiom*. Using *zip* we can, for instance, lift pairing to idioms.

$$\begin{aligned}
&(\star) :: (Idiom\ \iota) \Rightarrow \iota\ \alpha \to \iota\ \beta \to \iota\ (\alpha, \beta) \\
&(\star) = zip\ (,)
\end{aligned}$$

The quizzical '(,)' is Haskell's pairing constructor. We need to introduce a new operator for lifted pairing as we cannot overload '(,)'. Overloading pairing would not be a terribly good idea anyway, since it is a polymorphic operation. If '(,)' has both the type $\alpha \to \beta \to (\alpha, \beta)$ and $Stream\ \alpha \to Stream\ \beta \to Stream\ (\alpha, \beta)$, then expressions such as (s, t) are inherently ambiguous: is (s, t) a pair of streams or a stream of pairs?

For convenience and conciseness of notation, we overload the arithmetic operations to also denote the corresponding operations lifted to streams. In Haskell, this is easily accomplished using the numeric type classes. Here is an excerpt of the code.[2]

```
instance (Num α) ⇒ Num (Stream α) where
  (+)           = zip (+)
  (−)           = zip (−)
  (∗ )          = zip (∗ )
  negate        = map negate   -- unary minus
  fromInteger i = repeat (fromInteger i)
```

This instance declaration allows us, in particular, to use integer constants as streams—in Haskell, the literal 3 abbreviates $fromInteger\ (3 :: Integer)$. This

Somewhat surprisingly, *pure* 0 is not *A*000000. Just in case you were wondering, the first sequence (*A*000001) lists the number of groups of order *n*.

[2] Since *repeat*, *map* and *zip* work for an arbitrary idiom, we should actually be able to define a generic instance of the form $(Idiom\ \iota, Num\ \alpha) \Rightarrow Num\ (\iota\ \alpha)$. Unfortunately, this does not quite work with the Standard Haskell libraries, as *Num* has two super-classes, *Eq* and *Show*, which cannot sensibly be defined generically.

convention effectively overloads integer literals. Depending on the context, i, j, k, m and n either denote an integer or a constant stream of integers.

We shall make intensive use of overloading, going beyond Haskell's predefined numeric classes. For instance, we also lift exponentiation and binomial coefficients to streams: s^t and $\binom{s}{t}$. To reduce clutter, we omit the necessary class and instance declarations—they are entirely straightforward—and just note the use of overloading as we go along.

Of course, the elements of a stream are by no means confined to numbers. Streams of Booleans are equally useful. For this element type, it is convenient to lift and overload the Boolean constants, *false* and *true*, and the Boolean operators, $\neg$, $\wedge$ and $\vee$, to streams. Again, the definitions are straightforward and omitted. One might be tempted to also overload $==$ to denote lifted equality of streams. However, the predefined *Eq* class dictates that $==$ returns a Boolean. Consequently, lifted equality, which returns a stream of Booleans, cannot be made an instance of that class. Instead, we introduce a new operator.

$$(\dot{==}) :: (\mathit{Idiom}\ \iota, \mathit{Eq}\ \alpha) \Rightarrow \iota\ \alpha \to \iota\ \alpha \to \iota\ \mathit{Bool}$$
$$(\dot{==}) = \mathit{zip}\ (==)$$

Likewise, we also define dotted variants of $\neq$, $<$, $\leqslant$, $>$ and $\geqslant$.

Using this vocabulary we can already define the usual suspects: the natural numbers (*A*001477), the factorial numbers (*A*000142), and the Fibonacci numbers (*A*000045).

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

Note that $\prec$ binds less tightly than $+$, see Table 1. 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. As an aside, we will use the convention that the identifier x' denotes the tail of x, and x'' the tail of x'. The Fibonacci numbers provide an example of mutual recursion: fib' refers to fib and vice versa. Actually, in this case mutual recursion is not necessary, as a quick calculation shows: $\mathit{fib}' = 1 \prec \mathit{fib} + \mathit{fib}' = (1 \prec \mathit{fib}) + (0 \prec \mathit{fib}') = (1 \prec \mathit{fib}) + \mathit{fib}$. So, an alternative definition is

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

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

$$\gg\ \mathit{fib}$$
$$\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots \rangle$$
$$\gg\ \mathit{nat} * \mathit{nat}$$
$$\langle 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, \ldots \rangle$$

$\gg$ $fib'^2 - fib * fib''$
$\langle 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, \ldots \rangle$
$\gg$ $fib'^2 - fib * fib'' \doteq (-1)^{nat}$
$\langle True, True, True, True, True, True, True, True, True, True, True, True, True, True, \ldots \rangle$

The part after the prompt, $\gg$, is the user's input. The result of each submission is shown in the subsequent line. This document has been produced using lhs2TEX (Hinze & Löh, 2008). The session displays the actual output of the Haskell interpreter, generated automatically with lhs2TEX's active features.

Interleaving Another important operator is *interleaving* of two streams.

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

Though the symbol is symmetric, $\curlyvee$ is not commutative. Neither is it associative. Let us 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.

$$bin = 0 \prec\ 2 * bin + 1 \curlyvee 2 * bin + 2$$

Since $\curlyvee$ has lower precedence than the arithmetic operators, see Table 1, the right-hand side of the equation above is grouped $0 \prec\ ((2 * bin + 1) \curlyvee (2 * bin + 2))$.

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 'Yes!' Proving the equality of streams and of stream operators is one of our main activities in this paper. However, we postpone a proof of $nat = bin$, until we have the prerequisites at hand.

Many numeric sequences are actually interleavings in disguise: for instance, $(-1)^{nat} = 1 \curlyvee -1$, nat **div** $2 = nat \curlyvee nat$, and nat **mod** $2 = 0 \curlyvee 1$.

Just as binary numbers can be generalised to $n$ary numbers, $\curlyvee$ can be generalised to interleave n argument streams, represented by a list of streams.

$$\begin{aligned} &interleave \qquad\quad :: [Stream\ \alpha] \to Stream\ \alpha \\ &interleave\ (s : x) = head\ s \prec\ interleave\ (x +\!\!+ [tail\ s]) \end{aligned}$$

Tabulation The Fibonacci function is the folklore example of a function whose straightforward definition leads to a very inefficient program.

$$\begin{aligned} \mathscr{F}_0 &= 0 \\ \mathscr{F}_1 &= 1 \\ \mathscr{F}_{n+2} &= \mathscr{F}_n + \mathscr{F}_{n+1} \end{aligned}$$

To compute $\mathscr{F}_n$ a total of $\mathscr{F}_{n+1} - 1$ additions are necessary. In other words, the running time of $\mathscr{F}$ is proportional to the result. By contrast, the stream definition, fib, does not suffer from this problem: to determine the nth element only $n \dot{-} 1 =$

$max\ \{n-1,0\}$ additions are required. The following display visualises the workings of *fib*.

$$\begin{array}{cccccccccccclcccc}
0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & \cdots & & \mathit{fib} & & & \\
\| & \| & \| & \| & \| & \| & \| & \| & \| & \| & \| & \cdots & & \| & & & \\
0 & 1 & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & \cdots & & 0 & \prec & (1 & \prec\ \mathit{fib}) \\
 & + & + & + & + & + & + & + & + & + & + & \cdots & & & & & + \\
 & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & \cdots & & & & & \mathit{fib}
\end{array}$$

The speed-up is, of course, not a coincidence: *fib* is the memoised version of $\mathscr{F}$. In fact, streams are in one-to-one correspondence to functions from the natural numbers:

$$Stream\ \alpha \quad \cong \quad Nat \rightarrow \alpha\,.$$

A stream can be seen as the tabulation of a function from the natural numbers. Conversely, a function of type $Nat \rightarrow \alpha$ can be implemented by looking up a memo-table. Here are the functions that witness the isomorphism.

$$\begin{array}{lll}
\mathbf{data}\ Nat & = 0 \mid Nat + 1 & \\
tabulate & :: (Nat \rightarrow \alpha) \rightarrow Stream\ \alpha & \\
tabulate\ f & = f\ 0 \prec tabulate\ (f \cdot (+1)) & \\
lookup & :: Stream\ \alpha \rightarrow (Nat \rightarrow \alpha) & \\
lookup\ s\ 0 & = head\ s & \\
lookup\ s\ (n+1) & = lookup\ (tail\ s)\ n &
\end{array}$$

We have $\mathscr{F} = lookup\ fib$ and $tabulate\ \mathscr{F} = fib$. Section 3.1 shows how to systematically turn functions given by recurrence relations into streams defined by recursion equations.

For conciseness of notation, we sometimes abbreviate $lookup\ s\ n$ by $(s)_n$, so that, in particular, $head\ s = (s)_0$.

Many idioms are, in fact, tabulations: the diagonal functor $\Delta = Id \mathbin{\dot{\times}} Id$, for instance, tabulates functions from the Booleans as $\Delta\ \alpha \cong Bool \rightarrow \alpha$. In general, the type constructor ι is said to tabulate $\tau \rightarrow$ iff $\iota\ \alpha \cong \tau \rightarrow \alpha$.

The type constructor $\tau \rightarrow$ itself,[3] the so-called *environment monad*, also gives rise to an idiom.

$$\begin{array}{l}
\mathbf{instance}\ Idiom\ (\tau \rightarrow)\ \mathbf{where} \\
\quad pure\ a = \lambda x \rightarrow a \\
\quad f \diamond g \ \ = \lambda x \rightarrow (f\ x)\ (g\ x)
\end{array}$$

Here, *pure* is the K combinator and $\diamond$ is the S combinator from combinatory logic. Like for streams, lifting is defined pointwise: $zip\ (+)\ f\ g = pure\ (+) \diamond f \diamond g = \lambda x \rightarrow f\ x + g\ x$.

[3] In Haskell, the type constructor $\rightarrow$ is curried, it has kind $* \rightarrow * \rightarrow *$. The operator section $\tau \rightarrow$ is shorthand for $(\rightarrow)\ \tau$. Unfortunately, the section is not legal Haskell syntax.

Monad Since $\tau \rightarrow$ is a monad, *Stream* can also be turned into a monad: *return* is *pure* and *join* is given by

$$\begin{aligned} &join :: Stream\ (Stream\ \alpha) \rightarrow Stream\ \alpha \\ &join\ s = head\ (head\ s) \prec join\ (tail\ (tail\ s)). \end{aligned}$$

We will, however, not make use of the monadic structure.

Final coalgebra The stream *nat* is constructed by repeatedly mapping a function over a stream. We can capture this recursion scheme using a combinator.

$$\begin{aligned} &recurse :: (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow Stream\ \alpha) \\ &recurse\ f\ a = s\ \textbf{where}\ s = a \prec map\ f\ s \end{aligned}$$

So, $nat = recurse\ (+1)\ s$.

Alternatively, we can build a stream by repeatedly applying a given function to a given initial seed.

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

So, $iterate\ (+1)\ 0$ is yet another definition of the naturals. In fact, $recurse\ f\ a$ and $iterate\ f\ a$ always yield the same stream (Section 2.5).

The iterator is a special case of the *unfold* or *anamorphism* for the *Stream* datatype, which captures another common corecursion pattern.

$$\begin{aligned} &unfold :: (\sigma \rightarrow \alpha) \rightarrow (\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow Stream\ \alpha) \\ &unfold\ g\ f\ a = g\ a \prec unfold\ g\ f\ (f\ a) \end{aligned}$$

The type σ can be seen as a type of states. The anamorphism implements an automaton over this state space: the third argument is the initial state; the second argument is the transition function that maps the current state to the next state; and the first argument is the output function that maps the current state to an 'output symbol'.

The functions *iterate* and *unfold* are inter-definable: $iterate\ f = unfold\ id\ f$ and conversely $unfold\ h\ t = map\ h \cdot iterate\ t$. We prefer *iterate* over *unfold*, since the extra generality is barely needed. Section 3.1 explains why.

Comonad Finally, *Stream* has the structure of a *comonad*, the categorical dual of a *monad*: *head* is the counit and *tails*, defined below, is the comultiplication.

$$\begin{aligned} &tails :: Stream\ \alpha \rightarrow Stream\ (Stream\ \alpha) \\ &tails = iterate\ tail \end{aligned}$$

We shall neither make use of the comonadic structure.

2.2 Stream comprehensions

Throughout the paper we emphasise the holistic view on streams. However, from time to time it is useful to define a stream pointwise or to relate a given stream to a pointwise definition. *Parallel stream comprehensions* allow us to do exactly this.

$$\langle e \mid x_1 \leftarrow e_1 \mid \ldots \mid x_n \leftarrow e_n \rangle = pure\ (\lambda x_1 \ldots x_n \rightarrow e) \diamond e_1 \diamond \cdots \diamond e_n$$

In fact, since the notation is defined in terms of the idiomatic primitives, it makes sense for an arbitrary idiom. The notational advantage of a comprehension over the idiomatic expression on the right-hand side is that the bound variable x_i and the stream it iterates over e_i are adjacent. For streams, the order of the binders is not significant: any permutation of $x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n$ generates the same stream. This does not hold for arbitrary idioms: in the *IO* idiom, for instance, the order is significant.

The functional programming language Miranda[4] supports a second form of generators, so-called *iterative generators*, which capture recurrence relations.

$$\langle e_1 \mid x \leftarrow e_2, e_3 \,..\rangle = \mathit{pure}\ (\lambda x \rightarrow e_1) \diamond \mathit{iterate}\ (\lambda x \rightarrow e_3)\ e_2$$

The iterative generator avoids the duplication of the binder '$\lambda x \rightarrow$' on the right-hand side. In parallel stream comprehensions, iterative generators can be freely mixed with ordinary generators. As an example, here is an alternative definition of *fib*

$$\mathit{fib} = \langle a \mid (a, b) \leftarrow (0, 1), (b, a + b) \,..\rangle,$$

which uses tupling to achieve linear running time. We will derive this version in Section 3.2.1.

2.3 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; when viewed as stream equations they are ambiguous.[5] 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* (Rutten, 2003, Theorem 3.1). 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 other stream identifiers. However, neither h nor t may contain *head* or *tail*.

If x is a parametrised stream or a stream operator,

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

then h and t may use $\mathit{head}\ x_i$ or $\mathit{tail}\ x_i$ provided x_i is of the right type. Apart from that, no other uses of *head* or *tail* are permitted. Equations of this form are called *admissible*.

[4] Miranda is a registered trademark of Research Software Limited.

[5] There is a slight mismatch between the theoretical framework of streams and the Haskell implementation of streams. Since products are lifted in Haskell, *Stream* τ additionally contains partial streams such as $\bot$, $a_0 \prec \bot$, $a_0 \prec a_1 \prec \bot$ and so forth. We ignore this extra complication, viewing Haskell as a meta-language for Set.

Looking back, we find that the definitions we have encountered so far, including those of *pure*, $\diamond$ and $\curlyvee$, are admissible. In general, we consider systems of admissible equations to cater for mutually recursive definitions. In fact, most of the definitions cannot be read in isolation. For instance, *nat*'s defining equation, $nat = 0 \prec nat + 1$, silently refers to *pure* and $\diamond$, so *nat* is actually given by a system of three equations

$$\begin{aligned} nat &= 0 \prec pure\ (+) \diamond nat \diamond pure\ 1 \\ pure\ a &= a \prec pure\ a \\ s \diamond t &= (head\ s)\ (head\ t) \prec (tail\ s) \diamond (tail\ t), \end{aligned}$$

all of which are admissible.

If the requirements are violated, then a simple rewrite sometimes saves the day. As an example, the folklore definition of *fib*

$$fib = 0 \prec 1 \prec fib + tail\ fib$$

is not admissible, since *tail* is applied to the stream defined. We can, however, eliminate the call to *tail* by introducing a name for it. If we replace *tail fib* by fib', we obtain the two equations given in Section 2.1. The alternative definition $fib = 0 \prec (1 \prec fib) + fib$ is also admissible. Nested occurrences of $\prec$ are unproblematic since they can be seen as shorthand for mutually recursive equations: $fib = 0 \prec cons\ 1\ fib + fib$ and $cons\ h\ t = h \prec t$.

Sometimes the situation is more subtle. The two stream operators

$$\begin{aligned} even, odd &:: Stream\ \alpha \to Stream\ \alpha \\ even\ s &= head\ s \prec odd\ \ (tail\ s) \\ odd\ s &= \phantom{head\ s \prec{}} even\ (tail\ s) \end{aligned}$$

are well-defined, even though *odd* does not meet the syntactic requirements. If $s = x \curlyvee y$, then $even\ s = x$ and $odd\ s = y$. The operators are ruled out, however, since their *use* in other recursive definitions is possibly problematic: the equation $x = 0 \prec even\ x$, for instance, is admissible, yet it has infinitely many solutions.

Not all is lost, however. Consider as another example the equation

$$s = a \prec f\ (even\ s) \curlyvee g\ (odd\ s)$$

or, equivalently,

$$x \curlyvee y = a \prec f\ x \curlyvee g\ y, \tag{1}$$

where f and g are some stream operators. In this case, the equation uniquely determines s, and hence x and y, since it is equivalent to the recursive equation $x = a \prec g\ (f\ x)$ and the non-recursive $y = f\ x$. So, we may view Equation (1) as a notational shorthand for the two latter equations. The scheme nicely captures alternations, for instance,

$$d \curlyvee i = 1 \prec d * 2 \curlyvee i + 1$$

alternates between doubling and incrementing (*A*075427).

By the way, non-recursive definitions like

$$nat' = nat + 1$$

are unproblematic and unrestricted as they can always be inlined.

2.4 Laws

This section provides an account of the most important properties of streams, serving mainly as a handy reference. If you wish, you can skim through the material or proceed directly to Section 2.5.

We have noted before that the type *Stream* has a rich structure: it is an idiom, a tabulation and a final coalgebra. The laws are categorised according to these different views.

First of all, streams enjoy the following *extensionality* property.

$$s \quad = \quad head\ s \prec tail\ s \tag{2}$$

Idiom Streams satisfy the four *idiom laws.*

$$\begin{aligned} &pure\ id \diamond u &&= u && \text{(identity)} \\ &pure\ (\cdot) \diamond u \diamond v \diamond w &&= u \diamond (v \diamond w) && \text{(composition)} \\ &pure\ f \diamond pure\ x &&= pure\ (f\ x) && \text{(homomorphism)} \\ &u \diamond pure\ x &&= pure\ (\lambda f \to f\ x) \diamond u && \text{(interchange)} \end{aligned}$$

The first two imply the well-known *functor laws*: *map* preserves identity and composition (hence the names of the idiom laws).

$$\begin{aligned} map\ id &= id \\ map\ (f \cdot g) &= map\ f \cdot map\ g \end{aligned}$$

Index manipulations are often applications of the second functor law, which becomes evident, if we rephrase the laws using stream comprehensions.

$$\langle a \mid a \leftarrow s \rangle \quad = \quad s \tag{3}$$

$$\langle f\ (g\ a) \mid a \leftarrow s \rangle \quad = \quad \langle f\ b \mid b \leftarrow \langle g\ a \mid a \leftarrow s \rangle \rangle \tag{4}$$

To illustrate, here is a short calculation that demonstrates a simple index manipulation.

$$\begin{aligned} &\langle 2 * i \mid i \leftarrow nat + 1 \rangle \\ = \quad &\{\ nat + 1 = \langle j + 1 \mid j \leftarrow nat \rangle\ \} \\ &\langle 2 * i \mid i \leftarrow \langle j + 1 \mid j \leftarrow nat \rangle \rangle \\ = \quad &\{\ \text{functor law (4)}\ \} \\ &\langle 2 * (j + 1) \mid j \leftarrow nat \rangle \end{aligned}$$

Every structure comes equipped with structure-preserving maps; so do idioms: a natural transformation $h :: \iota\ \alpha \rightarrow \kappa\ \alpha$ is an *idiom homomorphism* iff

$$h\,(pure\ a) \quad = \quad pure\ a \tag{5}$$

$$h\,(x \diamond y) \quad = \quad h\ x \diamond h\ y \tag{6}$$

The two conditions imply, in fact, the naturality property: $map\ f \cdot h = h \cdot map\ f$. The identity function is an idiom homomorphism, and homomorphisms are closed under composition. The projection function *head* is a homomorphism (from *Stream* to the identity idiom *Id*), so is *tail*. The latter projection is an *idiom endomorphism*, a homomorphism from an idiom to itself. Also *pure* itself is a homomorphism (from *Id* to the idiom). Condition (6) for *pure* is equivalent to the third idiom law (hence its name).

This leaves us to explain the fourth idiom law: it captures the fact that a pure, effect-free computation commutes with an impure, effectful computation.

Lifted pairing $\star$ satisfies

$$map\ fst\ (s \star t) \quad = \quad s \tag{7}$$

$$map\ snd\ (s \star t) \quad = \quad t \tag{8}$$

$$map\ fst\ p \star map\ snd\ p \quad = \quad p \tag{9}$$

It is important to note that the Laws (7)–(9) do not hold for arbitrary idioms. Perhaps surprisingly, (the uncurried version of) $\star$ is also a homomorphism (from $\Delta \cdot Stream$ to $Stream \cdot \Delta$). In the case of streams or, more generally, *commutative idioms*, $\star$ is even an idiom isomorphism, an idiom homomorphism which has an inverse.

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

Interleaving The interleaving operator interacts nicely with lifting.

$$pure\ a \curlyvee pure\ a \quad = \quad pure\ a \tag{10}$$

$$(s_1 \diamond s_2) \curlyvee (t_1 \diamond t_2) \quad = \quad (s_1 \curlyvee t_1) \diamond (s_2 \curlyvee t_2) \tag{11}$$

A simple consequence is $(s \curlyvee t)+1 = s+1 \curlyvee t+1$ or, more generally, $map\ f\ (s \curlyvee t) = map\ f\ s \curlyvee map\ g\ t$. The two laws show, in fact, that interleaving is a homomorphism (from $\Delta \cdot Stream$ to *Stream*). Interleaving is even an isomorphism; the reader is encouraged to work out the details.

Property (11) is also 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.

$$\begin{array}{ccc} s_1 & \diamond & s_2 \\ & \curlyvee & \\ t_1 & \diamond & t_2 \end{array} \; = \; \begin{array}{ccc} s_1 & & s_2 \\ \curlyvee & \diamond & \curlyvee \\ t_1 & & t_2 \end{array} \qquad\qquad \frac{s_1 \mid s_2}{t_1 \mid t_2} \; = \; \left.\frac{s_1}{t_1}\,\right|\,\frac{s_2}{t_2}$$

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

due to Bird (1987). The geometrical interpretation can be emphasised further by writing the two operators | and –, like on the right-hand side (Fokkinga, 1992).

Tabulation The functions *lookup* and *tabulate* are mutually inverse and they satisfy the following naturality properties.

$$\begin{aligned} \mathit{map}\ f \cdot \mathit{tabulate} &= \mathit{tabulate} \cdot (f\ \cdot) \\ (f\ \cdot) \cdot \mathit{lookup} &= \mathit{lookup} \cdot \mathit{map}\ f \end{aligned}$$

Note that post-composition $(f\ \cdot)$ is the mapping function for the functor $\tau \rightarrow$. The laws are somewhat easier to memorise, if we write them in a pointwise style.

$$\mathit{map}\ f\ (\mathit{tabulate}\ g) \quad = \quad \mathit{tabulate}\ (f \cdot g) \tag{12}$$

$$f \cdot \mathit{lookup}\ t \quad = \quad \mathit{lookup}\ (\mathit{map}\ f\ t) \tag{13}$$

Furthermore, *tabulate* and *lookup* are idiom isomorphisms between *Stream* and *Nat* →. We will exploit this fact in Section 3.2.1, when we look at tabulation in more detail.

Final coalgebra On streams, recursion and iteration coincide.

$$\mathit{recurse}\ f\ a \quad = \quad \mathit{iterate}\ f\ a$$

The function *iterate* satisfies an important fusion law, which amounts to the free theorem of $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \mathit{Stream}\ \alpha)$.

$$\mathit{map}\ h \cdot \mathit{iterate}\ f_1 = \mathit{iterate}\ f_2 \cdot h \quad \Longleftarrow \quad h \cdot f_1 = f_2 \cdot h \tag{14}$$

The *unfold* or anamorphism of the *Stream* datatype is characterised by the following *universal property*.

$$h = \mathit{unfold}\ g\ f \quad \Longleftrightarrow \quad \mathit{head} \cdot h = g \ \text{ and } \ \mathit{tail} \cdot h = h \cdot f \tag{15}$$

Read from left to right the universal property states that *unfold* $g\ f$ is a solution of the equations $\mathit{head} \cdot h = g$ and $\mathit{tail} \cdot h = h \cdot f$. Read from right to left the property asserts that *unfold* $g\ f$ is indeed the unique solution.

The universal property has a multitude of consequences. Instantiating g to *head* and f to *tail*, we obtain the *reflection law.*

$$\mathit{unfold}\ \mathit{head}\ \mathit{tail} \quad = \quad \mathit{id}$$

Instantiating h to *unfold* $g\ f$, gives the two *computation rules*, which are point-free versions of the defining equation of *unfold*.

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

Since the type of *unfold* involves two type variables, *unfold* enjoys two fusion laws: *fusion* and *functor fusion.*

$$\mathit{unfold}\ g\ f \cdot h = \mathit{unfold}\ g'\ f' \quad \Longleftarrow \quad g \cdot h = g' \ \text{ and } \ f \cdot h = h \cdot f'$$

$$\mathit{map}\ h \cdot \mathit{unfold}\ g\ f \quad = \quad \mathit{unfold}\ (h \cdot g)\ f$$

Both are, in fact, also consequences of the *iterate*-fusion law (14).

As an aside, the pair of functions $(head, tail)$ forms a so-called coalgebra for the functor $F\ X = A \times X$. The stream coalgebra is final as there is a unique coalgebra homomorphism from every other coalgebra (g, f) to the stream coalgebra: $unfold\ g\ f$—this is the import of the universal property (15).

2.5 Proofs

The last section has listed a multitude of laws. It is high time that we show how to prove them correct. In Section 2.3 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 $x = \phi\ x$ is an admissible equation, we denote its unique solution by $fix\ \phi$. (The equation implicitly defines a function in x. 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 property.

$$fix\ \phi = s \iff \phi\ s = s$$

Read from left to right it states that $fix\ \phi$ is indeed a solution of $x = \phi\ x$. Read from right to left it asserts that any solution is equal to $fix\ \phi$. The universal property of *unfold* is an instance of this scheme. Now, if we want to prove $s = t$ where $s = fix\ \phi$, then it suffices to show that $\phi\ t = t$.

As a first example, let us prove the *idiom homomorphism law*.

$$\begin{aligned}
& pure\ f \diamond pure\ a \\
= \;& \{\ \text{definition of } \diamond\ \} \\
& (head\ (pure\ f))\ (head\ (pure\ a)) \prec tail\ (pure\ f) \diamond tail\ (pure\ a) \\
= \;& \{\ \text{definition of } pure\ \} \\
& f\ a \prec pure\ f \diamond pure\ a
\end{aligned}$$

Consequently, $pure\ f \diamond pure\ a$ equals the unique solution of $x = f\ a \prec x$, which by definition is $pure\ (f\ a)$.

That was easy. The next proof is not much harder. We show that the natural numbers are even and odd numbers interleaved: $nat = 2 * nat \curlyvee 2 * nat + 1$.

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

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

Now, if both s and t are given as fixed points, $s = fix\ \phi$ and $t = fix\ \psi$, 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.

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

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*: $fib'^2 - fib * fib'' = (-1)^{nat}$ (recall that exponentiation is also lifted to streams).

$$\begin{array}{ll} & fib'^2 - fib * fib'' \\ = & \{\ \text{definition of } fib'' \text{ and arithmetic}\ \} \\ & fib'^2 - (fib^2 + fib * fib') \\ = & \{\ \text{definition of } fib \text{ and definition of } fib'\ \} \\ & 1 \prec (fib''^2 - (fib'^2 + fib' * fib'')) \\ = & \{\ fib'' - fib' = fib \text{ and arithmetic}\ \} \\ & 1 \prec (-1) * (fib'^2 - fib * fib'') \\ \subset & \{\ x = 1 \prec (-1) * x \text{ has a unique solution}\ \} \\ & 1 \prec (-1) * (-1)^{nat} \\ = & \{\ \text{definition of } nat \text{ and arithmetic}\ \} \\ & (-1)^{nat} \end{array}$$

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 and stream operators! As an example, let us prove the abide law by showing $f = g$ where

$$f\ s_1\ s_2\ t_1\ t_2 = (s_1 \diamond s_2) \curlyvee (t_1 \diamond t_2) \quad \text{and} \quad g\ s_1\ s_2\ t_1\ t_2 = (s_1 \curlyvee t_1) \diamond (s_2 \curlyvee t_2).$$

The proof is straightforward involving only bureaucratic steps.

$$\begin{aligned}
& f\ a\ b\ c\ d \\
= \;& \{ \text{definition of } f \} \\
& (a \diamond b) \curlyvee (c \diamond d) \\
= \;& \{ \text{definition of } \diamond \text{ and definition of } \curlyvee \} \\
& head\ a \diamond head\ b \prec (c \diamond d) \curlyvee (tail\ a \diamond tail\ b) \\
= \;& \{ \text{definition of } f \} \\
& head\ a \diamond head\ b \prec f\ c\ d\ (tail\ a)\ (tail\ b) \\
\subset \;& \{ x\ s_1\ s_2\ t_1\ t_2 = head\ s_1 \diamond head\ s_2 \prec x\ t_1\ t_2\ (tail\ s_1)\ (tail\ s_2) \text{ has a unique solution} \} \\
& head\ a \diamond head\ b \prec g\ c\ d\ (tail\ a)\ (tail\ b) \\
= \;& \{ \text{definition of } g \} \\
& head\ a \diamond head\ b \prec (c \curlyvee tail\ a) \diamond (d \curlyvee tail\ b) \\
= \;& \{ \text{definition of } \diamond \text{ and definition of } \curlyvee \} \\
& (a \curlyvee c) \diamond (b \curlyvee d) \\
= \;& \{ \text{definition of } g \} \\
& g\ a\ b\ c\ d
\end{aligned}$$

Henceforth, we 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 (14). The 'unique fixed-point proof' is short and sweet; it compares favourably to the ones given by Gibbons & Hutton (2005).

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

Unsurprisingly, the linking equation $g\ a = h\ a \prec g\ (f_1\ a)$ corresponds to the *unfold* operator, which as we have noted can be defined in terms of *map* and *iterate*.

The fusion law implies $map\ f \cdot iterate\ f = iterate\ f \cdot f$, which is the key for proving $nat = iterate\ (+1)\ 0$, or, more generally, $recurse\ f\ a = iterate\ f\ a$. We show that $iterate\ f\ a$ is the unique solution of $x = a \prec map\ f\ x$.

$$\begin{aligned} & iterate\ f\ a \\ = & \quad \{ \text{ definition of } iterate \} \\ & a \prec iterate\ f\ (f\ a) \\ = & \quad \{ \text{ iterate fusion law (14): } h = f_1 = f_2 = f \} \\ & a \prec map\ f\ (iterate\ f\ a) \end{aligned}$$

Let us conclude by noting that the proof techniques are not always applicable. For example, they cannot be used to show the *extensionality* law (2), $s = head\ s \prec tail\ s$, simply because s is not given by a recursion equation. In this case, we have to resort to coinduction.

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

Using $\prec$ and $\curlyvee$ we can easily capture recurrences or functions from the natural numbers. Though functions from the naturals and streams are in one-to-one correspondence, a stream is often easier to manipulate than a function or a recurrence, because a stream is a single entity, typically defined by a single equation. Before we consider concrete examples, we first explore tabulation in more depth.

3.1 Tabulation

The simplest recurrences are of the form $a_0 = k$ and $a_{n+1} = f(a_n)$, for some natural number k and some function f on the naturals. This recurrence is captured by the stream equation $a = k \prec map\ f\ a$, or more succinctly by $recurse\ f\ k$. Though fairly obvious, the relation is worth exploring. On the face of it, the linear recurrence corresponds to the *fold* or *catamorphism* of the inductive type *Nat*.

$$\begin{aligned} & fold && :: (\alpha \to \alpha) \to \alpha \to (Nat \to \alpha) \\ & fold\ s\ z\ 0 && = z \\ & fold\ s\ z\ (n+1) && = s\ (fold\ s\ z\ n) \end{aligned}$$

Catamorphisms are dual to anamorphisms, enjoying a dual characterisation.

$$h = fold\ s\ z \iff h\ 0 = z \text{ and } h \cdot (+1) = s \cdot h$$

Some simple consequences of the universal property are the *reflection law*, $fold\ (+1)\ 0 = id$, and the *computation rules*, $fold\ s\ z\ 0 = z$ and $fold\ s\ z \cdot (+1) = s \cdot fold\ s\ z$.

We already noted that tabulating $fold\ s\ z$ gives $recurse\ s\ z$. The proof of this fact makes crucial use of *tabulate*'s naturality property.

$$\begin{aligned} & tabulate\ (fold\ s\ z) \\ = & \quad \{ \text{ definition of } tabulate \} \\ & fold\ s\ z\ 0 \prec tabulate\ (fold\ s\ z \cdot (+1)) \end{aligned}$$

$$
\begin{aligned}
&= \quad \{\text{ computation rules, see above }\}\\
&\quad\; z \prec \mathit{tabulate}\ (s \cdot \mathit{fold}\ s\ z)\\
&= \quad \{\text{ naturality of } \mathit{tabulate}\ (12)\ \}\\
&\quad\; z \prec \mathit{map}\ s\ (\mathit{tabulate}\ (\mathit{fold}\ s\ z))
\end{aligned}
$$

Consequently, there are, at least, four equivalent ways of expressing the linear recurrence $a_0 = z$ and $a_{n+1} = s(a_n)$.

$$\mathit{tabulate}\ (\mathit{fold}\ s\ z) = \mathit{recurse}\ s\ z = \mathit{iterate}\ s\ z = \langle x \mid x \leftarrow z, s\ x \mathinner{..}\rangle$$

Using the reflection law for Nat, this furthermore implies that nat is the tabulation of the identity function: $\mathit{tabulate}\ \mathit{id} = \mathit{tabulate}\ (\mathit{fold}\ (+1)\ 0) = \mathit{iterate}\ (+1)\ 0 = \mathit{nat}$.

If we 'un-tabulate' *iterate*, setting $\mathit{loop}\ f = \mathit{lookup} \cdot \mathit{iterate}\ f$, we obtain a *tail-recursive* function, which can be seen as the counterpart of Haskell's *foldl* for natural numbers.

$$
\begin{aligned}
&\mathit{loop} && :: (\alpha \to \alpha) \to \alpha \to (\mathit{Nat} \to \alpha)\\
&\mathit{loop}\ f\ a\ 0 && = a\\
&\mathit{loop}\ f\ a\ (n+1) && = \mathit{loop}\ f\ (f\ a)\ n
\end{aligned}
$$

Mathematically speaking, *loop* programs n-fold composition: $\mathit{loop}\ f\ a\ n = f^{(n)}\ a$.

The fact that nat tabulates id has a very simple, yet important consequence:

$$
\begin{aligned}
&\quad\; \mathit{tabulate}\ f\\
&= \quad \{\ f \cdot \mathit{id} = f\ \}\\
&\quad\; \mathit{tabulate}\ (f \cdot \mathit{id})\\
&= \quad \{\text{ naturality of } \mathit{tabulate}\ (12)\ \}\\
&\quad\; \mathit{map}\ f\ (\mathit{tabulate}\ \mathit{id})\\
&= \quad \{\ \mathit{tabulate}\ \mathit{id} = \mathit{nat}\ \}\\
&\quad\; \mathit{map}\ f\ \mathit{nat}\\
&= \quad \{\text{ stream comprehensions }\}\\
&\quad\; \langle f\ n \mid n \leftarrow \mathit{nat}\rangle.
\end{aligned}
$$

Setting $f = \mathit{lookup}\ s$ in the equation above, gives the following *extensionality law*

$$s \quad = \quad \langle (s)_n \mid n \leftarrow \mathit{nat}\rangle, \tag{16}$$

which allows us to switch swiftly between a pointwise and a point-free or holistic view.

Now, let us tackle a slightly more involved class of recurrences. The sequence given by the 'binary' recurrence $a_0 = k$, $a_{2n+1} = f(a_n)$ and $a_{2n+2} = g(a_n)$ corresponds to the stream $a = k \prec \mathit{map}\ f\ a \curlyvee \mathit{map}\ g\ a$. Turning to the proof, observe that the extensionality law (16) implies $a = \langle a_n \mid n \leftarrow \mathit{nat}\rangle = \langle a_n \mid n \leftarrow \mathit{bin}\rangle$. The calculation below then proves that the latter expression satisfies the stream equation

above.

$$
\begin{aligned}
& \langle a_n \mid n \leftarrow bin \rangle \\
= \; & \{ \text{ definition of } bin \text{ and abide law (11) } \} \\
& a_0 \prec \langle a_n \mid n \leftarrow 2 * bin + 1 \rangle \curlyvee \langle a_n \mid n \leftarrow 2 * bin + 2 \rangle \\
= \; & \{ \text{ functor law (4) } \} \\
& a_0 \prec \langle a_{2*n+1} \mid n \leftarrow bin \rangle \curlyvee \langle a_{2*n+2} \mid n \leftarrow bin \rangle \\
= \; & \{ \text{ definition of } a \ \} \\
& k \prec \langle f(a_n) \mid n \leftarrow bin \rangle \curlyvee \langle g(a_n) \mid n \leftarrow bin \rangle \\
= \; & \{ \text{ functor law (4) } \} \\
& k \prec map\ f\ \langle a_n \mid n \leftarrow bin \rangle \curlyvee map\ g\ \langle a_n \mid n \leftarrow bin \rangle
\end{aligned}
$$

The crucial step in the derivation is easily overlooked: it is the application of $nat = bin$ that makes the calculation fly.

To tabulate an arbitrary recurrence or an arbitrary function from the naturals, we can either apply *tabulate* or simply map the function over the stream of natural numbers. Of course, if f is recursively defined, $f = \phi\ f$, then this is a wasted opportunity, as the recursive calls do not re-use any of the tabulated values. In order to also replace these calls by table look-ups, we invoke the *rolling rule* of fixed-point calculus (Backhouse, 2001).

$$
\begin{aligned}
& fix\ \phi \\
= \; & \{ \ lookup \cdot tabulate = id \ \} \\
& fix\ (lookup \cdot tabulate \cdot \phi) \\
= \; & \{ \text{ rolling rule: } fix\ (f \cdot g) = f\ (fix\ (g \cdot f)) \ \} \\
& lookup\ (fix\ (tabulate \cdot \phi \cdot lookup))
\end{aligned}
$$

The calculation transforms a fixed point over functions into a fixed point over streams. Introducing names for the participants, we obtain $f = lookup\ s$ and $s = tabulate\ (\phi\ f)$. While the derivation is completely general, each call of f now involves a linear look-up. By contrast, the tabulations for linear and 'binary' recurrences avoid the look-up by maintaining 'pointers' into the memo table. As an aside, the derivation above is an instance of the *worker-wrapper transformation* (Gill & Hutton, 2009).

3.2 Examples

In the previous section we have considered abstract recurrences. Let us tackle some concrete examples now. In the next section we re-visit our all-time-favourite, the Fibonacci numbers, and then move on to examine some less known examples in the sections thereafter.

3.2.1 Fibonacci numbers

There and Back Again
John Ronald Reuel Tolkien (1937)

The recurrence of $\mathscr{F}$, see Section 2.1, does not fit the schemes previously discussed, so we have to start afresh. (We could apply the general scheme for recursive functions, but this involves linear look-ups.) The calculations are effortless, however, if we make use of the fact that *tabulate* is an idiom homomorphism. Using the environment idiom we can rewrite the last equation of $\mathscr{F}$ in a point-free style: $\mathscr{F} \cdot (+2) = \mathscr{F} + \mathscr{F} \cdot (+1)$.

$$\begin{aligned}
& \mathit{tabulate}\ \mathscr{F} \\
= \ & \{ \text{definition of } \mathit{tabulate}\text{, twice} \} \\
& \mathscr{F}_0 \prec \mathscr{F}_1 \prec \mathit{tabulate}\ (\mathscr{F} \cdot (+2)) \\
= \ & \{ \text{definition of } \mathscr{F} \} \\
& 0 \prec 1 \prec \mathit{tabulate}\ (\mathscr{F} + \mathscr{F} \cdot (+1)) \\
= \ & \{ \mathit{tabulate} \text{ is an idiom homomorphism} \} \\
& 0 \prec 1 \prec \mathit{tabulate}\ \mathscr{F} + \mathit{tabulate}\ (\mathscr{F} \cdot (+1)) \\
= \ & \{ \text{definition of } \mathit{tabulate} \} \\
& 0 \prec 1 \prec \mathit{tabulate}\ \mathscr{F} + \mathit{tail}\ (\mathit{tabulate}\ \mathscr{F})
\end{aligned}$$

The only non-trivial step is the second but last one, which employs $\mathit{tabulate}\ (f+g) = \mathit{tabulate}\ f + \mathit{tabulate}\ g$, which in turn is syntactic sugar for $\mathit{tabulate}\ (\mathit{pure}\ (+) \diamond f \diamond g) = \mathit{pure}\ (+) \diamond \mathit{tabulate}\ f \diamond \mathit{tabulate}\ g$. Since *tabulate* preserves the idiomatic structure, the derivation goes through nicely.

Tabulation and look-up allow us to switch swiftly between functions from the naturals and streams. So, even if coinductive structures are not available in your language of choice, you can still use stream calculus for program transformations. Let us illustrate this point by deriving an efficient *iterative* implementation of $\mathscr{F}$. To this end we have to transform *fib* into a form that fits the recursion equation of *iterate* or *recurse*. This can be easily achieved using the standard technique of *tupling* (Bird, 1998), which boils down to an application of lifted pairing $\star$.

$$\begin{aligned}
& \mathit{fib} \star \mathit{fib}' && \text{(A)} \\
= \ & \{ \text{definition of } \mathit{fib} \text{ and definition of } \mathit{fib}' \} \\
& (0 \prec \mathit{fib}') \star (1 \prec \mathit{fib} + \mathit{fib}') \\
= \ & \{ \text{definition of } \star \} \\
& (0,1) \prec \mathit{fib}' \star (\mathit{fib} + \mathit{fib}') \\
= \ & \{ \text{introduce } \mathit{step}\ (a,b) = (b, a+b) \text{ and lift the equality to streams} \} \\
& (0,1) \prec \mathit{map}\ \mathit{step}\ (\mathit{fib} \star \mathit{fib}')
\end{aligned}$$

The last step makes use of the fact that equations between elements, such as $\mathit{step}\ (a,b) = (b, a+b)$, can be lifted to equations between streams, $\mathit{map}\ \mathit{step}\ (s \star t) = t \star (s + t)$, see also (Hinze, 2010). The calculation shows that $\mathit{fib} \star \mathit{fib}'$ satisfies the

recursion equation of *iterate*, we have $fib \star fib' = iterate\ step\ (0,1)$. Consequently,

$$fib = map\ fst\ (iterate\ step\ (0,1)) = unfold\ fst\ step\ (0,1) = \langle a \mid (a,b) \leftarrow (0,1), (b, a+b) \,..\rangle.$$

To obtain an iterative or tail-recursive program, we simply 'un-tabulate' the derived stream expression.

$$\begin{aligned}
& lookup\ fib \\
= {} & \{ \text{ see above } \} \\
& lookup\ (map\ fst\ (iterate\ step\ (0,1))) \\
= {} & \{ \text{ naturality of } lookup \text{ (12) } \} \\
& fst\ (lookup\ (iterate\ step\ (0,1))) \\
= {} & \{ \text{ definition of } loop \} \\
& fst\ (loop\ step\ (0,1))
\end{aligned}$$

An alternative approach is based on the observation that the tails of *fib* are linear combinations of *fib* and fib'. Let i and j be constant streams, then

$$\begin{aligned}
& i * fib + j * fib' \qquad\qquad \text{(B)} \\
= {} & \{ \text{ definition of } fib \text{ and definition of } fib' \} \\
& i * (0 \prec fib') + j * (1 \prec fib + fib') \\
= {} & \{ \text{ arithmetic } \} \\
& j \prec i * fib' + j * (fib + fib') \\
= {} & \{ \text{ arithmetic } \} \\
& j \prec j * fib + (i+j) * fib'.
\end{aligned}$$

Viewing $i * fib + j * fib'$ as a function in (i,j), the calculation shows that the linear combination satisfies the recursion equation of *unfold*, we have $i * fib + j * fib' = unfold\ snd\ step\ (i,j)$ and hence $fib = unfold\ snd\ step\ (1,0)$.

The two iterative versions of *fib* are tantalisingly close. We may, in fact, be fooled into thinking that we can obtain one from the other by fusing with $swap\ (a,b) = (b,a)$. A moment's reflection shows that this does not work since the 'next' function is the same in both cases. However, the two derivations imply a simple formula for a generalised version of *fib* that abstracts away from the two initial values.

$$\begin{aligned}
pfib\ i\ j = s\ \textbf{where}\ s &= i \prec s' \\
s' &= j \prec s + s'
\end{aligned}$$

We can express $pfib\ i\ j$ in terms of *fib* and fib', since

$$\begin{aligned}
& tail\ (pfib\ i\ j) \\
= {} & \{ \text{ tupling, analogous to the first derivation (A) } \} \\
& map\ snd\ (iterate\ step\ (i,j))
\end{aligned}$$

$$= \quad \{\ \mathit{map}\ g \cdot \mathit{iterate}\ f = \mathit{unfold}\ g\ f\ \}$$
$$\mathit{unfold}\ \mathit{snd}\ \mathit{step}\ (i, j)$$
$$= \quad \{\ \text{second derivation (B)}\ \}$$
$$i * \mathit{fib} + j * \mathit{fib}'$$

Furthermore, $\mathit{pfib}\ i\ j = (j - i) * \mathit{fib} + i * \mathit{fib}'$. So, in a sense, the extra generality of *pfib* is not needed.

As an example application, let us calculate the number of binary strings of a given length that do not contain adjacent 1s. These binary strings are used in the Fibonacci number system (Graham *et al.*, 1994). The empty string clearly satisfies the condition; a leading 0 can be followed by any string that satisfies the criterion; if the string starts with a 1, then it must be followed by a string that does not start with a 1. The description can be captured by the following grammar in Backus-Naur Form (*Nao* is short for n̲o a̲djacent o̲nes).

$$\mathit{Nao} = \epsilon \mid 0\ \mathit{Nao} \mid 1\ \mathit{Nap}$$
$$\mathit{Nap} = \epsilon \mid 0\ \mathit{Nao}$$

Here, *Nap* produces *Nao* strings that do not start with a 1. Now, to solve the original problem we have to group the strings by length. Let *nao* (*nap*) be the stream whose nth element specifies the number of words of length n produced by *Nao* (*Nap*). The language *Nao* contains one string of length 0 and $\mathit{nao} + \mathit{nap}$ strings of length $n + 1$, motivating the following stream definitions.

$$\mathit{nao} = 1 \prec \mathit{nao} + \mathit{nap}$$
$$\mathit{nap} = 1 \prec \mathit{nao}$$

By the token above, $\mathit{nao} = \mathit{tail}\ (\mathit{pfib}\ 1\ 1) = 1 * \mathit{fib} + 1 * \mathit{fib}' = \mathit{fib}''$. In Section 5.5 we will show how to systematise the translation of grammars into stream equations.

The elements of a stream are by no means confined to numbers. Using a lifted version of equality, $\doteqdot$, we can, for instance, generate a stream of Booleans. To illustrate the use of $\doteqdot$, let us prove that every third Fibonacci number is even: $(\mathit{fib}\ \mathbf{mod}\ 2 \doteqdot 0) = (\mathit{nat}\ \mathbf{mod}\ 3 \doteqdot 0)$. The proof makes essential use of the fact that equality on *Bool* is associative (Backhouse & Fokkinga, 2001). We show that *even fib* satisfies $x = \mathit{true} \prec \mathit{false} \prec \mathit{false} \prec x$, where *even* is lifted to streams—recall that, like integer constants, *false* and *true* are overloaded.

$$\mathit{even}\ \mathit{fib}$$
$$= \quad \{\ \text{definition of } \mathit{fib}, \text{ definition of } \mathit{fib}' \text{ and definition of } \mathit{even}\ \}$$
$$\mathit{true} \prec \mathit{false} \prec \mathit{even}\ (\mathit{fib} + \mathit{fib}')$$
$$= \quad \{\ \mathit{even}\ (a + b) == \mathit{even}\ a == \mathit{even}\ b, \text{ and lifting}\ \}$$
$$\mathit{true} \prec \mathit{false} \prec (\mathit{even}\ \mathit{fib} \doteqdot \mathit{even}\ \mathit{fib}')$$
$$= \quad \{\ \text{definition of } \mathit{fib} \text{ and definition of } \mathit{fib}'\ \}$$
$$\mathit{true} \prec \mathit{false} \prec ((\mathit{true} \prec \mathit{even}\ \mathit{fib}') \doteqdot (\mathit{false} \prec (\mathit{even}\ \mathit{fib} \doteqdot \mathit{even}\ \mathit{fib}')))$$
$$= \quad \{\ \text{definition of } \doteqdot\ \}$$

$$
\begin{aligned}
& \mathit{true} \prec \mathit{false} \prec \mathit{false} \prec (\mathit{even}\ \mathit{fib}' \mathrel{\dot{=}} \mathit{even}\ \mathit{fib} \mathrel{\dot{=}} \mathit{even}\ \mathit{fib}') \\
= \quad & \{\ a == a == \mathit{true} \text{ (twice), and lifting } \} \\
& \mathit{true} \prec \mathit{false} \prec \mathit{false} \prec \mathit{even}\ \mathit{fib}
\end{aligned}
$$

3.2.2 *Bit fiddling*

Bitte ein Bit
Slogan of "Bitburger Brauerei" (1951)

Let us move on to some less known examples, which are meant to illustrate the use of interleaving. All of the examples involve fiddling with bits, so we start off defining the stream of all bit strings.

$$\mathit{bit} = [\,] \prec 0 +\!\!+ \mathit{bit} \curlyvee 1 +\!\!+ \mathit{bit}$$

Here list concatenation $+\!\!+$ is lifted to streams. Furthermore, the literals 0 and 1 are doubly lifted, first to lists and then to streams: n expands to the stream of singleton lists $\mathit{pure}\ [n]$. The stream of positive numbers in binary representation, least significant bit first, is then $\mathit{bit} +\!\!+ 1$, and the stream of binary naturals is $[\,] \prec \mathit{bit} +\!\!+ 1$.

$\gg\ \mathit{bit}$
$\langle [\,], [0], [1], [0,0], [1,0], [0,1], [1,1], [0,0,0], [1,0,0], [0,1,0], [1,1,0], [0,0,1], \ldots \rangle$
$\gg\ [\,] \prec \mathit{bit} +\!\!+ 1$
$\langle [\,], [1], [0,1], [1,1], [0,0,1], [1,0,1], [0,1,1], [1,1,1], [0,0,0,1], [1,0,0,1], \ldots \rangle$

The function $\mathit{lookup}\ ([\,] \prec \mathit{bit} +\!\!+ 1)$ maps a natural number to its binary representation. Its inverse is the evaluation function $\mathit{val} = \mathit{foldr}\ (\lambda b\ n \to b + 2 * n)\ 0$.

It will be useful to define a variant of bit where all the bits have been zeroed out.

$$\mathit{null} = [\,] \prec 0 +\!\!+ \mathit{null} \curlyvee 0 +\!\!+ \mathit{null}$$

Now, how can we characterise the *pots*, the positive numbers that are powers of two ($A036987$)? A simple specification is

$$\mathit{pot} = (\mathit{bit} +\!\!+ 1 \mathrel{\dot{=}} \mathit{null} +\!\!+ 1).$$

All the bits of a pot are zero, except for the most significant bit. The specification above is equivalent to $\mathit{pot} = (\mathit{bit} \mathrel{\dot{=}} \mathit{null})$—recall that identities between elements, such as $x +\!\!+ z == y +\!\!+ z \iff x == y$, can be lifted to identities between streams. From the simplified specification we can easily calculate a definition of pot.

$$
\begin{aligned}
& \mathit{bit} \mathrel{\dot{=}} \mathit{null} \\
= \quad & \{\ \text{definition of } \mathit{bit} \text{ and definition of } \mathit{null}\ \} \\
& ([\,] \prec 0 +\!\!+ \mathit{bit} \curlyvee 1 +\!\!+ \mathit{bit}) \mathrel{\dot{=}} ([\,] \prec 0 +\!\!+ \mathit{null} \curlyvee 0 +\!\!+ \mathit{null}) \\
= \quad & \{\ \text{definition of } \dot{=} \text{ and abide law (11)}\ \} \\
& ([\,] == [\,]) \prec (0 +\!\!+ \mathit{bit} \mathrel{\dot{=}} 0 +\!\!+ \mathit{null}) \curlyvee (1 +\!\!+ \mathit{bit} \mathrel{\dot{=}} 0 +\!\!+ \mathit{null}) \\
= \quad & \{\ \text{equality}\ \} \\
& \mathit{true} \prec (\mathit{bit} \mathrel{\dot{=}} \mathit{null}) \curlyvee \mathit{false}
\end{aligned}
$$

Joining the first with the last line we obtain a definition for *pot*.

$$pot = true \prec pot \curlyvee false$$

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

$$msb = val\ (null + 1).$$

A similar derivation to the one above gives

$$msb = 1 \prec 2 * msb \curlyvee 2 * msb.$$

Put differently, *msb* is the largest pot at most nat'. Here and in what follows we also lift relations, 'x and y are related by R', to streams. So, instead of '$n + 1$ is odd iff n is even', where n is implicitly universally quantified, we say '$nat + 1$ is odd iff nat is even', where the relations are implicitly lifted to streams.

We can use *msb* to bit-reverse a positive number: all bits except the most significant one are reversed. The stream *brp* is specified ($A059893$)

$$brp \quad = \quad val\ (reverse\ bit + 1)$$

where *val* and *reverse* are both lifted to streams. The derivation makes use of a simple property of *val*: if the bit strings x and y have the same length, $length\ x = length\ y$, then

$$val\ (x + z) - val\ x \quad = \quad val\ (y + z) - val\ y.$$

Since $val\ (null + 1) - val\ null = msb$, this implies the property

$$val\ (reverse\ bit + 1) - val\ (reverse\ bit) \quad = \quad msb \tag{17}$$

between streams. The derivation proceeds as follows

$$\begin{aligned}
& val\ (reverse\ bit + 1) \\
= \quad & \{\ (17)\ \} \\
& val\ (reverse\ bit) + msb \\
= \quad & \{\ \text{definition of } bit, \text{ definition of } reverse, \text{ and definition of } val\ \} \\
& (0 \prec val\ (reverse\ bit) \curlyvee val\ (reverse\ bit + 1)) + msb \\
= \quad & \{\ \text{definition of } msb \text{ and abide law (11)}\ \} \\
& 1 \prec val\ (reverse\ bit) + 2 * msb \curlyvee val\ (reverse\ bit + 1) + 2 * msb \\
= \quad & \{\ (17)\ \} \\
& 1 \prec val\ (reverse\ bit + 1) + msb \curlyvee val\ (reverse\ bit + 1) + 2 * msb.
\end{aligned}$$

Consequently, *brp* is defined

$$brp = 1 \prec brp + msb \curlyvee brp + 2 * msb.$$

Another example along these lines is the 1s-counting sequence ($A000120$), also known as the *binary weight*. The binary representation of the even number $2 * nat$ has the same number of 1s as *nat*; the odd number $2 * nat + 1$ has one 1 more.

Hence, the sequence satisfies $ones = ones \curlyvee ones + 1$. Adding two initial values, we can turn the property into a definition.

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

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

Alternatively, $ones'$ can be derived from the specification $ones' = map\ sum\ (bit +\!\!\!+ 1)$, where sum computes the sum of a list of numbers. The details are left to the reader.

Let us inspect the sequences.

$$\begin{aligned} &\gg\ msb \\ &\langle 1, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, 16, \ldots \rangle \\ &\gg\ nat' - msb \\ &\langle 0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, \ldots \rangle \\ &\gg\ brp \\ &\langle 1, 2, 3, 4, 6, 5, 7, 8, 12, 10, 14, 9, 13, 11, 15, 16, \ldots \rangle \\ &\gg\ ones \\ &\langle 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, \ldots \rangle \end{aligned}$$

The sequence $nat' - msb$ (A053645) exhibits a nice pattern; it describes the distance to the largest pot at most nat'.

3.2.3 Binary carry sequence

Here is a sequence that every computer scientist should know: the *binary carry sequence* or *ruler function* (A007814). The sequence gives the exponent of the largest pot dividing nat', that is, the number of leading zeros in the binary representation (least significant bit first). Put differently, it quantifies the running time of the binary increment. To specify the sequence we make use of the divisibility relation on numbers: $a \setminus b \iff \exists n \in \mathbb{N}\ .\ a * n = b$. The stream $carry$ is then specified[6]

$$\forall s\ . \quad carry \mathrel{\dot{\geqslant}} s \quad = \quad 2^s \setminus nat'. \tag{18}$$

Note that the variable s ranges over integer streams. Ordering and divisibility are lifted pointwise to streams—recall that the lifted variant of $\geqslant$ is denoted $\dot{\geqslant}$. The specification precisely captures the verbal description: read from left to right it implies $2^{carry} \setminus nat'$; read from right to left it formalises that $carry$ is the largest such sequence. Now we calculate

$$\begin{aligned} &\quad 2^s \setminus nat' \\ &= \quad \{\ nat' = 2 * nat + 1 \curlyvee 2 * nat' \text{ and } s = s_1 \curlyvee s_2\ \} \\ &\quad 2^{s_1 \curlyvee s_2} \setminus (2 * nat + 1 \curlyvee 2 * nat') \\ &= \quad \{\ (10) \text{ and abide law } (11)\ \} \end{aligned}$$

[6] The author is grateful to Roland Backhouse for suggesting this specification.

$$
\begin{aligned}
&(2^{s_1} \setminus (2 * nat + 1)) \curlyvee (2^{s_2} \setminus (2 * nat')) \\
=\ & \{\ 2^a \setminus (2 * b + 1) \Longleftrightarrow 0 \geqslant a, \text{ and lifting }\} \\
&(0 \mathrel{\dot{\geqslant}} s_1) \curlyvee (2^{s_2} \setminus (2 * nat')) \\
=\ & \{\ 2^a \setminus (2 * b) \Longleftrightarrow 2^{a-1} \setminus b, \text{ and lifting }\} \\
&(0 \mathrel{\dot{\geqslant}} s_1) \curlyvee (2^{s_2 - 1} \setminus nat') \\
=\ & \{\ \text{specification (18)}\ \} \\
&(0 \mathrel{\dot{\geqslant}} s_1) \curlyvee (carry \mathrel{\dot{\geqslant}} s_2 - 1) \\
=\ & \{\ \text{arithmetic}\ \} \\
&(0 \mathrel{\dot{\geqslant}} s_1) \curlyvee (carry + 1 \mathrel{\dot{\geqslant}} s_2) \\
=\ & \{\ \text{abide law (11) and } s = s_1 \curlyvee s_2\ \} \\
&(0 \curlyvee carry + 1) \mathrel{\dot{\geqslant}} s.
\end{aligned}
$$

Since (18) uniquely determines *carry*, we have

$$carry = 0 \curlyvee carry + 1,$$

which is, in fact, an executable program. (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: $carry = 0 \prec carry + 1 \curlyvee 0$. The unfolding works as long as the first argument of $\curlyvee$ is a sequence defined elsewhere.)

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

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

The call *turn n* yields the heights of the nodes of a perfect binary tree of depth *n*. Stewart refers to this recursion pattern as the 'army method' with an $(n+1)$-star general delegating work to two *n*-star generals (1999). 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 infinite tree is a so-called *sideways tree*: it has no root, but extends infinitely upwards. The following parametrised stream captures the traversal:

$$tree\ n = n \prec turn\ n \lll tree\ (n+1),$$

where $\lll$ prepends a list to a sequence.

$$
\begin{aligned}
&(\lll) &&:: [\alpha] \to Stream\ \alpha \to Stream\ \alpha \\
&as \lll s &&= foldr\ (\prec)\ s\ as
\end{aligned}
$$

Here is the punch line: *tree* 0 also yields the binary carry sequence! The following table of binary numbers illustrates the correspondence.

															1
							1	1	1	1	1	1	1	1	0
			1	1	1	1	0	0	0	0	1	1	1	1	0
	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

For emphasis, prefixes of zeros are underlined. The lines correspond to the marks on a binary ruler; this is why *carry* is also called the ruler function. If we connect each 0-prefix of length n with the nearest 0-prefix of length $n+1$, we obtain the sideways tree.

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

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

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

$$(x \lll s) \curlyvee (y \lll t) \quad = \quad (x \curlyvee y) \lll (s \curlyvee t), \tag{19}$$

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

$$\begin{aligned} & \mathit{tree}\ (k+1) \\ = \ & \{ \text{definition of } \mathit{tree} \} \\ & k+1 \prec \mathit{turn}\ (k+1) \lll \mathit{tree}\ (k+2) \\ \subset \ & \{ x\ n = n+1 \prec \mathit{turn}\ (n+1) \lll x\ (n+1) \text{ has a unique solution} \} \\ & k+1 \prec \mathit{turn}\ (k+1) \lll (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\ = \ & \{ \text{proof obligation, see below} \} \\ & k+1 \prec (\mathit{replicate}\ 2^k\ 0 \curlyvee \mathit{turn}\ k + 1) \lll (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\ = \ & \{ \text{definition of } \curlyvee \text{ and definition of } \lll \} \\ & ((k+1 : \mathit{turn}\ k + 1) \curlyvee \mathit{replicate}\ 2^k\ 0) \lll (\mathit{tree}\ (k+1) + 1 \curlyvee 0) \\ = \ & \{ \text{mixed abide law (19)} \} \\ & ((k+1 : \mathit{turn}\ k + 1) \lll \mathit{tree}\ (k+1) + 1) \curlyvee (\mathit{replicate}\ 2^k\ 0 \lll 0) \\ = \ & \{ \text{arithmetic and definition of } \lll \} \\ & (k \prec \mathit{turn}\ k \lll \mathit{tree}\ (k+1)) + 1 \curlyvee 0 \\ = \ & \{ \text{definition of } \mathit{tree} \} \\ & \mathit{tree}\ k + 1 \curlyvee 0 \end{aligned}$$

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

3.2.4 Fractal sequences

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

$$frac = nat \curlyvee frac$$

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:

$$god = 2 * frac + 1$$

is the *greatest odd divisor* of nat'. Since we have $god = 2 * nat + 1 \curlyvee god$ and $nat' = 2 * nat + 1 \curlyvee 2 * nat'$, the property follows from a simple case analysis: the greatest odd divisor of an odd number, $2 * nat + 1$, is the number itself; the greatest odd divisor of an even number, $2 * nat'$, is the god of nat'.

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

$$2^{carry} * god \quad = \quad nat'.$$

The proof is surprisingly straightforward.

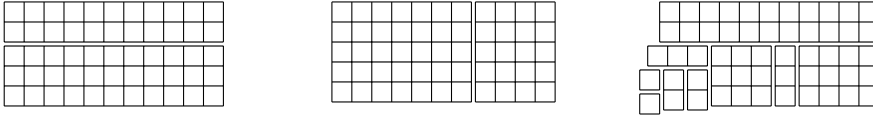
$$\begin{aligned}
& 2^{carry} * god \\
= \quad & \{ \text{ definition of } carry \text{ and definition of } god \ \} \\
& 2^{0 \curlyvee carry+1} * (2 * nat + 1 \curlyvee god) \\
= \quad & \{ \text{ arithmetic and abide law (11) } \} \\
& 2 * nat + 1 \curlyvee 2 * 2^{carry} * god \\
\subset \quad & \{ \ x = 2 * nat + 1 \curlyvee 2 * x \text{ has a unique solution } \} \\
& 2 * nat + 1 \curlyvee 2 * nat' \\
= \quad & \{ \ nat' = 2 * nat + 1 \curlyvee 2 * nat' \ \} \\
& nat'
\end{aligned}$$

3.2.5 Sprague-Grundy numbers

Quadratisch. Praktisch. Gut.

Slogan of "Ritter Sport" (1970)

There is an intriguing connection between the sequence *frac* and the *chocolate game*. Two players participate in this game, eating up a chocolate bar according the following rule: the player whose turn it is breaks the bar in two pieces, either horizontally or vertically, and eats the smaller one—or the more appealing one in case of a tie.



The players make alternating moves; the game ends when the bar consists of a single square. The player whose turn it is loses and consequently has to pay for the chocolate bar.

Representing a game position by the two dimensions of the bar, a possible sequence of moves is, see picture on the right above:

$$(11,5) \mapsto (11,3) \mapsto (7,3) \mapsto (6,3) \mapsto (3,3) \mapsto (3,2) \mapsto (2,2) \mapsto (1,2) \mapsto (1,1).$$

Since the number of moves is even, the first player loses. In fact, the first position, $(11,5)$, is a *losing position*: every move leads to a winning position. Conversely, from a winning position there is at least one move to a losing position.

How can we determine whether a position is a winning or a losing position? This is where the sequence *frac* enters the scene: (i,j) is a winning position iff $(\mathit{frac})_i = (\mathit{frac})_j$! By this token, square bars—the famous Ritter Sport—are losing positions, but also $(7,3)$ and $(11,5)$ as $(\mathit{frac})_7 = 0 = (\mathit{frac})_3$ and $(\mathit{frac})_{11} = 1 = (\mathit{frac})_5$.

The chocolate game is an example of an *impartial two-person game*. For reasons of space, we can only sketch the necessary theory. For a more leisurely exposition we refer the interested reader to the treatment by Backhouse & Michaelis (2005). An impartial game is fully determined by a function $\mathit{move} :: \mathit{Pos} \to [\mathit{Pos}]$, where *Pos* is the type of game positions. The chocolate game can be viewed as a combination of two identical games, breaking a $1 \times n$ bar, whose individual *move* function is

$$\mathit{move}\ n = \mathit{take}\ (n\ \mathbf{div}\ 2)\ [n-1, n-2\,..\,1]$$

A position in the combined game is then a pair of positions. The combined game is known as a *sum game*, because a player can either make a move in the left or in the right component game. The central idea for determining winning and losing positions in a sum game is to assign a number to each component position, the so-called *Sprague-Grundy number*, such that (i,j) is a losing position iff $\mathit{sg}\ i = \mathit{sg}\ j$. The numbers are chosen so that every move from a losing position makes the numbers unequal, and for every winning position there is a move that makes them equal. One can show that the following definition of *sg* satisfies these requirements.

$$\begin{aligned} \mathit{sg}\ p &= \mathit{mex}\ \{\mathit{sg}\ q \mid q \leftarrow \mathit{move}\ p\} \\ \mathit{mex}\ x &= \mathit{head}\ \langle n \mid n \leftarrow \mathit{nat}, n \notin x\rangle \end{aligned}$$

Here, $\mathit{mex}\ x$ yields the *minimal excludent* of x, the least natural number that is not contained in the finite set x. The Sprague-Grundy number of p is then the minimal excludent of the Sprague-Grundy numbers of positions reachable from p.

The claim is that frac' is the tabulation of *sg*:

$$\mathit{map}\ \mathit{sg}\ \mathit{nat}' = \mathit{frac}'$$

(The positions only comprise the positive numbers; hence we have to take the tail of the two sequences.) This equation is not at all obvious, and, indeed, it takes some

effort to prove it correct. The proof is, however, quite instructive, as it shows how to deal with course-of-value recursion and interleavings.

First observe that the executable specification of sg is not good for a program, as it is horribly inefficient: $sg\ p$ spawns $p\ \mathbf{div}\ 2$ recursive calls, each of which triggers another avalanche of calls. For instance, $sg\ 25$ generates a total of 8,986,539 calls. The call structure of sg is an instance of *course-of-value recursion*. In this recursion scheme, the value of g_n is defined in terms of all previous values: g_{n-1}, g_{n-2}, ..., g_0. For tabulating functions defined by course-of-value recursion, it is useful to introduce a stream operator that maps a stream to the stream of its partial reverses:

$$\begin{array}{l}course \quad :: Stream\ \alpha \rightarrow Stream\ [\alpha] \\ course\ s = t\ \mathbf{where}\ t = [\,] \prec s : t\end{array}$$

where : is lifted to streams. In what follows, we shall also lift the list processing functions $take$, $init$, $last$, set (which takes a list to a set), and the set operations $\{-\}$ (singleton set), $\cup$ (set union), and $\setminus$ (set difference) to streams.

The functioning of $course$ is illustrated below.

$$\begin{array}{cccccccccccccccc}
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} & \cdots & & t & & \\
\parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \cdots & & \parallel & & \\
[\,] & s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & \cdots & & [\,] & \prec & s \\
 & : & : & : & : & : & : & : & : & : & : & \cdots & & & & : \\
 & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & \cdots & & & & t
\end{array}$$

If f is the function that takes the list of previous values $[g_{n-1}, g_{n-2}, \ldots, g_0]$ to g_n, then $tabulate\ g$ is the unique solution of

$$s = map\ f\ (course\ s)$$

(The equation has a unique solution, as a simple unfolding of $course$ and map turns it into the required syntactic form.) Again, we avoid index manipulations by introducing a suitable stream operator. In our example, f is essentially mex after $take$. Some straightforward calculations turn the recursive definition of sg into a recursion equation over streams: sg's tabulation, $map\ sg\ nat'$, satisfies

$$x = map\ mex\ (set\ (take\ (nat'\ \mathbf{div}\ 2)\ (course\ x)))$$

Since the equation has a unique solution, we are left with the task of showing that $frac'$ also satisfies the equation, which is what we tackle next.

In a succession of calculations we determine $r = course\ frac'$, $s = take\ (nat'\ \mathbf{div} 2)\ r$, $t = set\ s$, and finally $map\ mex\ t$. Observe that both $nat'\ \mathbf{div}\ 2$ and $frac'$ involve interleavings: $nat'\ \mathbf{div}\ 2 = nat \curlyvee nat'$ and $frac' = frac \curlyvee nat'$. To deal effectively with interleavings, it is helpful to recall that the equation $x \curlyvee y = a \prec f\ x \curlyvee g\ y$ uniquely determines x and y, see Section 2.3. With a single equation we capture three sequences: the entire stream, the stream of elements at even positions and the stream of elements at odd positions.

Now, setting $re \curlyvee ro = course\ frac'$, we reason

$$
\begin{aligned}
& re \curlyvee ro \\
= \ & \{ \text{ definition of } re \curlyvee ro \ \} \\
& course\ frac' \\
= \ & \{ \text{ definition of } course \ \} \\
& [\,] \prec frac' : course\ frac' \\
= \ & \{ \text{ definition of } frac' \text{ and definition of } re \curlyvee ro \ \} \\
& [\,] \prec (frac \curlyvee nat') : (re \curlyvee ro) \\
= \ & \{ \text{ abide law (11) } \} \\
& [\,] \prec frac : re \curlyvee nat' : ro.
\end{aligned}
$$

Next, let $se \curlyvee so = take\ (nat \curlyvee nat')\ (re \curlyvee ro)$. Using an analogous calculation we can show that $se \curlyvee so = [\,] \prec frac : se \curlyvee nat' : init\ so$. Finally, let $te \curlyvee to = set\ (se \curlyvee so)$, then

$$
\begin{aligned}
& te \curlyvee to \\
= \ & \{ \text{ definition of } te \curlyvee to \ \} \\
& set\ (se \curlyvee so) \\
= \ & \{ \text{ see above } \} \\
& set\ ([\,] \prec frac : se \curlyvee nat' : init\ so) \\
= \ & \{ \text{ definition of } set\text{: } set\ [\,] = \{\} \text{ and } set\ (a : x) = \{a\} \cup set\ x \ \} \\
& \{\} \prec \{frac\} \cup set\ se \curlyvee \{nat'\} \cup set\ (init\ so)
\end{aligned}
$$

We are stuck. We have to express $set\ (init\ so)$ in terms of $set\ so$. We can rewrite the former stream expression to $set\ so \setminus \{last\ so\}$, where $\setminus$ denotes set difference. However, to make further progress we need to know $last\ so$. So let us peek at some values.

$$
\begin{aligned}
& \gg\ last\ so \\
& \langle 0, 1, 0, 2, 1, 3, 0, 4, 2, 5, 1, 6, 3, 7, 0, 8, \ldots \rangle \\
& \gg\ frac' \\
& \langle 0, 1, 0, 2, 1, 3, 0, 4, 2, 5, 1, 6, 3, 7, 0, 8, \ldots \rangle
\end{aligned}
$$

Somewhat surprisingly, $last\ so$ seems to be $frac'$! We leave this conjecture as a proof obligation and finish off ($\setminus$ binds more tightly than $\cup$).

$$
\begin{aligned}
= \ & \{ \text{ proof obligation: } last\ so = frac' \ \} \\
& \{\} \prec \{frac\} \cup set\ se \curlyvee \{nat'\} \cup set\ so \setminus \{frac'\} \\
= \ & \{ \text{ definition of } te \curlyvee to \text{ and abide law (11) } \} \\
& \{\} \prec \{frac\} \cup te \curlyvee \{nat'\} \cup to \setminus \{frac'\}
\end{aligned}
$$

We are nearly done. It is not hard to see that $te \curlyvee to = \{\} \prec \{frac\} \cup te \curlyvee \{nat'\} \cup to \setminus \{frac'\}$ has the unique solution $\{0 \mathinner{..} nat\} \setminus \{frac\} \curlyvee \{0 \mathinner{..} nat\}$. (Here we

also lift the mixfix notation $\{a\,..\,b\}$ to streams.) Consequently,

$$
\begin{aligned}
& \mathit{map}\ \mathit{mex}\ (\mathit{te} \curlyvee \mathit{to}) \\
= \quad & \{ \text{ see above } \} \\
& \mathit{map}\ \mathit{mex}\ (\{0\,..\,\mathit{nat}\} \setminus \{\mathit{frac}\} \curlyvee \{0\,..\,\mathit{nat}\}) \\
= \quad & \{ \text{ abide law (11) } \} \\
& \mathit{map}\ \mathit{mex}\ (\{0\,..\,\mathit{nat}\} \setminus \{\mathit{frac}\}) \curlyvee \mathit{map}\ \mathit{mex}\ \{0\,..\,\mathit{nat}\} \\
= \quad & \{ \text{ definition of } \mathit{mex} \text{ and } (\mathit{frac} \mathrel{\dot{<}} \mathit{nat}') = \mathit{true} \} \\
& \mathit{frac} \curlyvee \mathit{nat}' \\
= \quad & \{ \text{ definition of } \mathit{frac}' \} \\
& \mathit{frac}'
\end{aligned}
$$

Hurray! The second but last step depends on the crucial property that *frac* is smaller than nat'—this follows from the fact that $\mathit{god} = 2 * \mathit{frac} + 1$ is the greatest odd divisor of nat'.

We still have to discharge the proof obligation. Since the stream *so* is given by the equation $\mathit{so} = [0] \prec \mathit{frac}' : \mathit{nat}' : \mathit{init}\ \mathit{so}$, we have to show that $\mathit{last}\ \mathit{so} = \mathit{frac}'$. This is, in fact, an instance of a more general property: if x is given by $x = [\mathit{head}\ s] \prec \mathit{tail}\ s : t : \mathit{init}\ x$, then $\mathit{last}\ x = s \curlyvee t$. The mostly straightforward proof is left as an exercise to the reader.

3.2.6 Josephus problem

Our final example is a variant of the *Josephus problem* (Graham *et al.*, 1994). 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 $\not 2$ 3 $\not 4$ 5 $\not 6$. Renumbering 1 3 5 to 1 2 3, we observe that if *nat* survives in the sequence of first-round survivors, then $2 * \mathit{nat} - 1$ survives in the original sequence. Likewise for odd numbers: 1 2 3 4 5 6 7 becomes $\not 1$ $\not 2$ 3 $\not 4$ 5 $\not 6$ 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 *nat* survives in the remaining sequence, then $2 * \mathit{nat} + 1$ survives in the original sequence. The verbal description is captured by the stream equation below.

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

The nth element of *jos* determines the survivor's number, when there are n persons initially. It is quite revealing to inspect the sequence.

$$
\begin{aligned}
& \gg\ \mathit{jos} \\
& \langle 1, 1, 3, 1, 3, 5, 7, 1, 3, 5, 7, 9, 11, 13, 15, 1, \ldots \rangle \\
& \gg\ (\mathit{jos} - 1) \,/\, 2 \\
& \langle 0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, \ldots \rangle
\end{aligned}
$$

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: $nat' - msb$. Indeed, *jos* and *msb* are related by

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

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.

$$\begin{aligned}
& 2 * (nat' - msb) + 1 \\
= & \quad \{ \text{ definition of } msb \text{ and property of } nat' \ \} \\
& 2 * ((1 \prec 2 * nat' \curlyvee 2 * nat' + 1) - (1 \prec 2 * msb \curlyvee 2 * msb)) + 1 \\
= & \quad \{ \text{ abide law (11) } \} \\
& 2 * (0 \prec 2 * nat' - 2 * msb \curlyvee 2 * nat' + 1 - 2 * msb) + 1 \\
= & \quad \{ \text{ arithmetic } \} \\
& 1 \prec 2 * (2 * (nat' - msb) + 1) - 1 \curlyvee 2 * (2 * (nat' - msb) + 1) + 1
\end{aligned}$$

4 Finite calculus (Δ, Σ)

Let us move on to another application of streams: *finite calculus* (Graham *et al.*, 1994). 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.

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

Here are some examples (*A*003215, *A*000079, *A*094267, not listed).

$\gg \ \Delta\ nat^3$
$\langle 1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, \ldots \rangle$
$\gg \ \Delta\ 2^{nat}$
$\langle 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, \ldots \rangle$
$\gg \ \Delta\ carry$
$\langle 1, -1, 2, -2, 1, -1, 3, -3, 1, -1, 2, -2, 1, -1, 4, -4, \ldots \rangle$
$\gg \ \Delta\ jos$
$\langle 0, 2, -2, 2, 2, 2, -6, 2, 2, 2, 2, 2, 2, 2, -14, 2, \ldots \rangle$

The finite difference of *carry* exhibits an interesting pattern, so let us calculate the stream as a warm-up exercise.

$$\begin{aligned} & \Delta\, carry \\ = \;& \{ \text{ definition of } \Delta \text{ } \} \\ & tail\ carry - carry \\ = \;& \{ \text{ definition of carry } \} \\ & (carry + 1 \curlyvee 0) - (0 \curlyvee carry + 1) \\ = \;& \{ \text{ abide law (11) } \} \\ & (carry + 1) \curlyvee -(carry + 1) \end{aligned}$$

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

$$\begin{aligned} & \Delta\, (nat^{\underline{n+1}}) \\ = \;& \{ \text{ definition of } \Delta \text{ } \} \\ & tail\ (nat^{\underline{n+1}}) - nat^{\underline{n+1}} \\ = \;& \{ \text{ definition of } nat \text{ } \} \\ & (nat + 1)^{\underline{n+1}} - nat^{\underline{n+1}} \end{aligned}$$

Starting at the other end, we obtain

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

We can connect the loose ends, if the new power satisfies both $x^{\underline{n+1}} = x * (x - 1)^{\underline{n}}$ and $x^{\underline{n+1}} = x^{\underline{n}} * (x - n)$. That is 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* (Graham *et al.*, 1994). The following session shows that they behave as expected.

$\gg\ nat^{\underline{3}}$
$\langle 0, 0, 0, 6, 24, 60, 120, 210, 336, 504, 720, 990, 1320, 1716, 2184, 2730, \ldots \rangle$
$\gg\ \Delta\ (nat^{\underline{3}})$
$\langle 0, 0, 6, 18, 36, 60, 90, 126, 168, 216, 270, 330, 396, 468, 546, 630, \ldots \rangle$
$\gg\ 3 * nat^{\underline{2}}$
$\langle 0, 0, 6, 18, 36, 60, 90, 126, 168, 216, 270, 330, 396, 468, 546, 630, \ldots \rangle$

Table 2. *Converting between powers and falling factorial powers*

$x^0 = x^{\underline{0}}$	$x^{\underline{0}} = x^0$
$x^1 = x^{\underline{1}}$	$x^{\underline{1}} = x^1$
$x^2 = x^{\underline{2}} + x^{\underline{1}}$	$x^{\underline{2}} = x^2 - x^1$
$x^3 = x^{\underline{3}} + 3 * x^{\underline{2}} + x^{\underline{1}}$	$x^{\underline{3}} = x^3 - 3 * x^2 + 2 * x^1$
$x^4 = x^{\underline{4}} + 6 * x^{\underline{3}} + 7 * x^{\underline{2}} + x^{\underline{1}}$	$x^{\underline{4}} = x^4 - 6 * x^3 + 11 * x^2 - 6 * x^1$

Table 3. *Laws for finite difference (c, k and n are constant streams)*

$\Delta\ (tail\ s) = tail\ (\Delta\ s)$	$\Delta\ (s + t) = \Delta\ s + \Delta\ t$
$\Delta\ (a \prec s) = head\ s - a \prec \Delta\ s$	$\Delta\ (s * t) = s * \Delta\ t + \Delta\ s * tail\ t$
$\Delta\ (s \curlyvee t) = (t - s) \curlyvee (tail\ s - t)$	$\Delta\ c^{nat} = (c - 1) * c^{nat}$
$\Delta\ n = 0$	$\Delta\ (nat^{\underline{n+1}}) = (n + 1) * nat^{\underline{n}}$
$\Delta\ (n * s) = n * \Delta\ s$	$\Delta \binom{nat}{k+1} = \binom{nat}{k}$

One can convert mechanically between powers and falling factorial powers using Stirling numbers (Graham *et al.*, 1994). The details are beyond the scope of this paper. For reference, Table 2 displays the correspondence up to the fourth power.

4.1.1 Laws

Table 3 lists the rules for finite differences. First of all, Δ is a *linear operator*: it distributes over sums. The stream 2^{nat} is the discrete analogue of e^x as $\Delta\ 2^{nat} = 2^{nat}$. In general,

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

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

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

We can avoid the unpleasant asymmetry of the law, if we further massage $\Delta\, s * tail\ t$.

$$
\begin{aligned}
&= \quad \{ \text{ definition of } \Delta \ \} \\
&\quad\ s * \Delta\, t + \Delta\, s * (t + \Delta\, t) \\
&= \quad \{ \text{ arithmetic } \} \\
&\quad\ s * \Delta\, t + \Delta\, s * t + \Delta\, s * \Delta\, t
\end{aligned}
$$

The rule for binomial coefficients is a special case of the rule for falling factorial powers as $\binom{n}{k} = n^{\underline{k}} \,/\, k!$.

4.1.2 Example: Josephus problem, continued

Let us get back to the Josephus problem: the interactive session in Section 4.1 suggests that $\Delta\, jos$ is almost always 2, except for pots, the powers of two. We can express this property using a *stream conditional*:

$$\Delta\, jos = (pot' \rightarrow -nat \ ; 2)$$

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

$$((s_1 \curlyvee s_2) \rightarrow (t_1 \curlyvee t_2) \,;\, (u_1 \curlyvee u_2)) \quad = \quad (s_1 \rightarrow t_1 \,;\, u_1) \curlyvee (s_2 \rightarrow t_2 \,;\, u_2) \qquad (20)$$

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

$$
\begin{aligned}
&\quad\ \Delta\, jos \\
&= \quad \{ \ \Delta \text{ law and arithmetic } \} \\
&\quad\ 0 \prec\ 2 \curlyvee 2 * (tail\ jos - jos) - 2 \\
&= \quad \{ \text{ definition of } \Delta \ \} \\
&\quad\ 0 \prec\ 2 \curlyvee 2 * \Delta\, jos - 2 \\
&\subset \quad \{ \ x = 0 \prec\ 2 \curlyvee 2 * x - 2 \text{ has a unique solution } \} \\
&\quad\ 0 \prec\ 2 \curlyvee 2 * (pot' \rightarrow -nat \ ; 2) - 2 \\
&= \quad \{ \text{ arithmetic and definition of } nat' \ \} \\
&\quad\ 0 \prec\ 2 \curlyvee (pot' \rightarrow -(2 * nat') \,;\, 2) \\
&= \quad \{ \text{ definition of } nat, \text{ definition of } pot \text{ and definition of } \curlyvee \ \} \\
&\quad\ (pot \rightarrow -(2 * nat) \,;\, 2) \curlyvee 2 \\
&= \quad \{ \text{ conditional and abide law (20) } \} \\
&\quad\ ((pot \curlyvee false) \rightarrow (-(2 * nat \curlyvee 2 * nat + 1)) \,;\, (2 \curlyvee 2)) \\
&= \quad \{ \text{ definition of } pot' \text{ and characterisation of } nat \ \} \\
&\quad\ (pot' \rightarrow -nat \ ; 2)
\end{aligned}
$$

The formula suggests a simple strategy to survive a shootout of increasing size: If the circle is extended by one person, move two positions clockwise, unless the total number is a power of two, in which case move to the first position.

4.1.3 Higher-order differences

An interesting pattern emerges, when we repeatedly apply the difference operator to a stream. A few quick calculations give the following formulas for higher-order differences (recall that $f^{(n)}$ is n-fold composition).

$$\begin{aligned}
\Delta^{(0)}\ s &= tail^{(0)}\ s \\
\Delta^{(1)}\ s &= tail^{(1)}\ s - tail^{(0)}\ s \\
\Delta^{(2)}\ s &= tail^{(2)}\ s - 2 * tail^{(1)}\ s + tail^{(0)}\ s \\
\Delta^{(3)}\ s &= tail^{(3)}\ s - 3 * tail^{(2)}\ s + 3 * tail^{(1)}\ s - tail^{(0)}\ s
\end{aligned}$$

The factors of the tails appear to be binomial coefficients with alternating signs. To reveal the connection, let us rewrite Δ in a point-free style.

$$\Delta = tail - id$$

Subtraction is now doubly lifted: first to functions and then to streams, both of which are idioms. Indeed, if we provide a *Num* instance for the environment idiom,[7] then the definition above is even legal Haskell!

Under this view the formulas for higher-order differences are simply instances of the Binomial Theorem lifted to stream operators.

$$(tail - id)^{(n)} \quad = \quad \sum_k repeat \binom{n}{k} * (tail^{(k)} \cdot (-id)^{(n-k)}) \tag{21}$$

(The unbounded sum on the right is really finite, as all terms are zero except those with $0 \leqslant k \leqslant n$.) The Binomial Theorem is applicable, because *tail* and $-id$ are linear operators and because they commute with each other. To see where these two conditions originate, let us prove the higher-order version of the Binomial Theorem for $n = 2$.

$$\begin{aligned}
& (f + g) \cdot (f + g) \\
= \quad & \{ \cdot \text{ right-distributes over } + \} \\
& f \cdot (f + g) + g \cdot (f + g) \\
= \quad & \{ f \text{ and } g \text{ are linear} \} \\
& f \cdot f + f \cdot g + g \cdot f + g \cdot g \\
= \quad & \{ f \cdot g = g \cdot f \} \\
& f \cdot f + 2 * (f \cdot g) + g \cdot g
\end{aligned}$$

4.2 Binomial transforms

Here is a little puzzle: continue the sequence $u = \langle 0, 1, 4, 12, 32, 80, 192, 448, \ldots \rangle$. Play a bit with the numbers, before you proceed.

[7] In principle, every idiom can be made an instance of *Num* (Section 2.1).

As noted before, a first routine step is to look at the higher-order differences. For instance, if $\Delta^{(n)}\ s$ is constant for some n, then the unknown stream s is a polynomial.

$\gg\ \Delta^{(0)}\ u$
$\langle 0, 1, 4, 12, 32, 80, 192, 448, 1024, 2304, 5120, 11264, 24576, 53248, 114688, 245760, \ldots\rangle$
$\gg\ \Delta^{(1)}\ u$
$\langle 1, 3, 8, 20, 48, 112, 256, 576, 1280, 2816, 6144, 13312, 28672, 61440, 131072, \ldots\rangle$
$\gg\ \Delta^{(2)}\ u$
$\langle 2, 5, 12, 28, 64, 144, 320, 704, 1536, 3328, 7168, 15360, 32768, 69632, 147456, \ldots\rangle$
$\gg\ \Delta^{(3)}\ u$
$\langle 3, 7, 16, 36, 80, 176, 384, 832, 1792, 3840, 8192, 17408, 36864, 77824, 163840, \ldots\rangle$

Unfortunately, the higher-order differences do not become smaller. However, their heads form a familiar sequence: $\mathscr{B}^{(-1)}\ u = nat$ where

$$\begin{aligned} &\mathscr{B}^{(-1)} :: (Num\ \alpha) \Rightarrow Stream\ \alpha \to Stream\ \alpha \\ &\mathscr{B}^{(-1)} = unfold\ head\ \Delta \end{aligned}$$

If we view the stream of higher-order differences *iterate* Δ as a matrix, then $\mathscr{B}^{(-1)}$ maps the first row to the first column. As a first step towards solving the puzzle, let us derive the inverse transformation, which maps the first column to the first row, characterised as follows:

$$\begin{aligned} head \cdot \mathscr{B} &= head \\ \Delta \cdot \mathscr{B} &= \mathscr{B} \cdot tail \end{aligned}$$

The first row and the first column share the left, upper corner; transforming the tail of the first column yields the second row, which is the difference of the first row. This data uniquely determines $\mathscr{B}$ as a quick calculation shows.

$$\begin{aligned} &\mathscr{B}\ (tail\ s) = \Delta\ (\mathscr{B}\ s) \\ \iff\ &\{\ \text{definition of } \Delta\ \} \\ &\mathscr{B}\ (tail\ s) = tail\ (\mathscr{B}\ s) - \mathscr{B}\ s \\ \iff\ &\{\ \text{arithmetic}\ \} \\ &tail\ (\mathscr{B}\ s) = \mathscr{B}\ s + \mathscr{B}\ (tail\ s) \end{aligned}$$

Factoring out $\mathscr{B}\ s$ for efficiency, we obtain the following definition of $\mathscr{B}$.

$$\begin{aligned} &\mathscr{B} :: (Num\ \alpha) \Rightarrow Stream\ \alpha \to Stream\ \alpha \\ &\mathscr{B}\ s = t\ \mathbf{where}\ t = head\ s \prec t + \mathscr{B}\ (tail\ s) \end{aligned}$$

The stream operator $\mathscr{B}$ is known as the *binomial transform*[8], as it sends $\langle s_0, s_1, s_2, s_3, \ldots\rangle$ to $\langle s_0, s_0 + s_1, s_0 + 2 * s_1 + s_2, s_0 + 3 * s_1 + 3 * s_2 + s_3, \cdots\rangle$. In general, we have

$$(\mathscr{B}^{(-1)}\ s)_n = \sum_k \binom{n}{k} * (s)_k * (-1)^{n-k} \quad \text{and} \quad (\mathscr{B}\ s)_n = \sum_k \binom{n}{k} * (s)_k \tag{22}$$

[8] There is another transformation that goes under the same name: Knuth (1998) defines a variant where the tail is the *difference* of $\mathscr{B}\ s$ and $\mathscr{B}\ (tail\ s)$, rather than the sum.

The first formula is simply the pointwise version of (21). The characterisation of $\mathscr{B}$ implies that geometric progressions have a particularly simple binomial transform:

$$\mathscr{B}\ c^{nat} \quad = \quad (c+1)^{nat}$$

which is, in fact, a special case of the Binomial Theorem. The property can also be shown using the fact that the binomial transform is a linear transformation. (The proofs are left as exercises to the reader.)

$$\mathscr{B}\ (n * s) = n * \mathscr{B}\ s$$
$$\mathscr{B}\ (s + t) = \mathscr{B}\ s + \mathscr{B}\ t$$

We are finally in a position to tackle the puzzle.

$$\begin{aligned}
& \mathscr{B}\ nat \\
= \quad & \{ \text{ definition of } \mathscr{B} \text{ and definition of } nat \ \} \\
& 0 \prec \ \mathscr{B}\ nat + \mathscr{B}\ (nat + 1) \\
= \quad & \{ \ \mathscr{B} \text{ is linear } \} \\
& 0 \prec \ 2 * \mathscr{B}\ nat + \mathscr{B}\ 1 \\
= \quad & \{ \ \mathscr{B}\ 1 = \mathscr{B}\ (1^{nat}) = 2^{nat} \ \} \\
& 0 \prec \ 2 * \mathscr{B}\ nat + 2^{nat}
\end{aligned}$$

We have to solve the recursion equation $x = 0 \prec \ 2 * x + 2^{nat}$. We shall later show how to do this systematically. For now, we just check that $nat * 2^{nat-1}$ is the solution of the equation and hence of the puzzle.

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

4.3 Newton series

The binomial transform $\mathscr{B}^{(-1)}$ is essentially the finite difference applied repeatedly. What happens if we in turn apply the binomial transform repeatedly? This sounds a bit scary, but it turns out that the operator is very well behaved. We consider its inverse first.

$$\begin{aligned}
& \mathscr{B}\ (\mathscr{B}\ s) \\
= \quad & \{ \text{ definition of } \mathscr{B}\text{, twice } \} \\
& head\ s \prec \ \mathscr{B}\ (\mathscr{B}\ s) + \mathscr{B}\ (\mathscr{B}\ s + \mathscr{B}\ (tail\ s)) \\
= \quad & \{ \ \mathscr{B} \text{ is linear } \} \\
& head\ s \prec \ \mathscr{B}\ (\mathscr{B}\ s) + \mathscr{B}\ (\mathscr{B}\ s) + \mathscr{B}\ (\mathscr{B}\ (tail\ s)) \\
= \quad & \{ \text{ arithmetic } \} \\
& head\ s \prec \ 2 * \mathscr{B}\ (\mathscr{B}\ s) + \mathscr{B}\ (\mathscr{B}\ (tail\ s))
\end{aligned}$$

The resulting recursion equation is identical to the definition of $\mathscr{B}$, except for an additional factor of two. If we apply $\mathscr{B}$ a third time, the factor increases to three. Abstracting away from the constant, we define a parametric version of the binomial transform.

$$\begin{aligned} &\mathscr{B} :: (Num\ \alpha) \Rightarrow \alpha \rightarrow Stream\ \alpha \rightarrow Stream\ \alpha \\ &\mathscr{B}_n\ s = t\ \mathbf{where}\ t = head\ s \prec repeat\ n * t + \mathscr{B}_n\ (tail\ s) \end{aligned}$$

The derivation makes clear that $\mathscr{B}_n$ is the n-fold composition of the binomial transform: $\mathscr{B}_n = \mathscr{B}^{(n)}$. Consequently, we have $\mathscr{B}_0 = id$ and $\mathscr{B}_m \cdot \mathscr{B}_n = \mathscr{B}_{m+n}$. The correspondence can even be extended to the integers, in particular, $\mathscr{B}_{-1} = \mathscr{B}^{(-1)}$. The unique fixed-point proof of this fact is too good to be missed: the calculation below shows that $\mathscr{B}_{-1}\ s$ satisfies the recursion equation of $\mathscr{B}^{(-1)} = unfold\ head\ \Delta$.

$$\begin{aligned} &\mathscr{B}_{-1}\ s \\ =\ &\{\ \text{definition of } \mathscr{B}_{-1}\ \} \\ &head\ s \prec -\mathscr{B}_{-1}\ s + \mathscr{B}_{-1}\ (tail\ s) \\ =\ &\{\ \mathscr{B}_{-1} \text{ is linear and definition of } \Delta\ \} \\ &head\ s \prec \mathscr{B}_{-1}\ (\Delta\ s) \end{aligned}$$

This in turn implies that $\mathscr{B}$ and $\mathscr{B}^{(-1)}$ are mutually inverse.

In the introduction to Section 4.2 we remarked that if the unknown sequence is a polynomial, it can be easily identified: the higher-order differences eventually become constant. But can we reconstruct the polynomial from the data obtained? Let us tackle a concrete example (p is some unknown polynomial).

$$\begin{aligned} &\gg\ \mathscr{B}^{(-1)}\ p \\ &\langle 0, 1, 14, 36, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle \end{aligned}$$

Since every element following $(\mathscr{B}^{(-1)}\ p)_4$ is zero, we know that the unknown sequence p is a polynomial of degree 4. But which one? To answer the question, let us put together what we have found out so far.

$$\begin{aligned} &s \\ =\ &\{\ \mathscr{B} \cdot \mathscr{B}^{(-1)} = id\ \} \\ &\mathscr{B}\ (\mathscr{B}^{(-1)}\ s) \\ =\ &\{\ \text{extensionality (16)}\ \} \\ &\langle (\mathscr{B}\ (\mathscr{B}^{(-1)}\ s))_n \mid n \leftarrow nat \rangle \\ =\ &\{\ \text{pointwise characterisation of } \mathscr{B} \text{ (22)}\ \} \\ &\left\langle \sum_k \binom{n}{k} * (\mathscr{B}^{(-1)}\ s)_k \mid n \leftarrow nat \right\rangle \\ =\ &\{\ \text{idioms, see below}\ \} \\ &\sum_k \binom{nat}{k} * repeat\ (\mathscr{B}^{(-1)}\ s)_k \\ =\ &\{\ \text{definition of binomial coefficients: } \binom{n}{k} = n^{\underline{k}} / k!\ \} \\ &\sum_k repeat\ (\mathscr{B}^{(-1)}\ s / fac)_k * nat^{\underline{k}} \end{aligned}$$

(The second but last step is a bit daring, since we turn an infinite sum of elements into an infinite sum of streams. However, infinite summation can be made precise (Rutten, 2005), so let us not worry about this.) The calculation shows that if we divide the sequence $\mathscr{B}^{(-1)}\ s$ by *fac*, then we obtain the coefficients of a polynomial of falling factorial powers.

$$\gg\ \mathscr{B}^{(-1)}\ p\ /\ \mathit{fac}$$
$$\langle 0,1,7,6,1,0,0,0,0,0,0,0,0,0,0,0,\ldots\rangle$$

Table 2 then tells us that $p = \mathit{nat}^{\underline{1}} + 7 * \mathit{nat}^{\underline{2}} + 6 * \mathit{nat}^{\underline{3}} + \mathit{nat}^{\underline{4}} = \mathit{nat}^4$.

As an aside, the sum of binomial coefficients in the second but last line of the derivation is called the *Newton series* of s, which is the finite calculus counterpart of infinite calculus' Taylor series. So, in a nutshell, $\mathscr{B}^{(-1)}$ maps a sequence to the sequence of coefficients of its Newton series.

4.4 Summation

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

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

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

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

We have additionally applied λ-dropping (Danvy, 1999), turning the higher-order equation for Σ into a first-order equation for $\Sigma\ s$ with s fixed. The firstification of the definition enables sharing of computations as illustrated below.

$$\begin{array}{cccccccccccccccc} t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} & \cdots & & t & & \\ \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \cdots & & \parallel & & \\ 0 & s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & \cdots & & 0 & \prec & s \\ & + & + & + & + & + & + & + & + & + & + & \cdots & & & & + \\ & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & \cdots & & & & t \end{array}$$

The illustration also makes clear that $(\Sigma\ s)_n$ is the sum of the first $n-1$ elements of s, excluding $(s)_n$. There is also a variant of Σ that includes $(s)_n$.

$$\begin{aligned} & \Sigma' \quad :: (\mathit{Num}\ \alpha) \Rightarrow \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \\ & \Sigma'\ s = t\ \textbf{where}\ t = \mathit{head}\ s \prec \mathit{tail}\ s + t \end{aligned}$$

The two variants are related by $\Sigma\ s = 0 \prec \Sigma'\ s$ and $\Sigma'\ s = \mathit{tail}\ (\Sigma\ s)$. Usually, Σ is preferable over Σ', as it has more attractive calculational properties. The following

pointwise characterisations relate the stream operators to summation.

$$(\Sigma\ s)_n = \sum_i (s)_i * [0 \leqslant i < n] \quad \text{and} \quad (\Sigma'\ s)_n = \sum_i (s)_i * [0 \leqslant i \leqslant n]$$

The formulas on the right-hand sides make use of *Iverson's convention*: the square brackets[9] turn a Boolean value into a number, with $[\mathit{false}] = 0$ and $[\mathit{true}] = 1$.

Here are some example summations (*A*004520, *A*000290, *A*011371, *A*002275).

$\gg\ \Sigma\ (0 \curlyvee 1)$
$\langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, \ldots \rangle$
$\gg\ \Sigma\ (2 * \mathit{nat} + 1)$
$\langle 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, \ldots \rangle$
$\gg\ \Sigma\ \mathit{carry}$
$\langle 0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 8, 10, 10, 11, 11, \ldots \rangle$
$\gg\ \Sigma\ 10^{\mathit{nat}}$
$\langle 0, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111, 111111111, 1111111111, \ldots \rangle$

The definition of Σ suggests an unusual approach for determining the sum of a sequence: if we observe that a stream satisfies $t = 0 \prec s + t$, then we may conclude that $\Sigma\ s = t$. For example, $\Sigma\ 1 = \mathit{nat}$ as $\mathit{nat} = 0 \prec \mathit{nat} + 1$, $\Sigma\ (2 * \mathit{nat} + 1) = \mathit{nat}^2$ as $\mathit{nat}^2 = 0 \prec \mathit{nat}^2 + 2 * \mathit{nat} + 1$, and $\Sigma\ (1 \prec \mathit{fib}) = \mathit{fib}$ as $\mathit{fib} = 0 \prec (1 \prec \mathit{fib}) + \mathit{fib}$. This is *summation by happenstance*. The latter example also shows that $1 \prec \mathit{fib}$ solves the *difference equation* $x = 1 \prec \Sigma\ x$.

Of course, if we already know the sum, we can use the definition of Σ to verify our conjecture. As an example, let us prove $\Sigma\ \mathit{fib}'^2 = \mathit{fib} * \mathit{fib}'$.

$$\begin{aligned} & \mathit{fib} * \mathit{fib}' \\ = & \quad \{ \text{ definition of } \mathit{fib} \text{ and definition of } \mathit{fib}' \ \} \\ & (0 \prec \mathit{fib}') * (1 \prec \mathit{fib} + \mathit{fib}') \\ = & \quad \{ \text{ arithmetic } \} \\ & 0 \prec \mathit{fib}'^2 + \mathit{fib} * \mathit{fib}' \end{aligned}$$

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

4.4.1 Laws

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

$$t = \Delta\ s \iff \Sigma\ t = s - \mathit{repeat}\ (\mathit{head}\ s) \tag{23}$$

The implication from right to left is easy to show using $\Delta\ (\Sigma\ t) = t$ and $\Delta\ c = 0$. For

[9] Unfortunately, Iverson's convention clashes with Haskell's notation for singleton lists. Both notations are used in this paper, but not simultaneously in the same section.

Table 4. *Laws for summation (c, k and n are constant streams)*

$\Sigma\ (\mathit{tail}\ s) = \mathit{tail}\ (\Sigma\ s) - \mathit{repeat}\ (\mathit{head}\ s)$	$\Sigma\ (n * s) = n * \Sigma\ s$
$\Sigma\ (a \prec s) = 0 \prec \mathit{repeat}\ a + \Sigma\ s$	$\Sigma\ (s + t) = \Sigma\ s + \Sigma\ t$
$\Sigma\ (s \curlyvee t) = (\Sigma\ s + \Sigma\ t) \curlyvee (s + \Sigma\ s + \Sigma\ t)$	$\Sigma\ c^{\mathit{nat}} = (c^{\mathit{nat}} - 1)\ /\ (c - 1)$
$\Sigma\ n = n * \mathit{nat}$	$\Sigma\ (\mathit{nat}^{\underline{n}}) = \mathit{nat}^{\underline{n+1}}\ /\ (n + 1)$
$\Sigma\ (s * \Delta\ t) = s * t - \Sigma\ (\Delta\ s * \mathit{tail}\ t) - \mathit{repeat}\ (\mathit{head}\ (s * t))$	$\Sigma\ \binom{\mathit{nat}}{k} = \binom{\mathit{nat}}{k+1}$

the reverse direction, we reason

$$\begin{aligned} & \Sigma\ (\Delta\ s) \\ = \ & \{\ \text{definition of } \Sigma\ \} \\ & 0 \prec \Sigma\ (\Delta\ s) + \Delta\ s \\ \subset \ & \{\ x = 0 \prec x + \Delta\ s \text{ has a unique solution}\ \} \\ & 0 \prec s - \mathit{repeat}\ (\mathit{head}\ s) + \Delta\ s \\ = \ & \{\ \text{definition of } \Delta \text{ and arithmetic}\ \} \\ & (\mathit{head}\ s \prec \mathit{tail}\ s) - \mathit{repeat}\ (\mathit{head}\ s) \\ = \ & \{\ \text{extensionality (16)}\ \} \\ & s - \mathit{repeat}\ (\mathit{head}\ s) \end{aligned}$$

For instance, $\Sigma\ 2^{\mathit{nat}} = 2^{\mathit{nat}} - 1$, since $2^{\mathit{nat}} = \Delta\ 2^{\mathit{nat}}$ and $\mathit{head}\ (2^{\mathit{nat}}) = 1$. Put differently, 2^{nat} solves the difference equation $x = 1 + \Sigma\ x$.

In Section 3.2.5, we briefly discussed course-of-value recursion, where the value of g_n is defined in terms of all previous values: $g_{n-1}, g_{n-2}, \ldots, g_0$. If we implemented g naîvely, how many recursive calls would g_n actually generate? A moment's reflection suggests that the total number is given by the difference equation $x = \mathit{nat} + \Sigma\ x$, the number of direct calls plus the sum of all indirect calls (note that $\mathit{map}\ \mathit{length} \cdot \mathit{course} = \mathit{nat}$ and $\mathit{map}\ \mathit{sum} \cdot \mathit{course} = \Sigma$).

$$\begin{aligned} & x = \mathit{nat} + \Sigma\ x \\ = \ & \{\ \text{arithmetic}\ \} \\ & \Sigma\ x = x - \mathit{nat} \\ = \ & \{\ \text{Fundamental Theorem (23) and } \mathit{head}\ (x - \mathit{nat}) = 0\ \} \\ & x = \Delta\ (x - \mathit{nat}) \\ = \ & \{\ \Delta \text{ is linear and } \Delta\ \mathit{nat} = 1\ \} \\ & x = \Delta\ x - 1 \end{aligned}$$

The resulting equation is solved by $2^{\mathit{nat}} - 1$, which shows why a naîve implementation of g is prohibitive.

Using the Fundamental Theorem we can transform the rules in Table 3 into rules for summation, see Table 4. As an example, the rule for products, *summation by*

parts, can be derived from the product rule of Δ. Let $c = repeat\ (head\ (s * t))$, then

$$
\begin{aligned}
& s * \Delta\, t + \Delta\, s * tail\ t = \Delta\, (s * t) \\
\iff\ & \{\ \text{Fundamental Theorem (23)}\ \} \\
& \Sigma\, (s * \Delta\, t + \Delta\, s * tail\ t) = s * t - c \\
\iff\ & \{\ \Sigma \text{ is linear}\ \} \\
& \Sigma\, (s * \Delta\, t) + \Sigma\, (\Delta\, s * tail\ t) = s * t - c \\
\iff\ & \{\ \text{arithmetic}\ \} \\
& \Sigma\, (s * \Delta\, t) = s * t - \Sigma\, (\Delta\, s * tail\ t) - c
\end{aligned}
$$

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 infinite calculus.

The only slightly tricky derivation is the one for interleaving.

$$
\begin{aligned}
& (t - s) \curlyvee (tail\ s - t) = \Delta\, (s \curlyvee t) \\
=\ & \{\ \text{Fundamental Theorem (23) and } head\ (s \curlyvee t) = head\ s\ \} \\
& \Sigma\, ((t - s) \curlyvee (tail\ s - t)) = (s \curlyvee t) - repeat\ (head\ s)
\end{aligned}
$$

At first glance, we are stuck. To make progress, let us introduce two fresh variables: $x = t - s$ and $y = tail\ s - t$. If we can express s and t in terms of x and y, then we have found the desired formula.

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

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

4.4.2 *Examples*

Using the rules in Table 4 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* ($0 \prec A000330$).

$$
\begin{aligned}
& \Sigma\, nat^{2} \\
= \quad & \{ \text{ converting to falling factorial powers (Table 2) } \} \\
& \Sigma\, (nat^{\underline{2}} + nat^{\underline{1}}) \\
= \quad & \{ \text{ summation laws (Table 4) } \} \\
& \tfrac{1}{3} * nat^{\underline{3}} + \tfrac{1}{2} * nat^{\underline{2}} \\
= \quad & \{ \text{ converting to ordinary powers (Table 2) } \} \\
& \tfrac{1}{3} * (nat^{3} - 3 * nat^{2} + 2 * nat) + \tfrac{1}{2} * (nat^{2} - nat) \\
= \quad & \{ \text{ arithmetic } \} \\
& \tfrac{1}{6} * (nat - 1) * nat * (2 * nat - 1)
\end{aligned}
$$

Calculating the summation of a product, say, $\Sigma\, (nat * c^{nat})$ 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^{nat}. Let us see how it goes.

$$
\begin{aligned}
& \Sigma\, (nat * c^{nat}) \\
= \quad & \{ \; \Delta\, c^{nat} = (c - 1) * c^{nat} \; \} \\
& \Sigma\, (nat * \Delta\, c^{nat} \,/\, (c - 1)) \\
= \quad & \{ \; \Sigma \text{ is linear } \} \\
& \Sigma\, (nat * \Delta\, c^{nat}) \,/\, (c - 1) \\
= \quad & \{ \text{ summation by parts (Table 4) } \} \\
& (nat * c^{nat} - \Sigma\, (\Delta\, nat * tail\ c^{nat})) \,/\, (c - 1) \\
= \quad & \{ \; \Delta\, nat = 1,\ c \text{ is constant, and definition of } nat \; \} \\
& (nat * c^{nat} - c * \Sigma\, c^{nat}) \,/\, (c - 1) \\
= \quad & \{ \text{ summation laws (Table 4) } \} \\
& (nat * c^{nat} - c * (c^{nat} - 1) \,/\, (c - 1)) \,/\, (c - 1) \\
= \quad & \{ \text{ arithmetic } \} \\
& (((c - 1) * nat - c) * c^{nat} + c) \,/\, (c - 1)^{2}
\end{aligned}
$$

That was not too hard.

The summation rule for interleaving is useful to solve summations that involve alternating signs.

$$\begin{aligned}
& \Sigma\,((-1)^{nat} * nat) \\
= {} & \quad \{\ (-1)^{nat} = 1 \curlyvee -1 \text{ and characterisation of } nat\ \} \\
& \Sigma\,(2 * nat \curlyvee -(2 * nat + 1)) \\
= {} & \quad \{\ \text{summation laws (Table 4)}\ \} \\
& \Sigma\,(-1) \curlyvee (2 * nat + \Sigma\,(-1)) \\
= {} & \quad \{\ \text{summation laws (Table 4) and arithmetic}\ \} \\
& -nat \curlyvee nat
\end{aligned}$$

As a final example, let us tackle another sum that involves the interleaving operator: $\Sigma\ carry$ ($A011371$). The sum is important, as it determines the amortised running time of the binary increment. Going back to the interactive session, Section 4.4, we observe that the sum is always at most nat, which would imply that the amortised running time, $\Sigma\ carry\ /\ nat$, is constant. That is nice, but can we actually quantify the difference? Let us 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 leading 1s to 0s and flips the first 0 to 1. Now, since *carry* defines the number of leading 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, showing that they both satisfy the recursion equation $x = 0 \prec 1 \prec x + 1 - (carry \curlyvee carry)$.) Now we can invoke the *summation by happenstance* rule—the fact that $\Sigma\ s$ is the unique solution of $x = 0 \prec s + x$.

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

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

That was fun. But surely, the interleaving rule would yield the result directly, would not it? Let us try.

$$
\begin{aligned}
& \Sigma\ \mathit{carry} \\
= \quad & \{ \text{ definition of } \mathit{carry} \ \} \\
& \Sigma\ (0 \curlyvee \mathit{carry} + 1) \\
= \quad & \{ \text{ summation laws (Table 4) } \} \\
& \Sigma\ (\mathit{carry} + 1) \curlyvee \Sigma\ (\mathit{carry} + 1) \\
= \quad & \{ \ \Sigma \text{ is linear and } \Sigma\ 1 = \mathit{nat} \ \} \\
& (\Sigma\ \mathit{carry} + \mathit{nat}) \curlyvee (\Sigma\ \mathit{carry} + \mathit{nat})
\end{aligned}
$$

This is quite a weird property. Since we know where we are aiming at, let us determine $\mathit{nat} - \Sigma\ \mathit{carry}$.

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

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

4.4.3 Perturbation method

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

$$\Sigma\ s = \Sigma\ (\mathit{tail}\ \ s) - s + \mathit{repeat}\ \ (\mathit{head}\ \ s) \tag{24}$$

The idea of the method is to try to express $\Sigma\ (\mathit{tail}\ \ s)$ in terms of $\Sigma\ s$. Then we obtain an equation whose solution is the sum we seek. Let us try the method on a sum we

have done before.

$$
\begin{aligned}
& \Sigma\ (nat * c^{nat}) \\
= \quad & \{ \text{ perturbation (24) and } head\ (nat * c^{nat}) = 0 \ \} \\
& \Sigma\ (tail\ (nat * c^{nat})) - nat * c^{nat} \\
= \quad & \{ \text{ definition of } nat \ \} \\
& \Sigma\ ((nat + 1) * c^{nat+1}) - nat * c^{nat} \\
= \quad & \{ \text{ summation laws (Table 4) } \} \\
& c * \Sigma\ (nat * c^{nat}) + c * \Sigma\ c^{nat} - nat * c^{nat} \\
= \quad & \{ \text{ summation laws (Table 4) } \} \\
& c * \Sigma\ (nat * c^{nat}) + c * (c^{nat} - 1)\ /\ (c - 1) - nat * c^{nat}
\end{aligned}
$$

The sum $\Sigma\ (nat * c^{nat})$ 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 Section 4.4.2:

$$
\begin{aligned}
& \Sigma\ (nat * c^{nat}) \\
= \quad & \{ \text{ see above } \} \\
& (c * (c^{nat} - 1)\ /\ (c - 1) - nat * c^{nat})\ /\ (1 - c) \\
= \quad & \{ \text{ arithmetic } \} \\
& (((c - 1) * nat - c) * c^{nat} + c)\ /\ (c - 1)^2
\end{aligned}
$$

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 naîve way of summing: the nth element is computed using n additions not re-using any previous results.

4.4.4 Example: Moessner's Theorem

In the 1950s Alfred Moessner discovered the following intriguing scheme for generating the natural nth powers (1951): From the sequence of natural numbers, delete every nth number and form the sequence of partial sums. From the resulting sequence, delete every $(n - 1)$-st number and form again the partial sums. Repeat this step $n - 1$ times.

If we repeat the transformation 0 times, we obtain the first powers of the naturals. The second simplest instance of the process yields the squares by summing up the odd numbers.

$$
\begin{array}{cccccccccc}
1 & \cancel{2} & 3 & \cancel{4} & 5 & \cancel{6} & 7 & \cancel{8} & \ldots \\
1 & & 4 & & 9 & & 16 & & \ldots
\end{array}
$$

For generating the cubes we perform two deletion-and-summation steps.

$$
\begin{array}{ccccccccccccc}
1 & 2 & \cancel{3} & 4 & 5 & \cancel{6} & 7 & 8 & \cancel{9} & 10 & 11 & \cancel{12} & \ldots \\
1 & \cancel{3} & & 7 & \cancel{12} & & 19 & \cancel{27} & & 37 & \cancel{48} & & \ldots \\
1 & & & 8 & & & 27 & & & 64 & & & \ldots
\end{array}
\tag{25}
$$

The second sequence is probably unfamiliar—the numbers are the 'three-quarter squares' (*A*077043)—but the final sequence is the desired sequence of cubes. Actually, we can add another deletion-and-summation step and start off with a sequence of ones.

$$\begin{array}{c|cccc|cccc|cccc|c} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\ 0 & & 1 & 2 & 3 & & 4 & 5 & 6 & & 7 & 8 & 9 & \cdots \\ 0 & & & 1 & 3 & & & 7 & 12 & & & 19 & 27 & \\ 0 & & & & 1 & & & & 8 & & & & 27 & \end{array} \tag{26}$$

As before, we have replaced the deleted elements by spaces in the rows below, so that the numbers form a sequence of triangles. To further emphasise the structure, we have shifted the triangles to the right: ◸ becomes ◹—the following calculations are slightly more attractive then. We have also added a 'seed' column to the left, so that every element of (25) is the sum of its north-western and its western neighbour in (26). Note that the first triangle is now a prefix of Pascal's triangle rotated 90° counterclockwise.

Turning to the specification of Moessner's process, the main challenge is to model the deletion step. One approach is simply to name the elements.

$$\begin{array}{c|cccc|cccc|cccc|c} 1 & a^0_{0,0} & a^0_{1,0} & \cdots & a^0_{n,0} & a^1_{0,0} & a^1_{1,0} & \cdots & a^1_{n,0} & a^2_{0,0} & a^2_{1,0} & \cdots & a^2_{n,0} & \\ 0 & & a^0_{1,1} & \cdots & a^0_{n,1} & & a^1_{1,1} & \cdots & a^1_{n,1} & & a^2_{1,1} & \cdots & a^2_{n,1} & \cdots \\ \vdots & & & \ddots & \vdots & & & \ddots & \vdots & & & \ddots & \vdots & \\ 0 & & & & a^0_{n,n} & & & & a^1_{n,n} & & & & a^2_{n,n} & \end{array}$$

The index variable i in $a^k_{i,j}$ ranges over the columns of a triangle, j over the rows, and k over the triangles ($0 \leqslant j \leqslant i \leqslant n$ and $k \geqslant 0$). The 'name and conquer' approach with its use of three index variables is not too attractive, even though this is the approach taken by most of the proofs in the literature (Perron, 1951; Paasche, 1952; Salié, 1952). Fortunately, since the process generates an infinite number of triangles, we can eliminate the index variable k and define a triangle of streams instead:

$$\begin{array}{c|cccc} r_0 & s_{0,0} & s_{1,0} & \cdots & s_{n,0} \\ r_1 & & s_{1,1} & \cdots & s_{n,1} \\ \vdots & & & \ddots & \vdots \\ r_n & & & & s_{n,n} \end{array}$$

where $(s_{i,j})_k = a^k_{i,j}$. The streams $r_0, r_1, \ldots, r_n$ model the columns to the left of the triangles: their heads form the seed column $(1, 0, \ldots, 0)$ and their tails equal the last column of the preceding triangles $(s_{n,0}, s_{n,1}, \ldots, s_{n,n})$.

$$r_j = [j = 0] \prec s_{n,j} \tag{27}$$

(Recall that the square brackets are syntax for Iverson's convention.) The following equations capture the manual process of calculating the sequences $s_{i,j}$.

$$\begin{aligned} s_{i,0} &= r_0 \\ s_{i+1,j+1} &= s_{i,j} + r_{j+1} * [i = j] + s_{i,j+1} * [i > j] \end{aligned} \tag{28}$$

Each of the streams $s_{i+1,j+1}$ is the sum of its north-western ($s_{i,j}$) and its western neighbour ($r_{j+1} * [i = j] + s_{i,j+1} * [i > j]$). Moessner's Theorem then states that

$$r_n = nat^n \qquad \text{or, equivalently,} \qquad s_{n,n} = (nat + 1)^n$$

Turning to the proof, first observe that r_j depends on $s_{n,j}$, which in turn (indirectly) depends on r_j. So, we have effectively turned the verbal description of Moessner's process into a system of recursion equations. To illustrate, here is the system for $n = 3$:

$$\begin{array}{lllll}
r_0 = 1 \prec s_{3,0} & s_{0,0} = r_0 & s_{1,0} = r_0 & s_{2,0} = r_0 & s_{3,0} = r_0 \\
r_1 = 0 \prec s_{3,1} & & s_{1,1} = s_{0,0} + r_1 & s_{2,1} = s_{1,0} + s_{1,1} & s_{3,1} = s_{2,0} + s_{2,1} \\
r_2 = 0 \prec s_{3,2} & & & s_{2,2} = s_{1,1} + r_2 & s_{3,2} = s_{2,1} + s_{2,2} \\
r_3 = 0 \prec s_{3,3} & & & & s_{3,3} = s_{2,2} + r_3
\end{array}$$

The form of the equations does not quite meet the requirements of Section 2.3. This can be remedied, however, if we repeatedly in-line the definitions of $s_{i,j}$. For $n = 3$, the resulting equations for $s_{i,j}$ are (for reasons of space we have omitted the multiplication symbol $*$):

$$\begin{array}{llll}
s_{0,0} = 1r_0 & s_{1,0} = 1r_0 & s_{2,0} = 1r_0 & s_{3,0} = 1r_0 \\
& s_{1,1} = 1r_0{+}1r_1 & s_{2,1} = 2r_0{+}1r_1 & s_{3,1} = 3r_0{+}1r_1 \\
& & s_{2,2} = 1r_0{+}1r_1{+}1r_2 & s_{3,2} = 3r_0{+}2r_1{+}1r_2 \\
& & & s_{3,3} = 1r_0{+}1r_1{+}1r_2{+}1r_3
\end{array}$$

The coefficients in each column form a prefix of Pascal's triangle rotated 90° clockwise. In general, the streams $s_{i,j}$ and r_j are related by

$$s_{i,j} \quad = \quad \sum_k \binom{i-k}{j-k} * r_k \tag{29}$$

The proof that (29) is equivalent to (28) is a straightforward nested induction, making essential use of the addition formula for binomial coefficients. It is worth noting that relation (29) holds for arbitrary r_j. (Also, r_j and $s_{i,j}$ do not have to be streams.) Using the relation we can simplify the original system of recursion equations, (27) and (28), to

$$r_j \quad = \quad [j = 0] \prec \sum_k \binom{n-k}{j-k} * r_k \tag{30}$$

For our running example ($n = 3$), we obtain the system on the left—we have simply inlined the definitions of $s_{n,j}$ into the equations $r_j = [j = 0] \prec s_{n,j}$.

$$\begin{array}{ll}
r_0 = 1 \prec 1 * r_0 & r_0 = 1 \\
r_1 = 0 \prec 3 * r_0 + 1 * r_1 & r_1 = \Sigma\,(3 * r_0) \\
r_2 = 0 \prec 3 * r_0 + 2 * r_1 + 1 * r_2 & r_2 = \Sigma\,(3 * r_0 + 2 * r_1) \\
r_3 = 0 \prec 1 * r_0 + 1 * r_1 + 1 * r_2 + 1 * r_3 & r_3 = \Sigma\,(1 * r_0 + 1 * r_1 + 1 * r_2)
\end{array}$$

Having eliminated the doubly indexed stream variable $s_{i,j}$, we are left with the task of solving a system of n equations. Since r_0 is the constant stream 1 and since

the other streams are partial sums (see above on the right)

$$r_{j+1} = \Sigma \left(\sum_k \binom{n-k}{j+1-k} * r_k * [k \leqslant j] \right) \tag{31}$$

we can systematically solve the recursion system top to bottom. Here are the calculations for $n = 3$:

$$\begin{aligned}
r_0 &= 1 \\
r_1 &= \Sigma \left(\binom{3}{1} * r_0 \right) && = 3 * \Sigma\, 1 && = 3 * nat \\
r_2 &= \Sigma \left(\binom{3}{2} * r_0 + \binom{2}{1} * r_1 \right) && = 3 * \Sigma\, (1 + 2 * nat) && = 3 * nat^2 \\
r_3 &= \Sigma \left(\binom{3}{3} * r_0 + \binom{2}{2} * r_1 + \binom{1}{1} * r_2 \right) && = \Sigma\, (1 + 3 * nat + 3 * nat^2) && = nat^3
\end{aligned}$$

The summations are easy to calculate by hand. In particular, we do not need to refer to Table 4. (Do you see why?)

Actually, it is not too hard to prove Moessner's Theorem in its full glory. The following calculation, which attempts to show $tail\ r_n = (nat + 1)^n$, motivates a formula for r_j.

$$\begin{aligned}
& tail\ r_n \\
= \quad & \{ \text{ characterisation of } r_n \text{ (30) } \} \\
& \sum_k \binom{n-k}{n-k} * r_k \\
= \quad & \left\{ \binom{i}{i} = [0 \leqslant i] \right\} \\
& \sum_k r_k * [k \leqslant n] \\
= \quad & \left\{ \text{conjecture} : r_j = \binom{n}{j} * nat^j \right\} \\
& \sum_k \binom{n}{k} * nat^k * [k \leqslant n] \\
= \quad & \{ \text{ Binomial Theorem } \} \\
& (nat + 1)^n
\end{aligned}$$

Summarising our findings, we conjecture

$$r_j = \binom{n}{j} * nat^j \qquad \text{and} \qquad s_{i,j} = \sum_k \binom{i-k}{j-k} * \binom{n}{k} * nat^k$$

For the proof we show that the streams $\binom{n}{j} * nat^{j}$ are indeed the unique solutions of (30).

$$
\begin{aligned}
& [j = 0] \prec \sum_k \binom{n-k}{j-k} * \binom{n}{k} * nat^{k} \\
= \; & \{ \text{ trinomial revision (Graham et al., 1994, p. 174) } \} \\
& [j = 0] \prec \sum_k \binom{n}{j} * \binom{j}{k} * nat^{k} \\
= \; & \{ \text{ distributive law } \} \\
& [j = 0] \prec \binom{n}{j} * \sum_k \binom{j}{k} * nat^{k} \\
= \; & \{ \text{ Binomial Theorem } \} \\
& [j = 0] \prec \binom{n}{j} * (nat + 1)^{j} \\
= \; & \{ [j = 0] = 0^{j} \} \\
& 0^{j} \prec \binom{n}{j} * (nat + 1)^{j} \\
= \; & \{ \text{ definition of } nat \} \\
& \binom{n}{j} * nat^{j}
\end{aligned}
$$

Since $r_n = nat^{n}$, Moessner's Theorem follows. The proof illustrates the importance of avoiding unnecessary detail: the central step in the derivation is the elimination of the stream variables $s_{i,j}$ from the original system of recursion equations.

Section 5.3.3 presents an alternative proof of this theorem, which avoids double subscripts and binomial coefficients. Instead, it builds on the theory of generating functions, which we introduce next.

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, $a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots$, in some formal variable z. It is an alternative view and we shall see that it provides us with additional operators and techniques for manipulating numeric streams.

5.1 Power series

Let us put on the 'power series glasses'. The simplest series, the constant function a_0 and the identity z (*A*063524), are given by

$$
\begin{aligned}
& const \quad :: (Num\ \alpha) \Rightarrow \alpha \rightarrow Stream\ \alpha \\
& const\ n = n \prec repeat\ 0 \\
& z :: (Num\ \alpha) \Rightarrow Stream\ \alpha \\
& z = 0 \prec 1 \prec repeat\ 0
\end{aligned}
$$

The sum of two power series is implemented by lifted addition. The successor function, for instance, is $const\ 1 + z$. The product of two series, however, is not given by lifted multiplication, since, for example, $(const\ 1 + z) * (const\ 1 + z) = const\ 1 + z$. 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.

$$s = const\ (head\ s) + z \times tail\ s \tag{32}$$

The rule implies $z \times s = 0 \prec s$. The derivation of $\times$ proceeds as follows—we assume that $\times$ is associative and distributes over addition.

$$\begin{aligned}
& s \times t \\
= \; & \{ \text{ Horner's rule (32) } \} \\
& (const\ (head\ s) + z \times tail\ s) \times t \\
= \; & \{ \text{ arithmetic } \} \\
& const\ (head\ s) \times t + z \times tail\ s \times t \\
= \; & \{ \text{ Horner's rule (32) } \} \\
& const\ (head\ s) \times (const\ (head\ t) + z \times tail\ t) + z \times tail\ s \times t \\
= \; & \{ \text{ arithmetic } \} \\
& const\ (head\ s) \times const\ (head\ t) + const\ (head\ s) \times z \times tail\ t + z \times tail\ s \times t \\
= \; & \{ \; const\ a \times const\ b = const\ (a * b) \text{ and arithmetic } \} \\
& const\ (head\ s * head\ t) + z \times (const\ (head\ s) \times tail\ t + tail\ s \times t) \\
= \; & \{ \text{ Horner's rule (32) } \} \\
& head\ s * head\ t \prec const\ (head\ s) \times tail\ t + tail\ s \times t
\end{aligned}$$

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 $const\ k \times s$ by $repeat\ k * s$ (this law also follows from Horner's rule).

$$\begin{aligned}
&(\times) \quad :: (Num\ \alpha) \Rightarrow Stream\ \alpha \rightarrow Stream\ \alpha \rightarrow Stream\ \alpha \\
&s \times t = head\ s * head\ t \prec repeat\ (head\ s) * tail\ t + tail\ s \times t
\end{aligned}$$

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

$\gg\ nat \times 10^{nat}$
$\langle 0, 1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, 123456789, \ldots \rangle$
$\gg\ 9 * (nat \times 10^{nat})$
$\langle 0, 9, 108, 1107, 11106, 111105, 1111104, 11111103, 111111102, 1111111101, \ldots \rangle$
$\gg\ 9 * (nat \times 10^{nat}) + nat'$
$\langle 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111, 111111111, 1111111111, \ldots \rangle$

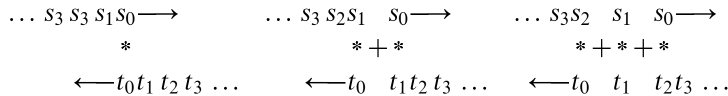
The operator $\times$ is also called *convolution product*. The first example suggests a pointwise characterisation: the convolution product of the streams $\langle s_0, s_1, s_2, \ldots \rangle$ and

$\langle t_0, t_1, t_2, \dots \rangle$ is $\langle s_0 * t_0, s_0 * t_1 + s_1 * t_0, s_0 * t_2 + s_1 * t_1 + s_2 * t_0, \dots \rangle$. In general, we have

$$(s \times t)_n = \sum_{i,j} (s)_i * (t)_j * [i + j = n]$$

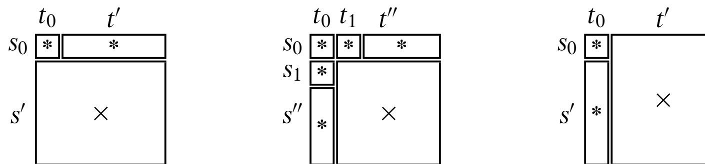
So the nth element of the convolution product is the sum of all products whose subscripts add up to n.

There is a nice non-mathematical explanation of the convolution product. Imagine two rows of people shaking hands while passing in opposite directions. First, the two leaders shake hands; then the first shakes hand with the second of the other row and vice versa; then the first shakes hand with the third of the other row and so forth.



The handshake corresponds to a multiplication, the results of which are added up.

Unfortunately, the symmetry of the description is lost in the implementation above. There are, at least, two other ways to set up the corecursion (we abbreviate *head* s by s_0 and *tail* s by s').



The first element of the convolution product is $s_0 * t_0$. Then we can either add *repeat* $s_0 * t'$ to $s' \times t$ (diagram on the left) or $s \times t'$ to $s' * \textit{repeat}\ t_0$ (diagram on the right). For a symmetric definition (diagram in the middle), we have to expand s and t twice. Fortunately, since addition is associative, all three variants are equivalent (Hinze, to appear a).

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 (*recip* is a member of the class *Fractional*).

$$s \times \textit{recip}\ s = \textit{const}\ 1$$

We reason

$$\begin{aligned} & \textit{head}\ s * \textit{head}\ (\textit{recip}\ s) = 1 \\ \iff & \quad \{ \text{ arithmetic } \} \\ & \textit{head}\ (\textit{recip}\ s) = \textit{recip}\ (\textit{head}\ s) \end{aligned}$$

and

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

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

$$
\begin{aligned}
& \mathit{recip}\ s = t\ \textbf{where}\ a = \mathit{recip}\ (\mathit{head}\ s) \\
& \qquad\qquad\qquad\quad t = a \prec \mathit{repeat}\ (-a) * (\mathit{tail}\ s \times t) \\
& s \div t = s \times \mathit{recip}\ t
\end{aligned}
$$

Note that $\mathit{recip}\ s$ is not defined if $\mathit{head}\ s = 0$. We use $s^{\lfloor n \rfloor}$, where n is a natural number, to denote iterated convolution with $s^{\lfloor 0 \rfloor} = \mathit{const}\ 1$ and set $s^{\lfloor -n \rfloor} = (\mathit{recip}\ s)^{\lfloor n \rfloor}$. Like $\mathit{recip}\ s$, the negative power $s^{\lfloor -n \rfloor}$ is not defined if $\mathit{head}\ s = 0$. For instance, $z^{\lfloor -1 \rfloor}$ does not exist.

5.2 Laws

The familiar arithmetic laws also hold for $\mathit{const}\ n$, $+$, $-$, $\times$ and $\div$. Perhaps surprisingly, we can reformulate the numeric streams we have 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* (gf) 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 $\mathit{const}\ n$, $\times$ and $s^{\lfloor n \rfloor}$ with $\mathit{repeat}\ n$, $*$ and s^n.

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

$$
\begin{aligned}
& \mathit{repeat}\ a = a \prec \mathit{repeat}\ a \\
\iff & \quad \{ \text{ Horner's rule (32) } \} \\
& \mathit{repeat}\ a = \mathit{const}\ a + z \times \mathit{repeat}\ a \\
\iff & \quad \{ \text{ arithmetic } \} \\
& \mathit{const}\ 1 \times \mathit{repeat}\ a - z \times \mathit{repeat}\ a = \mathit{const}\ a \\
\iff & \quad \{ \text{ arithmetic } \} \\
& (\mathit{const}\ 1 - z) \times \mathit{repeat}\ a = \mathit{const}\ a \\
\iff & \quad \{ \text{ arithmetic } \} \\
& \mathit{repeat}\ a = \mathit{const}\ a \div (\mathit{const}\ 1 - z)
\end{aligned}
$$

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

Table 5. *Streams and their generating functions*

$repeat\ a$	$=$	$const\ a \div (const\ 1 - z)$
$(repeat\ a)^{nat}$	$=$	$const\ 1 \div (const\ 1 - const\ a \times z)$
$\Sigma\ s$	$=$	$s \times z \div (const\ 1 - z)$
$\Sigma^{(n)}\ s$	$=$	$s \times z^{\lfloor n \rfloor} \div (const\ 1 - z)^{\lfloor n \rfloor}$
nat	$=$	$z \div (const\ 1 - z)^{\lfloor 2 \rfloor}$
$nat + 1$	$=$	$const\ 1 \div (const\ 1 - z)^{\lfloor 2 \rfloor}$
$\binom{n}{nat}$	$=$	$(const\ 1 + z)^{\lfloor n \rfloor}$
$\binom{n}{nat} * (repeat\ a)^{nat}$	$=$	$(const\ 1 + const\ a \times z)^{\lfloor n \rfloor}$
$\binom{nat}{k}$	$=$	$z^{\lfloor k \rfloor} \div (const\ 1 - z)^{\lfloor k+1 \rfloor}$

Geometric sequences are not much harder.

$$\begin{aligned}
& (repeat\ a)^{nat} \\
= \ & \{ \text{ definition of exponentiation and definition of } nat \} \\
& 1 \prec repeat\ a * (repeat\ a)^{nat} \\
= \ & \{ \text{ Horner's rule (32) and } repeat\ k * s = const\ k \times s \} \\
& const\ 1 + z \times const\ a \times (repeat\ a)^{nat}
\end{aligned}$$

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

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

$$\begin{aligned}
& \Sigma\ s = 0 \prec \Sigma\ s + s \\
\iff \ & \{ \text{ Horner's rule (32) } \} \\
& \Sigma\ s = z \times (\Sigma\ s + s) \\
\iff \ & \{ \text{ arithmetic } \} \\
& \Sigma\ s = s \times z \div (const\ 1 - z)
\end{aligned}$$

This implies that the generating function of the natural numbers is $nat = \Sigma\ (repeat\ 1) = z \div (const\ 1 - z)^{\lfloor 2 \rfloor}$. If we repeatedly sum the stream of ones, then we obtain the *columns* of Pascal's triangle.

$$\binom{nat}{k} = z^{\lfloor k \rfloor} \div (const\ 1 - z)^{\lfloor k+1 \rfloor}$$

The formula for a *row* is an immediate consequence of the Binomial Theorem.

$$\binom{n}{nat} = (z + const\ 1)^{\lfloor n \rfloor}$$

Repeated summation of an arbitrary stream is not any harder.

$$\Sigma^{(n)}\ s = s \times z^{\lfloor n \rfloor} \div (const\ 1 - z)^{\lfloor n \rfloor}$$

Table 5 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*, $(repeat\ a)^{nat}$ or *nat*. This is what we do in the next sections.

5.3 Examples

5.3.1 Vandermonde's convolution

Many identities between binomial coefficients have straightforward explanations in terms of generating functions. *Vandermonde's convolution* serves as a particularly beautiful illustration of this observation. Graham *et al.* (1994) phrase the identity as a sum of products of binomial coefficients.

$$\sum_k \binom{r}{m+k} * \binom{s}{n-k} \quad = \quad \binom{r+s}{m+n}$$

The bound variable k appears in two sub-terms, $m+k$ and $n-k$, whose sum is constant. In other words, the expression on the left-hand side is a convolution product in disguise. Using this insight, we can rephrase the identity as an equation over streams.

$$\binom{r}{nat} \times \binom{s}{nat} \quad = \quad \binom{r+s}{nat}$$

The proof of the law is a breeze, if we express the streams in terms of generating functions.

$$\begin{aligned}
& \binom{r}{nat} \times \binom{s}{nat} \\
= \quad & \{ \text{ Table 5 } \} \\
& (const\ 1 + z)^{\lfloor r \rfloor} \times (const\ 1 + z)^{\lfloor s \rfloor} \\
= \quad & \{ \text{ arithmetic: laws of exponentials } \} \\
& (const\ 1 + z)^{\lfloor r+s \rfloor} \\
= \quad & \{ \text{ Table 5 } \} \\
& \binom{r+s}{nat}
\end{aligned}$$

5.3.2 Repunits

The session in Section 5.1 shows an intriguing way to generate the sequence of repunits $\langle 1, 11, 111, \ldots \rangle$, where a repunit contains only the digit 1. The example is based on the decimal number system, but it can be readily generalised to an arbitrary base b.

$$repeat\ (b-1) * (nat \times (repeat\ b)^{nat}) + nat + 1 = tail\ (\Sigma\ (repeat\ b)^{nat})$$

The sum on the right-hand side models the sequence of base b repunits $\langle 1_b, 11_b, 111_b, \ldots\rangle$. The proof of the identity again uses generating functions (we abbreviate *const* 1 by $\dot{1}$).

$$\begin{aligned}
& \mathit{repeat}\ (b-1) * (\mathit{nat} \times (\mathit{repeat}\ b)^{\mathit{nat}}) + \mathit{nat} + 1 \\
= & \quad \{\ \mathit{repeat}\ k * s = \mathit{const}\ k \times s\ \} \\
& \mathit{const}\ (b-1) \times \mathit{nat} \times (\mathit{repeat}\ b)^{\mathit{nat}} + \mathit{nat} + 1 \\
= & \quad \{\ \text{Table 5}\ \} \\
& \mathit{const}\ (b-1) \times z \div (\dot{1} - z)^{\lfloor 2 \rfloor} \times \dot{1} \div (\dot{1} - \mathit{const}\ b \times z) + \dot{1} \div (\dot{1} - z)^{\lfloor 2 \rfloor} \\
= & \quad \{\ \text{arithmetic}\ \} \\
& \dot{1} \div (\dot{1} - z) \times \dot{1} \div (\dot{1} - \mathit{const}\ b \times z) \\
= & \quad \{\ \mathit{tail}\ (z \times s) = s\ \} \\
& \mathit{tail}\ (z \div (\dot{1} - z) \times \dot{1} \div (\dot{1} - \mathit{const}\ b \times z)) \\
= & \quad \{\ \text{Table 5}\ \} \\
& \mathit{tail}\ (\Sigma\ (\mathit{repeat}\ b)^{\mathit{nat}})
\end{aligned}$$

5.3.3 Moessner's Theorem, revisited

In Section 4.4.4 we have presented a proof of Moessner's Theorem, which involved some binomial wizardry. In this section we follow an alternative route, tackling the problem using convolutions and generating functions. The resulting proof is more calculational, in particular, it avoids double subscripts and binomial coefficients. For the calculations, it will be convenient to abbreviate *const* $(s)_n$ by $(s)_n$ and *const* $[b]$ by $[b]$, where the square brackets are syntax for Iverson's convention.

As a reminder, here is the process that generates the cubes.

$$\begin{array}{c|ccc|c|ccc|c|ccc|c|c}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
0 & & 1 & 2 & 3 & & 4 & 5 & 6 & & 7 & 8 & 9 & \ldots \\
0 & & & 1 & 3 & & & 7 & 12 & & & 19 & 27 & \ldots \\
0 & & & & 1 & & & & 8 & & & & 27 & \ldots
\end{array}$$

The verbal description suggests that the sequences are formed from top to bottom. An alternative view is that they are formed from left to right: We start with the sequence $\langle 1, 0, 0, 0, 0 \ldots\rangle$ (leftmost column read from top to bottom). The sequence goes through a *triangle transformation*, which yields the sequence $\langle 1, 3, 3, 1, 0 \ldots\rangle$. This sequence goes through the same transformation yielding $\langle 1, 6, 12, 8, 0 \ldots\rangle$, and so forth. The sequence of elements at position 3 are the cubes. In a sense, we have already justified this alternative view in Section 4.4.4, when we showed that the streams $s_{i,j}$ can be defined solely in terms of r_j—recall that the streams r_j model the columns to the left of the triangles. The slight twist here is that we switch from row-major to column-major order.

If we view the columns through our 'power series glasses', the summation steps can be modelled by a simple multiplication: if g_i is some column, then $g_i \times (\dot{1} + z)$ is the next column (again we abbreviate *const* 1 by $\dot{1}$). Recall that each element of

a triangle is the sum of its western and its north-western neighbour; g_i represents the former and $g_i \times z$ the latter (g_i shifted a row downwards). Additionally taking the 'input column', called f, into account, the columns $g_1, \ldots, g_n$ of a triangle are related by

$$\begin{aligned} g_0 &= (f)_0 \times z^{\lfloor 0 \rfloor} \\ g_{i+1} &= (f)_{i+1} \times z^{\lfloor i+1 \rfloor} + g_i \times (\dot{1} + z) \end{aligned}$$

For $n = 3$, we obtain the system of equations on the left below. For instance, if the input column f is $\langle 1, 6, 12, 8, 0 \ldots \rangle$, then the output column g_3 is $\langle 1, 9, 27, 27, 0 \ldots \rangle$.

$$\begin{aligned} g_0 &= (f)_0 \times z^{\lfloor 0 \rfloor} \\ g_1 &= (f)_1 \times z^{\lfloor 1 \rfloor} + g_0 \times (\dot{1} + z) \\ g_2 &= (f)_2 \times z^{\lfloor 2 \rfloor} + g_1 \times (\dot{1} + z) \\ g_3 &= (f)_3 \times z^{\lfloor 3 \rfloor} + g_2 \times (\dot{1} + z) \end{aligned}$$

f	g_0	g_1	g_2	g_3
1	1	1	1	1
6		7	8	9
12			19	27
8				27

Distributing the products over the sums, it is not hard to see that the generating functions satisfy the invariant

$$g_{i+1} \times (\dot{1} + z)^{\lfloor m-(i+1) \rfloor} = (f)_{i+1} \times z^{\lfloor i+1 \rfloor} \times (\dot{1} + z)^{\lfloor m-(i+1) \rfloor} + g_i \times (\dot{1} + z)^{\lfloor m-i \rfloor}$$

which allows us to define g_i solely in terms of f:

$$g_i = \sum_k (f)_k \times z^{\lfloor k \rfloor} \times (\dot{1} + z)^{\lfloor i-k \rfloor} \times [k \leqslant i]$$

The factor $(\dot{1} + z)^{\lfloor i-k \rfloor}$ indicates that binomial coefficients are lurking behind the scenes and, indeed,

$$(g_i)_j = \sum_k (f)_k * \binom{i-k}{i-j} * [k \leqslant i] = \sum_k (f)_k * \binom{i-k}{j-k}$$

which is Equation (29) modulo naming of variables. Fortunately, we do not need the pointwise characterisation in what follows. The central idea is to view the triangle transformation as a *function* that maps the input column f to the output column g_n:

$$\nabla_n\ f \quad = \quad \sum_k (f)_k \times z^{\lfloor k \rfloor} \times (\dot{1} + z)^{\lfloor n-k \rfloor} \times [k \leqslant n]$$

Moessner's process can be formalised by repeatedly applying the triangle transformation to the power series $\dot{1}$; the coefficients at position n then form the sequence of nth powers.

$$\mathit{moessner}\ n = \langle (f)_n \mid f \leftarrow \dot{1}, \nabla_n\ f \ldots \rangle$$

Turning to the proof of Moessner's Theorem, let us first study the stream operator ∇_n in more detail. The formula for ∇_n looks a bit daunting, but, as we shall see, the triangle transformation enjoys a number of pleasant properties. The first non-trivial instance gives

$$\begin{aligned} \nabla_1\ \dot{1} &= \dot{1} + z \\ \nabla_1\ z &= z \end{aligned}$$

Table 6. *Proof that ∇ distributes over $\times$*

$$
\begin{aligned}
& \nabla_{m+n}\,(f \times g) \\
= \quad & \{\ \text{definition of } \nabla\ \} \\
& \sum_{k} (f \times g)_k \times z^{\lfloor k \rfloor} \times (\dot{1} + z)^{\lfloor m+n-k \rfloor} \times [k \leqslant m + n] \\
= \quad & \{\ \text{pointwise characterisation of } \times\ \} \\
& \sum_{k} (\sum_{i} (f)_i \times (g)_{k-i}) \times z^{\lfloor k \rfloor} \times (\dot{1} + z)^{\lfloor m+n-k \rfloor} \times [k \leqslant m + n] \\
= \quad & \{\ \text{distributive law and interchanging the order of summation}\ \} \\
& \sum_{i,k} (f)_i \times (g)_{k-i} \times z^{\lfloor k \rfloor} \times (\dot{1} + z)^{\lfloor m+n-k \rfloor} \times [k \leqslant m + n] \\
= \quad & \{\ \text{replace } k \text{ by } i + j\ \} \\
& \sum_{i,j} (f)_i \times (g)_j \times z^{\lfloor i+j \rfloor} \times (\dot{1} + z)^{\lfloor m+n-(i+j) \rfloor} \times [i + j \leqslant m + n] \\
= \quad & \{\ \text{assumption: } (f)_i = 0 \text{ for } i > m \text{ and } (g)_j = 0 \text{ for } j > n\ \} \\
& \sum_{i,j} (f)_i \times (g)_j \times z^{\lfloor i+j \rfloor} \times (\dot{1} + z)^{\lfloor m-i+n-j \rfloor} \times [i \leqslant m \wedge j \leqslant n] \\
= \quad & \{\ \text{laws of exponentials and } [b \wedge c] = [b] \times [c]\ \} \\
& \sum_{i,j} (f)_i \times z^{\lfloor i \rfloor} \times (\dot{1} + z)^{\lfloor m-i \rfloor} \times [i \leqslant m] \times (g)_j \times z^{\lfloor j \rfloor} \times (\dot{1} + z)^{\lfloor n-j \rfloor} \times [j \leqslant n] \\
= \quad & \{\ \text{distributive law}\ \} \\
& (\sum_{i} (f)_i \times z^{\lfloor i \rfloor} \times (\dot{1} + z)^{\lfloor m-i \rfloor} \times [i \leqslant m]) \times (\sum_{j} (g)_j \times z^{\lfloor j \rfloor} \times (\dot{1} + z)^{\lfloor n-j \rfloor} \times [j \leqslant n]) \\
= \quad & \{\ \text{definition of } \nabla\ \} \\
& \nabla_m\ f \times \nabla_n\ g
\end{aligned}
$$

Furthermore, it is not hard to see that the operator is a linear transformation:

$$
\begin{aligned}
\nabla_n\,(\mathit{const}\ k \times f) &= \mathit{const}\ k \times \nabla_n\ f \\
\nabla_n\,(f + g) &= \nabla_n\ f + \nabla_n\ g
\end{aligned}
$$

For instance, using *const* 2 rather than *const* 1 as the initial seed yields the natural powers doubled. Perhaps surprisingly, the triangle transformation also distributes over the convolution product: if $(f)_i = 0$ for $i > m$ and $(g)_j = 0$ for $j > n$, then

$$\nabla_{m+n}\,(f \times g) = \nabla_m\ f \times \nabla_n\ g$$

The side condition requires that f is a polynomial of degree at most m and, likewise, that g has degree at most n. Table 6 displays the proof of this property. An immediate consequence is $\nabla_n\ \dot{1} = \nabla_n\,(\dot{1}^{\lfloor n \rfloor}) = (\nabla_1\ \dot{1})^{\lfloor n \rfloor} = (\dot{1} + z)^{\lfloor n \rfloor}$, or, more generally,

$$\nabla_n\,((\dot{1} + \mathit{const}\ i \times z)^{\lfloor n \rfloor}) = (\nabla_1\ (\dot{1} + \mathit{const}\ i \times z))^{\lfloor n \rfloor} = (\dot{1} + \mathit{const}\ (i+1) \times z)^{\lfloor n \rfloor} \quad (33)$$

For instance, the cube-generating process yields

$\dot{1}$				$(\dot{1}+z)^{\lfloor 3 \rfloor}$				$(\dot{1}+\dot{2}\times z)^{\lfloor 3 \rfloor}$				$(\dot{1}+\dot{3}\times z)^{\lfloor 3 \rfloor}$	…
1	1	1	1	1	1	1	1	1	1	1	1	1	…
0		1	2	3		4	5	6		7	8	9	…
0			1	3			7	12			19	27	…
0				1				8				27	…

In general, after i triangle transformations we obtain the sequence $(\dot{1}+const\ i\times z)^{\lfloor n \rfloor}$:

$$iterate\ \nabla_n\ \dot{1} \quad = \quad \langle (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle$$

Here is a simple unique fixed-point proof of this fact. We show that the right-hand side satisfies the equation $x = \dot{1} \prec map\ \nabla_n\ x$.

$$\begin{aligned}
& \langle (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle \\
= \quad & \{ \text{definition of } nat \} \\
& (\dot{1} + const\ 0 \times z)^{\lfloor n \rfloor} \prec \langle (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat + 1 \rangle \\
= \quad & \{ \text{arithmetic and functor law (4)} \} \\
& \dot{1} \prec \langle (\dot{1} + const\ (i+1) \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle \\
= \quad & \{ (33) \} \\
& \dot{1} \prec \langle \nabla_n\ (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle \\
= \quad & \{ \text{functor law (4)} \} \\
& \dot{1} \prec map\ \nabla_n\ \langle (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle
\end{aligned}$$

We are now in a position to re-prove Moessner's Theorem. Since the kth coefficient of $(\dot{1} + const\ i \times z)^{\lfloor n \rfloor}$ is $\binom{n}{k} * i^k$, see Table 5, we have

$$\begin{aligned}
& moessner\ n \\
= \quad & \{ \text{definition of } moessner \} \\
& \langle (f)_n \mid f \leftarrow iterate\ \nabla_n\ \dot{1} \rangle \\
= \quad & \{ \text{see above} \} \\
& \langle (f)_n \mid f \leftarrow \langle (\dot{1} + const\ i \times z)^{\lfloor n \rfloor} \mid i \leftarrow nat \rangle \rangle \\
= \quad & \{ \text{functor law (4)} \} \\
& \langle ((\dot{1} + const\ i \times z)^{\lfloor n \rfloor})_n \mid i \leftarrow nat \rangle \\
= \quad & \{ ((\dot{1} + const\ i \times z)^{\lfloor n \rfloor})_n = \tbinom{n}{n} * i^n = i^n \} \\
& \langle i^n \mid i \leftarrow nat \rangle \\
= \quad & \{ \text{extensionality (16)} \} \\
& nat^n
\end{aligned}$$

Voilà. The introduction of the stream operator ∇_n illustrates the important idea of avoiding unnecessary detail. The triangle transformation hides away the generating functions of the intermediate columns, which are not immediately needed to establish the overall result.

5.4 Solving recurrences

Generating functions are jolly useful to solve numeric recurrences, that is, to turn a recursive stream definition into a non-recursive one. Let us illustrate the approach using a simple recurrence, the closed form of which we can determine with an informed guess.

$$hanoi = 0 \prec hanoi + 1 + hanoi$$

The stream *hanoi* yields the number of moves to solve the Tower of Hanoï problem with *nat* discs. It is not hard to see that $hanoi = 2^{nat} - 1$.

As a first step, we express *hanoi* in terms of $\times$ and friends.

$$\begin{aligned}
& hanoi = 0 \prec hanoi + 1 + hanoi \\
= \quad & \{ \text{ Horner's rule (32) } \} \\
& hanoi = z \times (hanoi + 1 + hanoi) \\
= \quad & \{ \text{ Table 5 } \} \\
& hanoi = z \times (hanoi + (const\ 1 \div (const\ 1 - z)) + hanoi) \\
= \quad & \{ \text{ arithmetic } \} \\
& hanoi = z \div ((const\ 1 - z) \times (const\ 1 - const\ 2 \times z))
\end{aligned}$$

To reiterate, we have to be careful not to confuse $const\ n$ and $\times$ with $repeat\ n$ and $*$. In particular, recall that the numeral 1 is shorthand for $repeat\ 1$, so its generating function is $const\ 1 \div (const\ 1 - z)$, *not* $const\ 1$.

For the second step, we have to turn the right-hand side of the resulting equation into a generating function or a sum of generating functions whose coefficients we know. The following algebraic identity points us into the right direction ($\alpha \neq \beta$).

$$\frac{x}{(1-\alpha x)(1-\beta x)} = \frac{1}{\alpha - \beta}\left(\frac{1}{1-\alpha x} - \frac{1}{1-\beta x}\right) \tag{34}$$

Inspecting Table 5 we realise that we know the stream expression for the right-hand side:

$$repeat\ (1 \mathbin{/} \alpha - \beta) * ((repeat\ \alpha)^{nat} - (repeat\ \beta)^{nat}).$$

For our running example, $\alpha = 1$ and $\beta = 2$. Substituting the values and applying some algebraic simplifications, we obtain the expected result.

$$hanoi = 2^{nat} - 1$$

A noteworthy feature of the derivation is that it stays entirely within the world of streams.

Now, let us try to find a closed form for our all-time favourite, the Fibonacci sequence. The first step, determining the generating function, is probably routine by now.

$$
\begin{aligned}
& \mathit{fib} = 0 \prec (1 \prec \mathit{fib}) + \mathit{fib} \\
= \quad & \{ \text{ Horner's rule (32) } \} \\
& \mathit{fib} = z \times ((\mathit{const}\ 1 + z \times \mathit{fib}) + \mathit{fib}) \\
= \quad & \{ \text{ arithmetic } \} \\
& \mathit{fib} = z \div (\mathit{const}\ 1 - z - z^{\lfloor 2 \rfloor})
\end{aligned}
$$

To be able to apply (34), we have to transform $1-z-z^2$ 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: $z^2 - z - 1 = (z-\phi)(z-\hat{\phi})$. Consequently, $1-z-z^2 = (1-\phi z)(1-\hat{\phi} z)$. Since furthermore $\phi - \hat{\phi} = \sqrt{5}$, we have inferred that

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

We can reformulate the result in more familiar terms

$$\mathscr{F}_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

but this is only cosmetics.

Many recurrences can be solved along those lines. In general, the main effort goes into finding the *partial fraction expansion* of the rational function obtained as the result of the first step. Very briefly, every rational function (which is defined at 0) can be expressed as the sum of a polynomial and a linear combination of functions of the form

$$\mathit{const}\ 1 \div (\mathit{const}\ 1 - \mathit{const}\ a \times z)^{\lfloor k+1 \rfloor} = \binom{\mathit{nat}+k}{k} * (\mathit{repeat}\ a)^{\mathit{nat}} \tag{35}$$

For the details, we refer the interested reader to Graham *et al.* (1994, Section 7.3). As a quick example, consider the recursion equation $x = 0 \prec 2 * x + 2^{\mathit{nat}}$, which captures the puzzle posed in Section 4.2. Its generating function is $z \div (\mathit{const}\ 1 - \mathit{const}\ 2 \times z)^{\lfloor 2 \rfloor}$, which by (35) is equal to

$$z \times \left(\binom{\mathit{nat}+1}{1} * 2^{\mathit{nat}} \right) = 0 \prec (\mathit{nat}+1) * 2^{\mathit{nat}} = \mathit{nat} * 2^{\mathit{nat}-1}$$

5.5 Counting

In Section 3.2.1 we have turned a context-free grammar (in Backus-Naur Form) into a system of stream equations—for each non-terminal X we introduced a stream x whose nth element specified the number of words of length n produced by X. Using generating functions we can nicely mechanise the translation: the grammar equation

$X = E$ is turned into the stream equation $x = e$ whose right-hand side is formed according to

BNF expression			stream expression
	$\emptyset$	$\mathit{const}\ 0$	
	ϵ	$\mathit{const}\ 1$	
	a	z	
	X	x	
	$E_1 \mid E_2$	$e_1 + e_2$	
	$E_1\ E_2$	$e_1 \times e_2$	

Each terminal symbol a is replaced by z—we are only interested in the length of the words—each non-terminal symbol X is replaced by the corresponding stream variable x. Alternation corresponds to lifted addition and concatenation to the convolution product. To illustrate, here is the original grammar

$$\begin{aligned} \mathit{Nao} &= \epsilon \mid 0\ \mathit{Nao} \mid 1\ \mathit{Nap} \\ \mathit{Nap} &= \epsilon \mid 0\ \mathit{Nao} \end{aligned}$$

and here is the corresponding system of stream equations

$$\begin{aligned} \mathit{nao} &= \mathit{const}\ 1 + z \times \mathit{nao} + z \times \mathit{nap} \\ \mathit{nap} &= \mathit{const}\ 1 + z \times \mathit{nao} \end{aligned}$$

Horner's rule (32) immediately gives us the stream equations of Section 3.2.1.

There is one caveat, however. The translation tacitly assumes that $E_1 \mid E_2$ is a *disjoint* union. Otherwise, we do not count the number of words of some length, but rather the number of their parse trees. Fortunately, in our example, the languages generated by ϵ, $0\ \mathit{Nao}$ and $1\ \mathit{Nap}$ are clearly disjoint.

Essentially the same technique can also be applied to parametric datatypes, so-called *container types*, to count the number of containers of some given size. Consider as an example the inductive datatype of binary trees.

$$\mathbf{data}\ \mathit{Tree}\ \alpha = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha, \alpha, \mathit{Tree}\ \alpha)$$

There is one tree of size 0, one tree of size 1, two trees of size 3, five trees of size 4, and so forth. In general, the number of trees of a given size satisfies

$$\mathit{tree} = \mathit{const}\ 1 + \mathit{tree} \times z \times \mathit{tree}$$

Using Horner's rule (32) this specification translates into the executable stream equation

$$\mathit{tree} = 1 \prec \mathit{tree} \times \mathit{tree}$$

The correspondence between datatype declarations and stream equations can be seen more clearly if we write the former in a point-free or functorial style:

$$\mathit{Tree} = K\ 1 \dotplus \mathit{Tree} \dot{\times} \mathit{Id} \dot{\times} \mathit{Tree}$$

where K is the constant type constructor, $K\ \alpha\ \beta = \alpha$, Id is the identity, $\mathit{Id}\ \alpha = \alpha$, and $\dotplus$ and $\dot{\times}$ are lifted sum and product, $(\iota \dotplus \kappa)\ \alpha = \mathit{Either}\ (\iota\ \alpha)\ (\kappa\ \alpha)$ and $(\iota \dot{\times} \kappa)\ \alpha = (\iota\ \alpha, \kappa\ \alpha)$. As a second example, the functional programmer's favourite

datatype, the inductive type of lists,

$$\textbf{data}\ List\ \alpha = Nil \mid Cons\ (\alpha, List\ \alpha)$$

enjoys the following point-free definition.

$$List = K\ 1 \dotplus Id \mathbin{\dot\times} List$$

The corresponding stream equation is then

$$list = const\ 1 + z \times list$$

Horner's rule implies $list = 1 \prec list$ and, consequently, $list = repeat\ 1$. For each size, there is exactly one list container, as expected.

Quite amazingly, the technique also works for so-called *non-regular* or *nested datatypes* (Bird & Meertens, 1998). Consider as an example the type of perfect trees (Hinze, 2000a).

$$\textbf{data}\ Perfect\ \alpha = Zero\ \alpha \mid Succ\ (Perfect\ (\alpha, \alpha))$$

A perfect tree of height 0 is a singleton; a perfect tree of height $h+1$ is a perfect tree of height h whose elements are pairs. Thus, a tree of the form $Succ^h\ (Zero\ t)$ has exactly 2^h elements. It will be interesting to see how this property translates into the world of streams. As a first step, we put the type definition into a point-free form:

$$Perfect = Id \dotplus Perfect \circ (Id \mathbin{\dot\times} Id)$$

where $\circ$ is type composition, $(\iota \circ \kappa)\ \alpha = \iota\ (\kappa\ \alpha)$. Now, which stream operator models composition of type constructors? A moment's reflection reveals that the corresponding operator is simply composition of generating functions. Loosely speaking, the composition $s \circ t$ replaces the formal variable z in s by t, for instance, $(z \times z) \circ t = t \times t$. Let us postpone the implementation of composition and first return to our application. The number of perfect trees of some size then satisfies the stream equation

$$perfect = z + perfect \circ (z \times z) \tag{36}$$

As an intermediate summary, to count containers we transform the functor equation $X = E$ into the stream equation $x = e$ whose right-hand side is formed according to

functor expression			stream expression
	$K\ 0$	$const\ 0$	
	$K\ 1$	$const\ 1$	
	Id	z	
	X	x	
	$F_1 \dotplus F_2$	$e_1 + e_2$	
	$F_1 \mathbin{\dot\times} F_2$	$e_1 \times e_2$	
	$F_1 \circ F_2$	$e_1 \circ e_2$	

This time there is no caveat, since $\dotplus$ forms a disjoint union by definition. Of course, there is no guarantee that the resulting system of stream equations has a unique solution or even a solution at all: $x = x$ has infinitely many solutions, $x = const\ 1 + z \times x$ has a unique solution, and $x = z + x$ has no solution.

Table 7. *Properties of composition*

$z \circ s$	$=$	s	$(s \times t) \circ u$	$=$	$s \circ u \times t \circ u$
$s \circ z$	$=$	s	$(s \div t) \circ u$	$=$	$s \circ u \div t \circ u$
$(s \circ t) \circ u$	$=$	$s \circ (t \circ u)$	$recip\ s \circ u$	$=$	$recip\ (s \circ u)$
$const\ a \circ u$	$=$	$const\ a$	$s^{\lfloor n \rfloor} \circ u$	$=$	$(s \circ u)^{\lfloor n \rfloor}$
$(s + t) \circ u$	$=$	$s \circ u + t \circ u$	$s \curlyvee t$	$=$	$(s \circ z^{\lfloor 2 \rfloor}) + z \times (t \circ z^{\lfloor 2 \rfloor})$

Now, let us derive an implementation of composition assuming the laws listed in Table 7 ($\circ$ binds more tightly than $\times$).

$$
\begin{aligned}
& s \circ t \\
= \ & \{ \text{Horner's rule (32)} \} \\
& (const\ (head\ s) + z \times tail\ s) \circ t) \\
= \ & \{ \circ \text{ right-distributes over } + \text{ and } \times \} \\
& const\ (head\ s) \circ t + z \circ t \times tail\ s \circ t \\
= \ & \{ const\ a \circ u = const\ a \text{ and } z \circ u = u \} \\
& const\ (head\ s) + t \times tail\ s \circ t \\
= \ & \{ \textbf{assumption: } head\ t = 0 \} \\
& const\ (head\ s) + z \times tail\ t \times tail\ s \circ t \\
= \ & \{ \text{Horner's rule (32)} \} \\
& head\ s \prec tail\ t \times tail\ s \circ t
\end{aligned}
$$

In order to make progress, we have to assume that $head\ t = 0$. In general, the first coefficient of $s \circ t$ is given by an infinite sum, which possibly exists, but which we cannot compute mechanically in Haskell. Like *recip*, composition is a partial operation.

$$
\begin{aligned}
&(\circ) \ :: (Num\ \alpha) \Rightarrow Stream\ \alpha \rightarrow Stream\ \alpha \rightarrow Stream\ \alpha \\
&s \circ t = head\ s \prec tail\ t \times tail\ s \circ t
\end{aligned}
$$

Composition is associative with z as its neutral element; and it right-distributes over the arithmetic operations ($+$, $\times$, $\div$). Furthermore, there is an intriguing cross-connection to interleaving. Consider the generating function $s \circ z^{\lfloor 2 \rfloor}$; its even coefficients are those of s, its odd coefficients are zero. Consequently, we can express interleaving using composition:

$$s \curlyvee t = (s \circ z^{\lfloor 2 \rfloor}) + z \times (s \circ z^{\lfloor 2 \rfloor}).$$

Table 7 summarises our findings.

We can use the properties to turn the specification of *perfect* into an executable form. First of all, the stream equation is ambiguous as it does not determine the head of *perfect*. Since there is no empty perfect tree, we set $head\ perfect = 0$. For

the tail, we reason

$$\begin{aligned}
& \mathit{tail\ perfect} \\
= \;& \{ \text{ specification of } \mathit{perfect}\ (36) \} \\
& \mathit{tail}\ (z + \mathit{perfect} \circ z^{\lfloor 2 \rfloor}) \\
= \;& \{ \text{ definition of } z \text{ and } \circ \} \\
& \mathit{const}\ 1 + z \times (\mathit{tail\ perfect} \circ z^{\lfloor 2 \rfloor}) \\
= \;& \{ s \circ z^{\lfloor 2 \rfloor} = s \curlyvee 0 \text{ (Table 7) } \} \\
& \mathit{const}\ 1 + z \times (\mathit{tail\ perfect} \curlyvee 0) \\
= \;& \{ \text{ Horner's rule (32) } \} \\
& 1 \prec \mathit{tail\ perfect} \curlyvee 0
\end{aligned}$$

Thus, *tail perfect* is essentially *pot*—if we identify 0 with *false* and 1 with *true*.

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

It is high time to see the definitions in action (*A*000012, *A*000108, not listed).

≫ *list*
⟨1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …⟩
≫ *tree*
⟨1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, …⟩
≫ *perfect*
⟨0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, …⟩

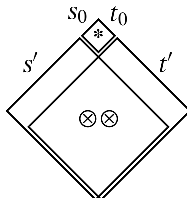
The stream *tree* captures a well-known combinatorial sequence, the *Catalan* or *Segner* numbers. Can you derive a closed form?

5.6 Exponential generating functions

The implementation of the convolution product exhibits an unpleasant asymmetry. There is a related operator, called *shuffle product*, whose definition is perfectly symmetric.

$$s \otimes t = \mathit{head}\ s * \mathit{head}\ t \prec \mathit{tail}\ s \otimes t + s \otimes \mathit{tail}\ t$$

The two-dimensional illustration of the corecursion scheme below shows that the shuffle products in the tail overlap, in a way reminiscent of the recurrence underlying Pascal's triangle (the picture is rotated 45° degrees clockwise to emphasise the correspondence).



For instance, the term $s_1 * t_1$ appears once in $tail\ s \otimes t$ and once in $s \otimes tail\ t$, and hence twice in $s \otimes t$.

The shuffle product implements the product of two *exponential generating functions* (egfs). An egf is similar to a gf except that the nth coefficient is additionally divided by n factorial: $a_0 + a_1 z + a_2 z^2/2! + a_3 z^3/3! + \cdots$. Like the convolution product, the shuffle product is associative, commutative, it distributes over sums, $const\ 1 \otimes s = s$, and furthermore $(c * s) \otimes t = c * (s \otimes t) = s \otimes (c * t)$.

To turn a stream representing a gf into a stream representing an egf, we simply have to multiply the former by fac.

$$\Lambda\ s = s * fac$$

The two products are then related by

$$\Lambda\ (s \times t) = \Lambda\ s \otimes \Lambda\ t$$

The formula enables the derivation of the following pointwise characterisation of the shuffle product, which links in with the earlier observation that the corecursion scheme of $\otimes$ is related to Pascal's triangle. Since $(i + j)\ ! /(i\ ! * j!) = \binom{i+j}{i}$, we have

$$(s \otimes t)_n \quad = \quad \sum_{i,j} \binom{n}{i} * (s)_i * (t)_j * [i + j = n]$$

So, the shuffle product is similar to the convolution product, except that each point-level product is multiplied by a binomial coefficient (which suggests a more efficient implementation of $\otimes$). Because of that, there is a revealing cross-connection to binomial transforms:

$$\mathscr{B}_n\ s = n^{nat} \otimes s$$

Indeed, if we unfold $n^{nat} \otimes s$, we obtain the recursion equation of $\mathscr{B}_n$. In particular, $\mathscr{B}\ s = 1 \otimes s$. All the properties of the binomial transform listed in Section 4.2 follow directly from this correspondence.

First of all, here is the stream version of the Binomial Theorem.

$$m^{nat} \otimes n^{nat} = (m + n)^{nat}$$

Egfs are well behaved, when their coefficients are geometric progressions, hence the name. The unique fixed-point proof of the above property is remarkable, as it does not mention binomial coefficients at all.

$$\begin{aligned}
& m^{nat} \otimes n^{nat} \\
= \; & \{ \text{ definition of } \otimes \} \\
& m^0 * n^0 \prec m^{nat+1} \otimes n^{nat} + m^{nat} \otimes n^{nat+1} \\
= \; & \{ \text{ arithmetic and } (c * s) \otimes t = c * (s \otimes t) = s \otimes (c * t) \} \\
& 1 \prec (m + n) * (m^{nat} \otimes n^{nat})
\end{aligned}$$

Back to the correspondence between shuffle product and binomial transforms: Proving, for instance, that $\mathscr{B}$ and $\mathscr{B}^{(-1)}$ are mutually inverse is now a breeze.

$$\mathscr{B}\ (\mathscr{B}^{(-1)}\ s) = 1^{nat} \otimes (-1)^{nat} \otimes s = 0^{nat} \otimes s = const\ 1 \otimes s = s$$

The other laws can be shown using similar arguments.

The moral of the story is that the *right* generalisation often makes our life easier.

6 Related work

As mentioned in the introduction, the two major sources of inspiration were Rutten's work on stream calculus (2003, 2005) and the textbook 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 lifted addition

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

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 shows the existence and uniqueness of solutions for a particular system of behavioural differential equations (2003, Theorem 3.1). [The conference version of this paper rephrases the proof using type classes and generalised algebraic data types (Hinze, 2008).] Bartels proves uniqueness for the general case in a categorical setting (2003).

Interestingly, 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; McIlroy, 2001). Both Karczmarczuk and McIlroy mention the proof technique of unique fixed points in passing by: Karczmarczuk sketches a proof of $\mathit{iterate}\ f \cdot f = \mathit{map}\ f \cdot \mathit{iterate}\ f$ and McIlroy shows $1/e^x = e^{-x}$. As an aside, both use lazy lists to represent streams, resulting in additional code to cover the empty list.

Various proof methods for corecursive programs are discussed by Gibbons and Hutton (2005). The technique of unique fixed points is not among them.[10] Unique fixed-point proofs are closely related to the principle of *guarded induction* (Coquand, 1994), which goes back to the work on process algebra (Milner 1989). 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,

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

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

[10] 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`.

proof transformer using guarded equations. Indeed, the central idea underlying guarded induction is to express proofs as lazy functional programs.

Recently, Niqui and Rutten introduced various operations for partitioning, projecting and merging streams (2010). While the operators generalise *even*, *odd* and $\curlyvee$, it is not clear under which conditions they can be used in other definitions—as we have pointed out, the equation $x = 0 \prec \mathit{even}\ x$, for instance, has infinitely many solutions.

The use of different structural views—idioms, tabulations and final coalgebras—to organise operations, laws and proofs appears to be original. Categorically, idioms are *lax monoidal functors* (Mac Lane, 1998) with strength. Programmatically, idioms arose as an interface for parsing combinators (Röjemo, 1995). McBride & Paterson (2008) introduced the notion to a wider audience. Idioms capture the notion of lifting. The type of streams is a very special idiom in that a lifted operator inherits the properties of the base-level operator. The recent paper by the author (2010) abstracts away from streams and answers the questions: 'What base-level identities can be lifted through any idiom?' and 'Which idioms satisfy every lifted base-level identity?' I previously discussed memo tables in a datatype-generic setting (2000b, 2010b). The observation that the memo table of an inductive datatype is given by a coinductive datatype is due to Altenkirch (2001).

This paper features two different proofs of Moessner's Theorem: the first is loosely modelled after a proof by Salié (1952), the second is based on a proof by Paasche (1952). A third alternative proof can be found in a recent paper of mine (to appear a), which generalises the number-theoretic operators Σ and $\times$ to polymorphic, higher-order combinators—scans and convolutions. In fact, the paper proves Paasche's generalisation of Moessner's Theorem (1952).

The proof techniques presented in this paper can be readily generalised to other coinductive datatypes. Silva and Rutten adopt coinductive definition and proof principles to infinite binary trees (2010). Independently, the author adopted the unique fixed-point techniques to the same structures, applying the techniques to the problem of enumerating the rationals (Hinze, 2009).

7 Conclusion

I hope you enjoyed the journey. Lazy functional programming has proven its worth: with a couple of one-liners we have 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.

Acknowledgements

Thanks are due to Nils Anders Danielsson, Jeremy Gibbons, Tom Harper, Daniel W. H. James, and the anonymous referees of ICFP 2008 for improving my English and for pointing out several typos. I owe a special debt of gratitude to the anonymous referees of this special issue, who went over and above the call of duty. Their

suggestions greatly helped to improve the quality of the paper. Roland Backhouse provided intensive feedback—19 pages of TEX'ed comments. He pointed out that the approach is more general than I initially thought and advocated the consequential use of lifting and overloading. He also challenged me to derive implementation from specifications, in order to close the gap between the verbal descriptions and their incarnations in stream calculus. Thank you Roland.

References

Altenkirch, T. (2001) Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, Lecture Notes in Computer Science, vol. 2044. Springer-Verlag, pp. 62–78.

Backhouse, R. (2001) *Galois Connections and Fixed Point Calculus*. Lecture Notes.

Backhouse, R. & Fokkinga, M. (2001) The associativity of equivalence and the towers of Hanoi problem, *Inf. Process. Lett.,* 77: 71–76.

Backhouse, R. & Michaelis, D. (2005) *A Calculational Presentation of Impartial Two-Person Games*. Abstract presented at RelMiCS8.

Bartels, F. (2003) Generalised coinduction, *Math. Struct. Comp. Sci.,* 13: 321–348.

Bird, R. (1998) *Introduction to Functional Programming using Haskell*, 2nd ed. Prentice Hall, Europe.

Bird, R. & Meertens, L. (1998) Nested datatypes. In *Proceedings of the Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, Jeuring, J. (ed), Lecture Notes in Computer Science, vol. 1422. Springer-Verlag, pp. 52–67.

Bird, R. (1987) An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design, Marktoberdorf, Germany*, Broy, M. (ed), Springer-Verlag, pp. 5–42.

Coquand, T. (1994) Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, Barendregt, H. & Nipkow, T. (eds), Lecture Notes in Computer Science, vol. 806. Springer-Verlag, pp. 62–78.

Danvy, O. (1999) An extensional characterization of lambda-lifting and lambda-dropping. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Middeldorp, A. & Sato, T. (eds), Lecture Notes in Computer Science, vol. 1722. Springer-Verlag, pp. 241–250.

Fokkinga, M. M. (1992) *Law and Order in Algorithmics*. PhD thesis, University of Twente.

Gibbons, J. & Hutton, G. (2005) Proof methods for corecursive programs. *Fundam. Inform.* (*Special Issue on Program Transformation*) **66**(4): 353–366.

Gill, A. & Hutton, G. (2009) The worker/wrapper transformation, *J. Funct. Program.,* **19** (2): 227–251.

Graham, R. L., Knuth, D. E. & Patashnik, O. (1994) *Concrete Mathematics*, 2nd edn. Addison-Wesley Publishing Company.

Hinze, R. (2000a) Functional Pearl: Perfect trees and bit-reversal permutations, *J. Funct. Program.,* **10**(3): 305–317.

Hinze, R. (2000b) Memo functions, polytypically!. In *Proceedings of the 2nd Workshop on Generic Programming*. Jeuring, J. (ed), *Ponte de Lima, Portugal*, pp. 17–32. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

Hinze, R. (2008) Functional Pearl: Streams and unique fixed points. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* Thiemann, P. (ed), ACM Press, pp. 189–200.

Hinze, R. (2009) Functional Pearl: The Bird tree, *J. Funct. Program.,* **19** (5): 491–508.

Hinze, R. (to appear a) Scans and convolutions—a calculational proof of Moessner's Theorem. In *Post-proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008), University of Hertfordshire* (*UK, September 10–12, 2008*). Scholz, S.-B. (ed), Lecture Notes in Computer Science, vol. 5836. Springer-Verlag.

Hinze, R. (2010) Lifting operators and laws. Available at: `http://www.comlab.ox.ac.uk/ralf.hinze/Lifting.pdf`.

Hinze, R. (to appear b) Type fusion. In *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, Pavlovic, D. & Johnson, M. (eds), Lecture Notes in Computer Science, vol. 6486. Springer-Verlag.

Hinze, R. & Löh, A. (2008) Guide2lhs2TeX (for version 1.13). Available at: `http://people.cs.uu.nl/andres/lhs2tex/`.

Karczmarczuk, J. (1997) Generating power of lazy semantics, *Theor. Comput. Sci.,* (*Special volume on computer algebra*), **187** (1–2): 203–219.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd edn. Addison-Wesley Publishing Company.

Mac Lane, S. (1998) *Categories for the Working Mathematician*, 2nd edn. Graduate Texts in Mathematics. Springer-Verlag.

McBride, C. & Paterson, R. (2008) Functional Pearl: Applicative programming with effects, *J. Funct. Program.* **18** (1): 1–13.

McIlroy, M. D. (1999) Power series, power serious, *J. Funct. Program.,* **3** (9): 325–337.

McIlroy, M. D. (2001) The music of streams, *Inf. Process. Lett.*, **77** (2–4): 189–195.

Milner, R. (1989) *Communication and Concurrency*. International Series in Computer Science. Prentice Hall International.

Moessner, A. (1951) Eine Bemerkung über die Potenzen der natürlichen Zahlen. In*Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften.* Mathematisch-naturwissenschaftliche Klasse, *Nr. 3* March: 29.

Niqui, M. & Rutten, J. (2010) Sampling, splitting and merging in coinductive stream calculus. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC '10)*, Bolduc, C., Desharnais, J. & Ktari, B. (eds), Lecture Notes in Computer Science, vol. 6120. Springer-Verlag, pp. 310–330.

Paasche, I. (1952) Ein neuer Beweis des Moessnerschen Satzes. In *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften.* Mathematisch-naturwissenschaftliche Klasse, *Nr. 1* February:1–5.

Perron, O. (1951) Beweis des Moessnerschen Satzes. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften.* Mathematisch-naturwissenschaftliche Klasse*, Nr. 4* May: 31–34.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries.* Cambridge University Press.

Rutten, J. (2003) Fundamental study: Behavioural differential equations: A coinductive calculus of streams, automata, and power series, *Theor. Comput. Sci.,* **308**: 1–53.

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

Röjemo, N. (1995) *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* PhD thesis, Chalmers University of Technology.

Salié, H. (1952) Bemerkung zu einem Satz von Moessner. In *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften.* Mathematisch-naturwissenschaftliche Klasse*, Nr. 2* February: 7–11.

Silva, A. & Rutten, J. (2010) A coinductive calculus of binary trees, *Inf. Comput.*, **208**: 578–593.

Sloane, N. J. A. (2009) The on-line encyclopedia of integer sequences [online]. Available at: `http://www.research.att.com/~njas/sequences/`.

Stewart, I. (1999) *The Magical Maze: Seeing the World Through Mathematical Eyes.* Wiley. ISBN 0471350656.