Scans and Convolutions
A Calculational Proof of Moessner's Theorem

Ralf Hinze

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

Abstract. The paper introduces two corecursion schemes for stream-generating functions, scans and convolutions, and discusses their properties. As an application of the framework, a calculational proof of Paasche's generalisation of Moessner's intriguing theorem is presented.

1 Introduction

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

The second simplest instance of the process yields the squares by summing up the odd numbers. (Of course if we repeat the transformation 0 times, we obtain the 1st powers of the naturals.)

$$\begin{array}{llllllllll} 1 & \not{2} & 3 & \not{4} & 5 & \not{6} & 7 & \not{8} & \ldots \\ 1 & & 4 & & 9 & & 16 & & \ldots \end{array}$$

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

$$\begin{array}{llllllllllll} 1 & 2 & \not{3} & 4 & 5 & \not{6} & 7 & 8 & \not{9} & 10 & 11 & \not{12} & \ldots \\ 1 & \not{3} & & 7 & \not{12} & & 19 & \not{27} & & 37 & \not{48} & & \ldots \\ 1 & & & 8 & & & 27 & & & 64 & & & \ldots \end{array}$$

The second sequence is probably unfamiliar—the numbers are the "three-quarter squares" ($A077043$[1])—but the final sequence is the desired sequence of cubes.

Actually, it is not surprising that we get *near* the kth powers by repeated summations. If we omit the deletions, we obtain the columns of Pascal's triangle, the *binomial coefficients*, which are related to the *falling factorial powers* [3].

$$\begin{array}{lllllll} 1 & 2 & 3 & 4 & 5 & \ldots \\ 1 & 3 & 6 & 10 & 15 & \ldots \\ 1 & 4 & 10 & 20 & 35 & \ldots \\ 1 & 5 & 15 & 35 & 70 & \ldots \end{array}$$

[1] Most sequences defined in this paper are recorded in Sloane's On-Line Encyclopedia of Integer Sequences [2]. Keys of the form *Annnnnn* refer to entries in that database.

It *is* surprising that the additional deletion step is exactly what is needed to generate the kth powers. However, the magic does not stop here. In the original scheme we deleted numbers at regular intervals. What happens if we steadily increase the size of the intervals?

~~1~~	2	~~3~~	4	5	~~6~~	7	8	9	~~10~~	11	12	13	14	~~15~~	...
	~~2~~		6	~~11~~		18	26	~~35~~		46	58	71	~~85~~		...
			~~6~~			24	~~50~~			96	154	~~225~~			...
						~~24~~				120	~~274~~				...
										~~120~~					...

We obtain the factorials! The crossed-out numbers form the right sides of ◸-shaped triangles of increasing size. The numbers in the lower left corner make up the resulting sequence. What sequence do we obtain if we start off by deleting the squares or the cubes? The general scheme becomes visible if we rewrite the preceding example slightly. We began by deleting the numbers

$$\begin{aligned} 1 &= 1{*}1 \\ 3 &= 2{*}1 + 1{*}1 \\ 6 &= 3{*}1 + 2{*}1 + 1{*}1 \\ 10 &= 4{*}1 + 3{*}1 + 2{*}1 + 1{*}1 \end{aligned}$$

and obtained at the end of Moessner's process the sequence

$$\begin{aligned} 1 &= 1\hat{}1 \\ 2 &= 2\hat{}1 * 1\hat{}1 \\ 6 &= 3\hat{}1 * 2\hat{}1 * 1\hat{}1 \\ 24 &= 4\hat{}1 * 3\hat{}1 * 2\hat{}1 * 1\hat{}1 \ . \end{aligned}$$

Now if we delete, say, the numbers

$$\begin{aligned} 2 &= 1{*}2 \\ 11 &= 2{*}2 + 1{*}7 \\ 26 &= 3{*}2 + 2{*}7 + 1{*}6 \\ 46 &= 4{*}2 + 3{*}7 + 2{*}6 + 1{*}5 \ , \end{aligned}$$

where $2, 7, 6, 5$ is the prefix of some arbitrary sequence, we obtain the numbers

$$\begin{aligned} 1 &= 1\hat{}2 \\ 4 &= 2\hat{}2 * 1\hat{}7 \\ 1152 &= 3\hat{}2 * 2\hat{}7 * 1\hat{}6 \\ 2239488 &= 4\hat{}2 * 3\hat{}7 * 2\hat{}6 * 1\hat{}5 \ . \end{aligned}$$

Quite magically, factors have become exponents and sums have become products.

A purpose of this paper is to formalise Moessner's process, in fact, Paasche's generalisation of it [4], and to establish the relationship between the sequence of deleted positions and the resulting sequence of numbers. Of course, this is not the first published proof of Moessner's theorem: several number-theoretic arguments have appeared in the literature [5, 4, 6]. We approach the problem from a different angle: Moessner's process can be captured by a *corecursive*

program that operates on *streams*, which are infinite sequences. In this setting Moessner's theorem amounts to an equivalence of two corecursive functions.

A central message of the paper is that a programming-language perspective on concrete mathematics is not only feasible, but also beneficial: the resulting proofs have more structure and require little mathematical background. Last but not least, since the artifacts are executable programs, we can play with the construction, check conjectures, explore variations etc.

The overall development is based on a single proof method: the *principle of unique fixed points* [7]. Under some mild restrictions, recursion equations over a coinductive datatype possess unique solutions. Uniqueness can be exploited to prove that two elements of a codatatype are equal: if they satisfy the same recursion equation, then they are!

Along the way we introduce two corecursion schemes for stream-generating functions, *scans* and *convolutions*, which are interesting in their own right. Scans generalise the anti-difference or summation operator, which is one of the building blocks of *finite calculus*, and convolutions generalise the convolution product, which is heavily used in the theory of *generating functions*. Using the unique-fixed-point principle, we show that the two combinators satisfy various fusion and distributive laws, which generalise properties of summation and convolution product. These laws are then used to establish Moessner's theorem.

The rest of the paper is structured as follows. To keep the development self-contained, Section 2 provides an overview of streams and explains the main proof principle; the material is taken partly from "Streams and Unique Fixed Points" [7]. Sections 3 and 4 introduce scans and convolutions, respectively. Using this vocabulary, Section 5 then formalises Moessner's process and Section 6 proves it correct. Finally, Section 7 reviews related work and Section 8 concludes.

2 Streams

The type of streams, $\mathit{Stream}\,\alpha$, is like Haskell's list datatype $[\alpha]$, except that there is no base constructor so we cannot construct a finite stream. The *Stream* type is not an inductive type, but rather a *coinductive type*, whose semantics is given by a *final coalgebra* [8].[2]

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

Streams are constructed using $\prec$, which prepends an element to a stream. They are destructed using *head* and *tail*, which yield the first element and the rest of the stream, respectively.

[2] The definitions are given in the purely functional programming language Haskell [9]. Since Haskell has a CPO semantics, initial algebras and final coalgebras actually coincide [10].

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

2.1 Operations

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

$$\begin{array}{lcl} repeat & :: & \alpha \to Stream\, \alpha \\ repeat\, a & = & s \quad \textbf{where} \quad s \;=\; a \prec s \end{array}$$

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to (Stream\, \alpha \to Stream\, \beta) \\ map\, f\, s & = & f\, (head\, s) \prec map\, f\, (tail\, s) \end{array}$$

$$\begin{array}{lcl} zip & :: & (\alpha \to \beta \to \gamma) \to (Stream\, \alpha \to Stream\, \beta \to Stream\, \gamma) \\ zip\, f\, s\, t & = & f\, (head\, s)\, (head\, t) \prec zip\, f\, (tail\, s)\, (tail\, t) \end{array}$$

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

The definitions above show that *Stream* is a so-called *applicative functor* or *idiom* [11]: *pure* is *repeat* and idiomatic apply can be defined in terms of *zip*.

$$\begin{array}{lcl} pure & :: & \alpha \to Stream\, \alpha \\ pure & = & repeat \end{array}$$

$$\begin{array}{l} \textbf{infixl}\, 9\ \diamond \\ \begin{array}{lcl} (\diamond) & :: & Stream\, (\alpha \to \beta) \to (Stream\, \alpha \to Stream\, \beta) \\ s \diamond t & = & zip\, (\$)\, s\, t \end{array} \end{array}$$

Here, \$ denotes function application. Conversely, we can define the 'lifting operators' in terms of the idiomatic primitives: $repeat = pure$, $map\, f\, s = pure\, f \diamond s$ and $zip\, g\, s\, t = pure\, g \diamond s \diamond t$. We will freely switch between these two views.

Of course, we have to show that *pure* and $\diamond$ satisfy the *idiom laws*.

$$\begin{array}{lcll} pure\, id \diamond s & = & s & \text{(identity)} \\ pure\, (\cdot) \diamond s \diamond t \diamond u & = & s \diamond (t \diamond u) & \text{(composition)} \\ pure\, f \diamond pure\, a & = & pure\, (f\ a) & \text{(homomorphism)} \\ s \diamond pure\, a & = & pure\, (\$\, a) \diamond s & \text{(interchange)} \end{array}$$

We postpone the proofs until we have the prerequisites at hand.

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

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

This instance declaration allows us, in particular, to use integer constants as streams—in Haskell, unqualified 3 abbreviates $\mathit{fromInteger}\,(3 :: \mathit{Integer})$. Also note that since the arithmetic operations are defined point-wise, the familiar arithmetic laws also hold for streams.

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

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

Note that we use the convention that the identifier x' denotes the tail of x. Furthermore, note that $\prec$ binds less tightly than $+$. For instance, $0 \prec \mathit{nat} + 1$ is grouped $0 \prec (\mathit{nat} + 1)$.

Another useful function is *iterate*, which builds a stream by repeatedly applying a given function to a given value.

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

Thus, $\mathit{iterate}\,(+1)\,0$ is an alternative definition of the stream of naturals.

2.2 Definitions

Not every legal Haskell definition of type $\mathit{Stream}\,\tau$ 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.[3] In fact, they admit infinitely many solutions: every constant stream is a solution of the first equation and every stream is a solution of the second. This situation is undesirable from both a practical and a theoretical standpoint. Fortunately, it is not hard to restrict the *syntactic form* of equations so that they possess *unique solutions*. We insist that equations adhere to the following form:

$$x = h \prec t\ ,$$

where x is an identifier of type $\mathit{Stream}\,\tau$; h is an expression of type τ; and t is an expression of type $\mathit{Stream}\,\tau$ possibly referring to x, or some other stream identifier in the case of mutual recursion. However, neither h nor t may use $\mathit{head}\,x$ or $\mathit{tail}\,x$.

If x is a parametrised stream or a stream operator

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

then h or t may use $\mathit{head}\,x_i$ or $\mathit{tail}\,x_i$ provided x_i is of the right type. Furthermore, t may contain recursive calls to x, but we are not allowed to take the head or

[3] There is a slight mismatch between the theoretical framework of streams and the Haskell implementation of streams. Since products are lifted in Haskell, $\mathit{Stream}\,\tau$ additionally contains partial streams such as $\bot$, $a_0 \prec \bot$, $a_0 \prec a_1 \prec \bot$ and so forth. We simply ignore this extra complication here.

tail of a recursive call. There are no further restrictions regarding the arguments of a recursive call. For a formal account of these requirements, we refer the interested reader to "Streams and Unique Fixed Points" [7], which also contains a constructive proof that equations of this form indeed have unique solutions.

2.3 Proofs

Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are! If $s = \varphi\, s$ is an admissible equation in the sense of Section 2.2, we denote its unique solution by $\mathit{fix}\, \varphi$. (The equation implicitly defines a function in s. A solution of the equation is a fixed point of this function and vice versa.) The fact that the solution is unique is captured by the following universal property of fix.

$$\mathit{fix}\, \varphi \;=\; s \quad\Longleftrightarrow\quad \varphi\, s \;=\; s$$

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

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

$$\begin{array}{ll} & \mathit{pure}\, f \diamond \mathit{pure}\, a \\ = & \quad \{\ \text{definition of } \diamond\ \} \\ & (\mathit{head}\, (\mathit{pure}\, f))\, (\mathit{head}\, (\mathit{pure}\, a)) \;\prec\; \mathit{tail}\, (\mathit{pure}\, f) \diamond \mathit{tail}\, (\mathit{pure}\, a) \\ = & \quad \{\ \text{definition of } \mathit{pure}\ \} \\ & f\ a \;\prec\; \mathit{pure}\, f \diamond \mathit{pure}\, a \end{array}$$

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

So far we have been concerned with proofs about streams, however, the proof technique applies equally well to parametrised streams or stream operators! As an example, let us show the so-called *iterate* fusion law, which amounts to the free theorem of $(\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\, \alpha)$.

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

We show that both $\mathit{map}\, h \cdot \mathit{iterate}\, f_1$ and $\mathit{iterate}\, f_2 \cdot h$ satisfy the equation $x\ a = h\ a \prec x\, (f_1\ a)$. Since the equation has a unique solution, the law follows.

$$\begin{array}{ll} & (\mathit{map}\, h \cdot \mathit{iterate}\, f_1)\ a \\ = & \quad \{\ \text{definition of } \mathit{iterate}\ \} \\ & \mathit{map}\, h\, (a \;\prec\; \mathit{iterate}\, f_1\, (f_1\ a)) \\ = & \quad \{\ \text{definition of } \mathit{map}\ \} \\ & h\ a \;\prec\; (\mathit{map}\, h \cdot \mathit{iterate}\, f_1)\, (f_1\ a) \end{array} \qquad \begin{array}{ll} & (\mathit{iterate}\, f_2 \cdot h)\ a \\ = & \quad \{\ \text{definition of } \mathit{iterate}\ \} \\ & h\ a \;\prec\; \mathit{iterate}\, f_2\, (f_2\, (h\ a)) \\ = & \quad \{\ \text{assumption: } h \cdot f_1 = f_2 \cdot h\ \} \\ & h\ a \;\prec\; (\mathit{iterate}\, f_2 \cdot h)\, (f_1\ a) \end{array}$$

The fusion law implies $map\, f \cdot iterate\, f = iterate\, f \cdot f$, which in turn is the key for proving that $iterate\, f\, a$ is the unique solution of $x = a \prec map\, f\, x$.

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

Consequently, $nat = iterate\, (+1)\, 0$.

3 Scans

Let's meet some old friends. Many list-processing functions can be ported to streams, in fact, most of the functions that *generate* lists, such as *repeat* or *iterate*. Functions that *consume* lists, such as *foldr* or *foldl*, can be adapted with varying success, depending on their strictness. The tail-strict *foldl*, for instance, cannot be made to work with streams. We can however turn *scanr* and *scanl*, the list-producing variants of *foldr* and *foldl*, into stream operators.

$$\begin{array}{l} scanr \;::\; (\alpha \to \beta \to \beta) \to \beta \to (Stream\, \alpha \to Stream\, \beta) \\ scanr\, (\circledast)\, e\, s \;=\; t \quad \textbf{where} \quad t \;=\; e \prec zip\, (\circledast)\, s\, t \\ scanl \;::\; (\beta \to \alpha \to \beta) \to \beta \to (Stream\, \alpha \to Stream\, \beta) \\ scanl\, (\circledast)\, e\, s \;=\; t \quad \textbf{where} \quad t \;=\; e \prec zip\, (\circledast)\, t\, s \end{array}$$

If we follow our convention of abbreviating $zip\, (\circledast)\, s\, t$ by $s \circledast t$, the definitions of the ts become $t = e \prec s \circledast t$ and $t = e \prec t \circledast s$, emphasising the symmetry of the two scans. The schema below illustrates the working of $scanr\, (\circledast)\, e\, s$.

$$\begin{array}{cccccccccccc} & s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & \cdots \\ & \circledast & \circledast & \circledast & \circledast & \circledast & \circledast & \circledast & \circledast & \circledast & \circledast & \cdots \\ e & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & \cdots \\ \| & \| & \| & \| & \| & \| & \| & \| & \| & \| & \| & \cdots \\ t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} & \cdots \end{array}$$

The diagram makes explicit that $scanr\, (\cdot)\, e\, (s_0 \prec s_1 \prec s_2 \prec s_3 \prec \cdots)$ generates

$$e \;\prec\; s_0 \cdot e \;\prec\; s_1 \cdot (s_0 \cdot e) \;\prec\; s_2 \cdot (s_1 \cdot (s_0 \cdot e)) \;\prec\; \cdots \;,$$

that is, the expressions are nested to the right, but the elements appear in *reverse* order.[4] For instance, $scanr\, (:)\, [\,]$ generates partial reversals of the argument stream and $scanr\, (\cdot)\, id$ partially 'forward composes' a stream of functions.

[4] While *scanl* is the true counterpart of its list namesake, *scanr* isn't. The reason is that the list version of *scanr* is *not incremental*: in order to produce the first element of the output list it consumes the entire input list.

We shall also need unary versions of the scans.

$$\begin{aligned}
&\mathit{scanr1} \;::\; (\alpha \to \alpha \to \alpha) \to (\mathit{Stream}\,\alpha \to \mathit{Stream}\,\alpha)\\
&\mathit{scanr1}\,(\circledast)\,s \;=\; \mathit{scanr}\,(\circledast)\,(\mathit{head}\,s)\,(\mathit{tail}\,s)\\
&\mathit{scanl1} \;::\; (\alpha \to \alpha \to \alpha) \to (\mathit{Stream}\,\alpha \to \mathit{Stream}\,\alpha)\\
&\mathit{scanl1}\,(\circledast)\,s \;=\; \mathit{scanl}\,(\circledast)\,(\mathit{head}\,s)\,(\mathit{tail}\,s)
\end{aligned}$$

Note that the types of *scanr1* and *scanl1* are more restricted than the types of *scanr* and *scanl*.

Two important instances of *scanr* are summation, which we will need time and again, and product.

$$\begin{array}{lcllcl}
\Sigma & = & \mathit{scanr}\,(+)\,0 & \Sigma' & = & \mathit{scanr1}\,(+)\\
\Pi & = & \mathit{scanr}\,(*)\,1 & \Pi' & = & \mathit{scanr1}\,(*)
\end{array}$$

Both stream operators satisfy a multitude of laws [7]. For instance,

$$\begin{array}{lcllcl}
\Sigma(c * s) & = & c * \Sigma\, s & \Pi(s \mathbin{\hat{}} c) & = & \Pi\, s \mathbin{\hat{}} c\\
\Sigma(s + t) & = & \Sigma\, s + \Sigma\, t & \Pi(s * t) & = & \Pi\, s * \Pi\, t\ .
\end{array}$$

The laws are in fact instances of general properties of scans. First of all, scans enjoy two fusion properties.

$$\begin{aligned}
&\mathit{map}\,h\,(\mathit{scanr}\,(\circledast)\,e\,s) \;=\; \mathit{scanr}\,(\oplus)\,n\,s && \text{(fusion)}\\
&\qquad\Longleftarrow\quad h\,e \;=\; n \;\wedge\; h\,(a \circledast b) \;=\; a \oplus h\,b\\
&\mathit{scanr}\,(\circledast)\,e\,(\mathit{map}\,h\,s) \;=\; \mathit{scanr}\,(\oplus)\,e\,s && \text{(functor fusion)}\\
&\qquad\Longleftarrow\quad h\,a \circledast b \;=\; a \oplus b\\
&\mathit{scanr}\,(\circledast)\,e \;=\; \mathit{scanl}\,(\mathit{flip}\,(\circledast))\,e && \text{(flip)}
\end{aligned}$$

The flip law relates *scanr* to *scanl*; the function *flip* is given by $\mathit{flip}\,f\,a\,b = f\,b\,a$.

The fusion laws can be shown using parametricity [12]. The type of *scanr* contains two type variables, α and β, the *fusion law* amounts to the free theorem in β and the *functor fusion law* to the free theorem in α. However, we need not rely on parametricity as we can also employ the unique-fixed-point principle. For fusion we show that $\mathit{map}\,h\,(\mathit{scanr}\,(\circledast)\,e\,s) = \mathit{pure}\,h \diamond \mathit{scanr}\,(\circledast)\,e\,s$ satisfies $x = n \prec s \oplus x$, the recursion equation of $\mathit{scanr}\,(\oplus)\,n\,s$.

$$\begin{aligned}
&\mathit{pure}\,h \diamond \mathit{scanr}\,(\circledast)\,e\,s\\
=\quad &\{\ \text{definition of } \mathit{scanr} \text{ and } \diamond\ \}\\
&h\,e \;\prec\; \mathit{pure}\,h \diamond (s \circledast \mathit{scanr}\,(\circledast)\,e\,s)\\
=\quad &\{\ \text{assumption: } h\,e = n \text{ and } h\,(a \circledast b) = a \oplus h\,b \text{ lifted to streams}\ \}\\
&n \;\prec\; s \oplus (\mathit{pure}\,h \diamond \mathit{scanr}\,(\circledast)\,e\,s)
\end{aligned}$$

The proof of functor fusion can be found in Appendix A, along with most of the remaining *purely structural* proofs.

Scans also satisfy two distributive laws: if $\circledast$ distributes over $\oplus$, then $\circledast$ also distributes over $scanr\,(\oplus)$; furthermore, $scanr\,(\oplus)$ distributes over $\oplus$.

$$\begin{array}{lcll} scanr\,(\oplus)\,(c \circledast e)\,(c \circledast s) & = & c \circledast scanr\,(\oplus)\,e\,s & \\ \Longleftarrow \quad c \circledast (a \oplus b) & = & (c \circledast a) \oplus (c \circledast b) & \text{(distributivity 1)} \\ scanr\,(\oplus)\,(a \oplus b)\,(s \oplus t) & = & scanr\,(\oplus)\,a\,s \oplus scanr\,(\oplus)\,b\,t & \\ \Longleftarrow \quad \oplus \text{ AC} & & & \text{(distributivity 2)} \end{array}$$

Note that we use $\oplus$ and $\circledast$ both lifted and unlifted; likewise, c stands both for a constant stream and the constant itself. Finally, AC is shorthand for associative and commutative.

The first law is in fact a direct consequence of the fusion laws.

$$\begin{array}{ll} & scanr\,(\oplus)\,(c \circledast e)\,(c \circledast s) \\ = & \quad \{ \text{ functor fusion: } h\,a = c \circledast a \,\} \\ & scanr\,(\lambda a\,b \to (c \circledast a) \oplus b)\,(c \circledast e)\,s \\ = & \quad \{ \text{ fusion: } h\,a = c \circledast a \text{ and } c \circledast (a \oplus b) = c \circledast a \oplus c \circledast b \text{ by assumption } \} \\ & c \circledast scanr\,(\oplus)\,e\,s \end{array}$$

For the second law, we show that $scanr\,(\oplus)\,a\,s \oplus scanr\,(\oplus)\,b\,t$ satisfies $x = (a \oplus b) \prec (s \oplus t) \oplus x$, the recursion equation of $scanr\,(\oplus)\,(a \oplus b)\,(s \oplus t)$.

$$\begin{array}{ll} & scanr\,(\oplus)\,a\,s \oplus scanr\,(\oplus)\,b\,t \\ = & \quad \{ \text{ definition of } scanr \,\} \\ & (a \;\prec\; s \oplus scanr\,(\oplus)\,a\,s) \oplus (b \;\prec\; t \oplus scanr\,(\oplus)\,b\,t) \\ = & \quad \{ \text{ definition of } zip\,(\oplus) \,\} \\ & (a \oplus b) \;\prec\; (s \oplus scanr\,(\oplus)\,a\,s) \oplus (t \oplus scanr\,(\oplus)\,b\,t) \\ = & \quad \{ \text{ assumption: } \oplus \text{ AC } \} \\ & (a \oplus b) \;\prec\; (s \oplus t) \oplus (scanr\,(\oplus)\,a\,s \oplus scanr\,(\oplus)\,b\,t) \end{array}$$

4 Convolutions

Now, let's make some new friends. Moessner's theorem is about repeated summations. We noted in the introduction that repeated summation of $repeat\,1$ yields the columns of Pascal's triangle: $repeat\,1 = \binom{nat}{0}$ and $\Sigma\binom{nat}{k} = \binom{nat}{k+1}$ where $\binom{s}{t}$ is the binomial coefficient lifted to streams. What happens if we repeatedly sum an arbitrary stream? For instance, $\Sigma' \cdot \Sigma'$ takes $t_1 \prec t_2 \prec t_3 \prec \cdots$ to

$$1 * t_1 \;\prec\; 2 * t_1 + 1 * t_2 \;\prec\; 3 * t_1 + 2 * t_2 + 1 * t_3 \;\prec\; \cdots \;.$$

Note that the factors are going down whereas the indices are going up: double summation is an example of a so-called *convolution*. To understand the workings

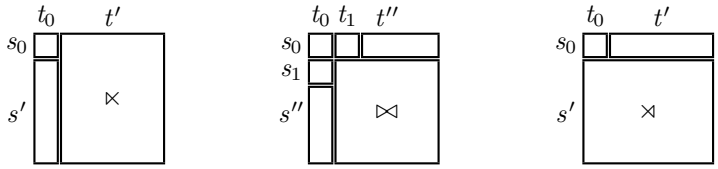
of a convolution, imagine two rows of people shaking hands while passing in opposite directions.

$$\begin{array}{lll} \ldots\ s_4\ s_3\ s_2\ s_1 \longrightarrow & \ldots\ s_4\ s_3\ s_2\ s_1 \longrightarrow & \ldots\ s_4\ s_3\ s_2\ s_1 \longrightarrow \\ \quad \longleftarrow\ t_1\ t_2\ t_3\ t_4\ \ldots & \quad \longleftarrow\ t_1\ t_2\ t_3\ t_4\ \ldots & \quad \longleftarrow\ t_1\ t_2\ t_3\ t_4\ \ldots \end{array}$$

Firstly, 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. Two operations are involved in a convolution: one operation that corresponds to the handshake and a second operation, typically associative, that combines the results of the handshake.

$$\begin{array}{lll} \ldots\ s_4\ s_3\ s_2\ s_1 \longrightarrow & \ldots\ s_4\ s_3\ s_2\ \ s_1 \longrightarrow & \ldots\ s_4\ s_3\ \ \ s_2\ \ \ s_1 \longrightarrow \\ \qquad\quad \circledast & \qquad\ \circledast \oplus \circledast & \qquad\ \circledast \oplus \circledast \oplus \circledast \\ \quad \longleftarrow\ t_1\ t_2\ t_3\ t_4\ \ldots & \quad \longleftarrow\ t_1\ \ \ t_2\ t_3\ t_4\ \ldots & \quad \longleftarrow\ t_1\ \ \ t_2\ \ \ t_3\ t_4\ \ldots \end{array}$$

Unfortunately, when it comes to the implementation, the symmetry of the description is lost. There are at least three ways to set up the corecursion (we abbreviate *head s* by s_0 and *tail s* by s').



Assume that $\circledast$ implements the handshake. The first element of the convolution is $s_0 \circledast t_0$. Then we can either combine $s' \circledast \mathit{pure}\, t_0$ with the convolution of s and t' (diagram on the left) or we can combine the convolution of s' and t with $\mathit{pure}\, s_0 \circledast t'$ (diagram on the right).

$$\begin{aligned} &\mathit{convolutel} \;::\; (\alpha \to \beta \to \gamma) \to (\gamma \to \gamma \to \gamma) \\ &\qquad\qquad \to (\mathit{Stream}\,\alpha \to \mathit{Stream}\,\beta \to \mathit{Stream}\,\gamma) \\ &\mathit{convolutel}\,(\circledast)\,(\oplus) \;=\; (\ltimes)\;\mathbf{where} \\ &\quad s \ltimes t \;=\; \mathit{head}\,s \circledast \mathit{head}\,t \prec \mathit{zip}\,(\oplus)\,(\mathit{map}\,(\circledast\,\mathit{head}\,t)\,(\mathit{tail}\,s))\,(s \ltimes \mathit{tail}\,t) \end{aligned}$$

$$\begin{aligned} &\mathit{convoluter} \;::\; (\alpha \to \beta \to \gamma) \to (\gamma \to \gamma \to \gamma) \\ &\qquad\qquad \to (\mathit{Stream}\,\alpha \to \mathit{Stream}\,\beta \to \mathit{Stream}\,\gamma) \\ &\mathit{convoluter}\,(\circledast)\,(\oplus) \;=\; (\rtimes)\;\mathbf{where} \\ &\quad s \rtimes t \;=\; \mathit{head}\,s \circledast \mathit{head}\,t \prec \mathit{zip}\,(\oplus)\,(\mathit{tail}\,s \rtimes t)\,(\mathit{map}\,(\mathit{head}\,s \circledast)\,(\mathit{tail}\,t)) \end{aligned}$$

It is not too hard to show that the two variants are equal if $\oplus$ is associative, see Appendix A. The proof makes essential use of the symmetric definition in the middle. We shall now assume associativity and abbreviate *convolutel* and *convoluter*, by *convolute*.

Two instances of this corecursion scheme are convolution product and convolution exponentiation.

$$\begin{aligned} &\mathbf{infixl}\,7\ \ast\!\ast \\ &s \ast\!\ast\, t \;=\; \mathit{convolute}\,(*)\,(+)\,s\,t \\ &\mathbf{infixr}\,8\ \frown \\ &s \frown t \;=\; \mathit{convolute}\,(\hat{}\,)\,(*)\,s\,t \end{aligned}$$

If you are familiar with generating functions [3], you will recognise $\ast\!\!\ast$ as the product of generating functions. Exponentiation $\frown$ is comparatively unknown; we shall need it for formalising Moessner's process, but more on that later.

Both operators satisfy a multitude of laws which merit careful study.

$$\begin{array}{rcl@{\qquad}rcl}
(c \ast s) \ast\!\!\ast t &=& c \ast (s \ast\!\!\ast t) & (s \mathbin{\hat{}} c) \frown t &=& (s \frown t) \mathbin{\hat{}} c \\
(s + t) \ast\!\!\ast u &=& s \ast\!\!\ast u + t \ast\!\!\ast u & (s \ast t) \frown u &=& s \frown u \ast t \frown u \\
\Sigma\, s \ast\!\!\ast t &=& \Sigma(s \ast\!\!\ast t) & \Pi\, s \frown t &=& \Pi(s \frown t) \\
s \ast\!\!\ast (t \ast c) &=& (s \ast\!\!\ast t) \ast c & s \frown (t \ast c) &=& (s \frown t) \mathbin{\hat{}} c \\
s \ast\!\!\ast (t + u) &=& s \ast\!\!\ast t + s \ast\!\!\ast u & s \frown (t + u) &=& s \frown t \ast s \frown u \\
s \ast\!\!\ast \Sigma\, t &=& \Sigma(s \ast\!\!\ast t) & s \frown \Sigma\, t &=& \Pi(s \frown t)
\end{array}$$

Note that we can shift Σ in and out of a convolution product. This allows us to express repeated summations as convolutions:

$$\begin{array}{ll}
& \Sigma\, s \\
= & \quad \{\ (1 \prec \mathit{repeat}\, 0) \ast\!\!\ast t = t\ \} \\
& \Sigma((1 \prec \mathit{repeat}\, 0) \ast\!\!\ast s) \\
= & \quad \{\ \Sigma\, t \ast\!\!\ast u = \Sigma(t \ast\!\!\ast u)\ \} \\
& \Sigma(1 \prec \mathit{repeat}\, 0) \ast\!\!\ast s \\
= & \quad \{\ \Sigma(1 \prec \mathit{repeat}\, 0) = 0 \prec \mathit{repeat}\, 1\ \} \\
& (0 \prec \mathit{repeat}\, 1) \ast\!\!\ast s \\
= & \quad \{\ (0 \prec t) \ast\!\!\ast u = 0 \prec t \ast\!\!\ast u\ \} \\
& 0 \prec \mathit{repeat}\, 1 \ast\!\!\ast s\ .
\end{array}$$

Hence, $\Sigma'\, s = \mathit{repeat}\, 1 \ast\!\!\ast s$ and (see motivating example),

$$\begin{array}{ll}
& \Sigma'(\Sigma'\, s) \\
= & \quad \{\ \text{see above}\ \} \\
& \Sigma'(\mathit{repeat}\, 1 \ast\!\!\ast s) \\
= & \quad \{\ \Sigma'\, t \ast\!\!\ast u = \Sigma'(t \ast\!\!\ast u)\ \} \\
& \Sigma'(\mathit{repeat}\, 1) \ast\!\!\ast s \\
= & \quad \{\ \Sigma'(\mathit{repeat}\, 1) = \mathit{nat}'\ \} \\
& \mathit{nat}' \ast\!\!\ast s\ .
\end{array}$$

Perhaps unsurprisingly, the laws above are instances of general properties of convolutions. Like scans, convolutions satisfy two fusion properties and a flip law.

$$\begin{array}{l}
\mathit{map}\, h\, (\mathit{convolute}\, (\circledast)\, (\oplus)\, s\, t) \;=\; \mathit{convolute}\, (\circledcirc)\, (\boxplus)\, s\, t \qquad \text{(fusion)} \\
\qquad \Longleftarrow \quad h\, (c_1 \oplus c_2) \;=\; h\, c_1 \boxplus h\, c_2 \quad \wedge \quad h\, (a \circledast b) \;=\; a \circledcirc b \\
\mathit{convolute}\, (\circledast)\, (\oplus)\, (\mathit{map}\, h\, s)\, (\mathit{map}\, k\, t) \;=\; \mathit{convolute}\, (\circledcirc)\, (\oplus)\, s\, t \\
\qquad \Longleftarrow \quad h\, a \circledast k\, b \;=\; a \circledcirc b \qquad \text{(functor fusion)} \\
\mathit{flip}\, (\mathit{convolute}\, (\circledast)\, (\oplus)) \;=\; \mathit{convolute}\, (\mathit{flip}\, (\circledast))\, (\oplus) \qquad \text{(flip)}
\end{array}$$

The laws for $\ast\!\!\ast$ and $\frown$ suggest that convolutions enjoy three different types of distributive laws. Let $s \bowtie t = \mathit{convolute}\,(\circledast)\,(\oplus)\,s\,t$, then

$$
\begin{array}{lll}
(c \boxplus s) \bowtie t & = & c \circledast (s \bowtie t) \qquad \text{(distributivity 1)} \\
\quad \Longleftarrow & & c \circledast (c_1 \oplus c_2) \;=\; (c \circledast c_1) \oplus (c \circledast c_2) \\
\quad \wedge & & c \circledast (a \circledast b) \;=\; (c \boxplus a) \circledast b \\
(s \boxplus t) \bowtie u & = & (s \bowtie u) \oplus (t \bowtie u) \qquad \text{(distributivity 2)} \\
\quad \Longleftarrow & & \oplus \text{ AC} \;\wedge\; (a_1 \boxplus a_2) \circledast b \;=\; (a_1 \circledast b) \oplus (a_2 \circledast b) \\
(\mathit{scanr}\,(\boxplus)\,n\,s) \bowtie t & = & \mathit{scanr}\,(\oplus)\,e\,(s \bowtie t) \qquad \text{(distributivity 3)} \\
\quad \Longleftarrow & & \oplus \text{ AC} \;\wedge\; (a_1 \boxplus a_2) \circledast b \;=\; (a_1 \circledast b) \oplus (a_2 \circledast b) \\
\quad \wedge & & n \circledast b \;=\; e \;\wedge\; a \oplus e \;=\; a\ .
\end{array}
$$

Again, the proofs of the laws can be found in Appendix A. Furthermore, there are analogous laws for right distributivity.

5 Moessner's process formalised

We are finally in a position to formalise Moessner's process. Consider the examples in the introductory section and note that the process generates a sequence of equilateral triangles. In the case of natural powers, the triangles were of the same size

1	2	3	4	5	6	7	8	9	10	11	12
1	3		7	12		19	27		37	48	
1			8			27			64	;	

for the factorials, we steadily increased their size

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2		6	11		18	26	35		46	58	71	85	
			6			24	50			96	154	225		
						24				120	274			
										120	.			

Our goal is to relate the elements in the upper right corners, the sequence of deleted positions, to the elements in the lower left corners, the sequence generated by Moessner's process. It turns out that this is most easily accomplished through a third sequence, the sequence of *size differences*. Assuming that we start off with an invisible triangle of size 0, the size differences for the above examples are $3 \prec \mathit{repeat}\,0$ and $\mathit{repeat}\,1$, respectively.

Paasche's generalisation of Moessner's theorem then reads: If d is a sequence of size differences, then

$$\mathit{nat}' \ast\!\!\ast\, d$$

is the sequence of deleted positions and

$$nat' \mathbin{\frown\!\!\!\frown} d$$

is the sequence obtained by Moessner's process.

The first part of this beautiful correspondence is easy to explain: if d is the sequence of size differences, then $\Sigma' d$ is the sequence of sizes and $\Sigma'(\Sigma' d) = nat' \mathbin{⁂} d$ is the sequence of deleted positions. Before we tackle the second part, let's have a look at some examples first.[5]

$$\begin{aligned}
&\gg\ nat' \mathbin{⁂} (2 \prec repeat\, 0)\\
&\langle 2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,\ldots\rangle\\
&\gg\ nat' \mathbin{\frown\!\!\!\frown} (2 \prec repeat\, 0)\\
&\langle 1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,\ldots\rangle\\
&\gg\ nat' \mathbin{⁂} (3 \prec repeat\, 0)\\
&\langle 3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,\ldots\rangle\\
&\gg\ nat' \mathbin{\frown\!\!\!\frown} (3 \prec repeat\, 0)\\
&\langle 1,8,27,64,125,216,343,512,729,1000,1331,1728,2197,2744,3375,\ldots\rangle\\
&\gg\ nat' \mathbin{⁂} repeat\, 1\\
&\langle 1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,\ldots\rangle\\
&\gg\ nat' \mathbin{\frown\!\!\!\frown} repeat\, 1\\
&\langle 1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,\ldots\rangle\\
&\gg\ nat' \mathbin{⁂} (2 \prec 7 \prec 6 \prec 5 \prec repeat\, 0)\\
&\langle 2,11,26,46,66,86,106,126,146,166,186,206,226,246,266,286,\ldots\rangle\\
&\gg\ nat' \mathbin{\frown\!\!\!\frown} (2 \prec 7 \prec 6 \prec 5 \prec repeat\, 0)\\
&\langle 1,4,1152,2239488,9555148800,2799360000000,219469824000000,\ldots\rangle
\end{aligned}$$

It is not too hard to calculate the results: We have $s \mathbin{⁂} (k \prec repeat\, 0) = s * repeat\, k$ and $s \mathbin{\frown\!\!\!\frown} (k \prec repeat\, 0) = s \mathbin{\hat{}} repeat\, k$, which implies Moessner's original theorem. Furthermore, $s \mathbin{⁂} 1 = \Sigma' s$ and $s \mathbin{\frown\!\!\!\frown} 1 = \Pi' s$ which explains why we obtain the factorials when we increase the size of the triangles by 1.

Let's get more adventurous. If we increase the *size difference*, we obtain the so-called *superfactorials* ($A000178$), the products of the first n factorials: $nat' \mathbin{\frown\!\!\!\frown} nat = nat' \mathbin{\frown\!\!\!\frown} \Sigma 1 = \Pi(nat' \mathbin{\frown\!\!\!\frown} 1) = \Pi(\Pi' nat') = \Pi fac'$. Taking this one step further, recall that $\Sigma \binom{nat}{k} = \binom{nat}{k+1}$. Consequently, $nat' \mathbin{\frown\!\!\!\frown} \binom{nat}{2}$ yields the *superduperfactorials* ($A055462$), the products of the first n superfactorials: $nat' \mathbin{\frown\!\!\!\frown} \binom{nat}{2} = nat' \mathbin{\frown\!\!\!\frown} \Sigma(\Sigma 1) = \Pi(\Pi fac')$.

In the introduction we asked for the sequences we obtain if we start off by deleting the squares or the cubes. This is easily answered using the left-inverse of Σ', the difference operator $\nabla s = head\, s \prec \Delta s$ where $\Delta s = tail\, s - s$ is the left-inverse of Σ. We have $\nabla(\nabla(nat' \mathbin{\hat{}} 2)) = 1 \prec repeat\, 2$ ($A040000$). Consequently, Moessner's process generates $nat' \mathbin{\frown\!\!\!\frown} (1 \prec repeat\, 2) = 1 \prec (nat+2) * (nat' \mathbin{\frown\!\!\!\frown} 2) = 1 \prec (nat+2) * fac' \mathbin{\hat{}} 2 = fac * fac'$ ($A010790$). We leave the cubes as an instructive exercise to the reader.

[5] This is an interactive session. The part after the prompt "$\gg$ " is the user's input. The result of each submission is shown in the subsequent line. The actual output of the Haskell interpreter is displayed; the session has been generated automatically using lhs2TEX's active features [13].

6 Moessner's process verified

⟨ . . . ⟩ *obtaining a correct solution demands particular attention to the avoidance of unnecessary detail.*

The Capacity-C Torch Problem—Roland Backhouse

We have noted that Moessner's process generates a sequence of triangles. Let's look at them more closely. Below is the process that generates the cubes, now with an additional row of ones on top.

	0	0	0	0		0	0	0	0		0	0	0	0
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			

Every element within a triangle is the sum of the element to its left and the element above. The values in the two margins are zero, except for the topmost element in the left margin, which is 1 and acts as the initial seed.

The verbal description suggests that the sequences are formed from top to bottom. An alternative view, which turns out to be more fruitful, is that they are formed from left to right. We start with the vector (0 0 0 1) (left margin read from bottom to top). The vector goes through a *binomial process*, which yields the diagonal vector (1 3 3 1). This vector goes through the same process yielding (8 12 6 1), and so forth. The first elements of these vectors are the cubes.

	0	0	0	0		0	0	0	0		0	0	0	0	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	2	3		3	4	5	6		6	7	8	9		9
0	1	3			3	7	12			12	19	27			27
0	1				1	8				8	27				27

In general, the input and output vectors are related by

	0	0	0	0	
a_3	a_3	a_3	a_3	a_3	b_3
a_2	$a_2 + a_3$	$a_2 + 2a_3$	$a_2 + 3a_3$		b_2
a_1	$a_1 + a_2 + a_3$	$a_1 + 2a_2 + 3a_3$			b_1
a_0	$a_0 + a_1 + a_2 + a_3$				b_0 ,

or somewhat snappier,

$$b_n = \sum_k \binom{k}{n} a_k \ ,$$

where k and n range over natural numbers. (This is really a finite sum, since only a finite number of coefficients are nonzero.) As to be expected, the formula involves a binomial coefficient. At this point, we could rely on our mathematical

skills and try to prove Moessner's theorem by manipulating binomial coefficients and this is indeed what some authors have done [5, 14].

But there is a more attractive line of attack: Let us view the vectors as coefficients of a polynomial so that $(a_0\, a_1\, a_2\, a_3 \ldots)$ represents $f(x) = \sum_n a_n x^n$. A triangle transformation is then a higher-order function, a function that maps polynomials to polynomials. We can calculate this higher-order mapping as follows.

$$
\begin{aligned}
& \sum_n b_n x^n \\
= \quad & \{ \text{ definition of } b_n \ \} \\
& \sum_n \left(\sum_k \binom{k}{n} a_k \right) x^n \\
= \quad & \{ \text{ distributive law } \} \\
& \sum_n \sum_k \binom{k}{n} a_k x^n \\
= \quad & \{ \text{ interchanging the order of summation } \} \\
& \sum_k \sum_n \binom{k}{n} a_k x^n \\
= \quad & \{ \text{ distributive law } \} \\
& \sum_k a_k \left(\sum_n \binom{k}{n} x^n \right) \\
= \quad & \{ \text{ binomial theorem } \} \\
& \sum_k a_k (x+1)^k
\end{aligned}
$$

Here is the punch line: Under the functional view, each triangle maps f to $f \ll 1$ where $\ll$, the *shift operator*, is given by

$$
\begin{aligned}
(\ll) \quad & :: \quad (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}) \\
f \ll n \quad & = \quad \lambda x \to f\,(x+n) \ .
\end{aligned}
$$

This change of representation simplifies matters dramatically. By going higher-order we avoid unnecessary detail in the sense of Roland Backhouse [15].

6.1 Moessner's original theorem

Moessner's original sequence, where the size of the triangles is constant, is then given by the sequence of polynomials idiomatically applied to 0—the application extracts the lowest coefficients.

$$
\begin{aligned}
& \mathit{moessner} \quad :: \quad \mathbb{Z} \to \mathit{Stream}\,\mathbb{Z} \\
& \mathit{moessner}\,k \quad = \quad \mathit{iterate}\,(\ll 1)\,\mathit{id}^k \diamond 0
\end{aligned}
$$

$$f^k \;=\; \lambda x \to (f\,x)^k$$

The seed polynomial is id^k, which is then repeatedly shifted by 1. The auxiliary definition lifts exponentiation to functions, so id^k is $\lambda x \to x^k$. (Below, we also use s^k to denote exponentiation lifted to streams.)

We are now in a position to prove Moessner's theorem: $moessner\,k = nat^k$. (Of course, this equation is a special case of the relation given in Section 5, but we prove it nonetheless as it serves as a good warm-up exercise.) The central observation is that two shifts can be contracted to one: $(q \ll i) \ll j = q \ll (i+j)$. This allows us to eliminate the iteration:

$$iterate\,(\ll 1)\,p = pure\,(\ll) \diamond pure\,p \diamond nat \;\;. \tag{1}$$

The proof of this equation relies on the fact that $iterate\,f\;a$ is the unique solution of $x = a \prec map\,f\,x$, see Section 2.3.

$$\begin{aligned}
&pure\,(\ll) \diamond pure\,p \diamond nat \\
=\;& \quad \{\ \text{definition of } pure \text{ and } nat\ \} \\
&p \ll 0 \;\prec\; pure\,(\ll) \diamond pure\,p \diamond (nat + 1) \\
=\;& \quad \{\ q \ll 0 = q \text{ and } (q \ll i) \ll j = q \ll (i+j) \text{ lifted to streams}\ \} \\
&p \;\prec\; pure\,(\ll) \diamond (pure\,(\ll) \diamond pure\,p \diamond nat) \diamond 1 \\
=\;& \quad \{\ \text{idioms}\ \} \\
&p \;\prec\; map\,(\ll 1) \diamond (pure\,(\ll) \diamond pure\,p \diamond nat)
\end{aligned}$$

The proof of Moessner's theorem then boils down to a three-liner.

$$\begin{aligned}
&iterate\,(\ll 1)\,id^k \diamond 0 \\
=\;& \quad \{\ \text{equation (1)}\ \} \\
&(pure\,(\ll) \diamond pure\,id^k \diamond nat) \diamond 0 \\
=\;& \quad \{\ (id^k \ll n)\,0 = n^k \text{ lifted to streams}\ \} \\
&nat^k
\end{aligned}$$

6.2 Paasche's generalisation of Moessner's theorem

I know also that formal calculation is not a spectator sport: $\langle \ldots \rangle$

Making Formality Work For Us—Roland Backhouse

Now, what changes when the size of the triangles increases by $i > 0$? In this case, the input vector must additionally be padded with i zeros to fit the size of the next triangle, for instance, $(1\,3\,3\,1)$ becomes $(0 \cdots 0\,1\,3\,3\,1)$. In other words, the polynomial must be multiplied by id^i. Lifting multiplication to functions,

$$f * g \;=\; \lambda x \to f\,x * g\,x \;\;,$$

a 'generalised triangle transformation' is captured by $step\,i$ where i is the increase in size.

$$\begin{array}{lcl} step & :: & \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z}) \\ step\,i\,f & = & (id^i * f) \ll 1 \end{array}$$

Finally, Paasche's process is the partial 'forward composition' of the $step\,i$ functions idiomatically applied to the initial polynomial id^0 idiomatically applied to the constant 0.

$$\begin{array}{lcl} paasche & :: & Stream\,\mathbb{Z} \to Stream\,\mathbb{Z} \\ paasche\,d & = & tail\,(scanr\,(\cdot)\,id\,(map\,step\,d) \diamond pure\,id^0 \diamond pure\,0) \end{array}$$

The *tail* discards the value of the initial polynomial, which is not part of the resulting sequence. A truly functional approach: a stream of 2nd-order functions is transformed into a stream of 1st-order functions, which in turn is transformed to a stream of numbers.

It remains to show that $paasche\,d = nat' \bowtie d$. We start off with some routine calculations manipulating the scan.

$$\begin{array}{ll} & scanr\,(\cdot)\,id\,(map\,step\,d) \diamond pure\,id^0 \\ = & \quad \{\ \text{idiom interchange}\ \} \\ & pure\,(\$\,id^0) \diamond scanr\,(\cdot)\,id\,(map\,step\,d) \\ = & \quad \{\ \text{scan fusion: } h\,g = g\,id^0\ \} \\ & scanr\,(\$)\,id^0\,(map\,step\,d) \\ = & \quad \{\ \text{scan functor fusion: } h = step \text{ and } (f\$) = f\ \} \\ & scanr\,step\,id^0\,d \\ = & \quad \{\ \text{definition of } step\ \} \\ & scanr\,(\lambda i\,g \to (id^i * g) \ll 1)\,id^0\,d \\ = & \quad \{\ (f * g) \ll k = (f \ll k) * (g \ll k)\ \} \\ & scanr\,(\lambda i\,g \to (id^i \ll 1) * (g \ll 1))\,id^0\,d \\ = & \quad \{\ id^0 = id^0 \ll 1 \text{ and scan functor fusion: } h\,i = id^i \ll 1\ \} \\ & scanr\,(\lambda f\,g \to f * (g \ll 1))\,(id^0 \ll 1)\,(map\,(\lambda i \to id^i \ll 1)\,d) \\ = & \quad \{\ \text{definition of } scanr1\ \} \\ & scanr1\,(\lambda f\,g \to f * (g \ll 1))\,(map\,(\lambda i \to id^i \ll 1)\,(0 \prec d)) \end{array}$$

How can we make further progress? Let's introduce a new operator for *scanr1*'s first argument,

$$f \triangleleft g \quad = \quad f * (g \ll 1)\ ,$$

and let's investigate what happens when we nest invocations of $\triangleleft$ (recall that *scanr1* arranges the elements in reverse order).

$$f_2 \triangleleft (f_1 \triangleleft f_0)$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } \triangleleft \} \\
& f_2 * (f_1 * f_0 \ll 1) \ll 1 \\
= & \{ \ (f * g) \ll k = (f \ll k) * (g \ll k) \ \} \\
& f_2 * f_1 \ll 1 * (f_0 \ll 1) \ll 1 \\
= & \{ \ f = f \ll 0 \text{ and } (f \ll i) \ll j = f \ll (i + j) \ \} \\
& f_2 \ll 0 * f_1 \ll 1 * f_0 \ll 2
\end{array}
$$

The scan corresponds to a convolution! Let $s \bowtie t = \mathit{convolute}\ (\ll)\ (*)\ s\ t$, then

$$\mathit{scanr1}\ (\triangleleft)\ s \quad = \quad s \bowtie t \quad \textbf{where} \quad t \quad = \quad 0 \prec t + \mathit{pure}\ 1 \ . \tag{2}$$

In our case, t equals nat. The relation, however, holds for arbitrary operators that satisfy the three laws: $(a_1 * a_2) \ll b = (a_1 \ll b) * (a_2 \ll b)$, $(a \ll b_1) \ll b_2 = a \ll (b_1 + b_2)$ and $a \ll 0 = a$ (see also convolution distributivity 1). In general, if $(A, +, 0)$ is a monoid and 1 an element of A, then t is the so-called sequence of 'powers' of 1.

Turning to the proof of equation (2), we show that $s \bowtie t$ satisfies $x = \mathit{head}\ s \prec \mathit{tail}\ s \triangleleft x$, the recursion equation of $\mathit{scanr1}\ (\triangleleft)\ s$.

$$
\begin{array}{ll}
& s \bowtie t \\
= & \{ \text{ definition of } \bowtie \text{ and definition of } t \ \} \\
& \mathit{head}\ s \ll 0 \ \prec \ \mathit{tail}\ s \ll 0 * s \bowtie (t + \mathit{pure}\ 1) \\
= & \{ \ a \ll 0 = a \ \} \\
& \mathit{head}\ s \ \prec \ \mathit{tail}\ s * s \bowtie (t + \mathit{pure}\ 1) \\
= & \{ \text{ convolution distributivity 1: } s \bowtie (t + \mathit{pure}\ 1) = (s \bowtie t) \ll \mathit{pure}\ 1 \ \} \\
& \mathit{head}\ s \ \prec \ \mathit{tail}\ s * (s \bowtie t) \ll \mathit{pure}\ 1 \\
= & \{ \text{ definition of } \triangleleft \} \\
& \mathit{head}\ s \ \prec \ \mathit{tail}\ s \triangleleft (s \bowtie t)
\end{array}
$$

By rewriting the scan as a convolution we can complete the proof of Moessner's theorem—the remaining steps are again mostly routine calculations. Let $e = 0 \prec d$, then

$$
\begin{array}{ll}
& \mathit{scanr1}\ (\lambda f\ g \to f * (g \ll 1))\ (\mathit{map}\ (\lambda i \to id^i \ll 1)\ e) \diamond \mathit{pure}\ 0 \\
= & \{ \text{ equation (2) } \} \\
& \mathit{convolute}\ (\ll)\ (*)\ (\mathit{map}\ (\lambda i \to id^i \ll 1)\ e)\ \mathit{nat} \diamond \mathit{pure}\ 0 \\
= & \{ \text{ convolution functor fusion: } h\ i = id^i \ll 1 \text{ and } k = id \ \} \\
& \mathit{convolute}\ (\lambda i\ n \to (id^i \ll 1) \ll n)\ (*)\ e\ \mathit{nat} \diamond \mathit{pure}\ 0 \\
= & \{ \ (f \ll i) \ll j = f \ll (i + j) \ \} \\
& \mathit{convolute}\ (\lambda i\ n \to id^i \ll (1 + n))\ (*)\ e\ \mathit{nat} \diamond \mathit{pure}\ 0 \\
= & \{ \text{ convolution functor fusion: } h = id \text{ and } k\ n = 1 + n \ \} \\
& \mathit{convolute}\ (\lambda i\ n \to id^i \ll n)\ (*)\ e\ \mathit{nat}' \diamond \mathit{pure}\ 0 \\
= & \{ \text{ idiom interchange } \}
\end{array}
$$

$$
\begin{aligned}
& \mathit{pure}\,(\$\,0) \diamond \mathit{convolute}\,(\lambda i\; n \to id^i \ll n)\,(*)\; e\; nat' \\
= \quad & \{ \text{ convolution fusion: } h\; g = g\,0 \text{ and } h\,(id^i \ll n) = n \mathbin{\hat{}} i \;\} \\
& \mathit{convolute}\,(\lambda i\; n \to n \mathbin{\hat{}} i)\,(*)\; e\; nat' \\
= \quad & \{ \text{ convolution flip } \} \\
& \mathit{convolute}\,(\mathbin{\hat{}})\,(*)\; nat'\; e \\
= \quad & \{ \text{ definition of } \mathit{convolute} \text{ and } \frown \} \\
& 1 \prec nat' \frown d \;\; .
\end{aligned}
$$

This completes the proof.

7 Related work

The tale of Moessner's theorem In a one-page note, Moessner conjectured what is now known as Moessner's theorem [1]. The conjecture was proven in the subsequent note by Perron [5]. The proof mainly manipulates binomial coefficients. A year later, Paasche generalised Moessner's process to non-decreasing intervals [4, 16], while Salié considered an arbitrary start sequence [6]. Paasche's proof of the general theorem builds on the theory of generating functions and is quite intricate—the generating functions are in fact the 'reversals' of the polynomials considered in this paper. A snappier, but less revealing proof of the original theorem can be found in the textbook "Concrete mathematics" [3, Ex. 7.54]. Van Yzeren observed that Moessner's theorem can be looked upon as a consequence of Horner's algorithm for evaluating polynomials [17]. His idea of viewing the diagonals as coefficients of polynomials is at the heart of the development in Section 6. Inspired by Moessner's process, Long generalised Pascal's triangle, which he then used to prove Salié's theorem [14]. The same author also wrote a more leisurely exposition of the subject [18]. The paper hints at the relationship between the sequence of deleted positions and the sequence obtained by Moessner's generalised process formalised in Section 5.

Scans and convolutions To the best of the author's knowledge the material on scans and convolutions is original. Of course, there are several papers, most notably [19–23], that deal with special instances of the combinators, with sums and the convolution product, in particular.

8 Conclusion

Moessner's theorem and its generalisation nicely illustrate scans and convolutions. Though the theorems are number-theoretic, programming language theory provides a fresh view leading to snappier statements and more structured proofs. While scans are well-known, convolutions are under-appreciated; I think they deserve to be better known. Liberating scans and convolutions from their number-theoretic origins seems to be worthwhile: by turning them into *polymorphic* combinators, we literally obtain theorems for free [12]. Even though

we don't rely on parametricity for the proofs, we use parametricity as a guiding principle for formulating fusion laws. Of equal importance are distributive laws: convolution distributivity, for instance, allowed us to rewrite a scan as a convolution, a central step in the proof of Moessner's theorem.

All in all a nice little theory for an intriguing theorem. I hope to see further applications of scans and convolutions in the future.

Acknowledgements

Special thanks are due to Roland Backhouse for posing the challenge to prove Moessner's theorem within stream calculus. Furthermore, a big thank you to Daniel James for improving my English. Thanks are finally due to the anonymous referees for pointing out several presentational problems.

References

1. Moessner, A.: Eine Bemerkung über die Potenzen der natürlichen Zahlen. Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1951 Nr. 3 (March 1951) 29
2. Sloane, N.J.A.: The on-line encyclopedia of integer sequences `http://www.research.att.com/~njas/sequences/`.
3. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete mathematics. 2nd edn. Addison-Wesley Publishing Company, Reading, Massachusetts (1994)
4. Paasche, I.: Ein neuer Beweis des Moessnerschen Satzes. Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1952 Nr. 1 (February 1952) 1–5
5. Perron, O.: Beweis des Moessnerschen Satzes. Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1951 Nr. 4 (May 1951) 31–34
6. Salié, H.: Bemerkung zu einem Satz von Moessner. Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1952 Nr. 2 (February 1952) 7–11
7. Hinze, R.: Functional pearl: Streams and unique fixed points. In Thiemann, P., ed.: Proceedings of the 2008 International Conference on Functional Programming, ACM Press (September 2008) 189–200
8. Aczel, P., Mendler, N.: A final coalgebra theorem. In Pitt, D., Rydeheard, D., Dybjer, P., Poigné, A., eds.: Category Theory and Computer Science (Manchester). Volume 389 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (1989) 357–365
9. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
10. Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Technical Report CS-R9104, Centre of Mathematics and Computer Science, CWI, Amsterdam (January 1991)
11. McBride, C., Paterson, R.: Functional pearl: Applicative programming with effects. Journal of Functional Programming **18**(1) (2008) 1–13
12. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK, Addison-Wesley Publishing Company (September 1989) 347–359

13. Hinze, R., Löh, A.: Guide2lhs2tex (for version 1.14) (October 2008) `http://people.cs.uu.nl/andres/lhs2tex/`.
14. Long, C.: On the Moessner theorem on integral powers. The American Mathematical Monthly **73**(8) (October 1966) 846–851
15. Backhouse, R.: The capacity-C torch problem. In Audebaud, P., Paulin-Mohring, C., eds.: 9th International Conference on Mathematics of Program Construction (MPC '08). Volume 5133 of Lecture Notes in Computer Science., Springer-Verlag (July 2008) 57–78
16. Paasche, I.: Eine Verallgemeinerung des Moessnerschen Satzes. Compositio Mathematica **12** (1954) 263–270
17. van Yzeren, J.: A note on an additive property of natural numbers. The American Mathematical Monthly **66**(1) (January 1959) 53–54
18. Long, C.T.: Strike it out: Add it up. The Mathematical Gazette **66**(438) (December 1982) 273–277
19. Karczmarczuk, J.: Generating power of lazy semantics. Theoretical Computer Science (187) (1997) 203–219
20. McIlroy, M.D.: Power series, power serious. J. Functional Programming **3**(9) (May 1999) 325–337
21. McIlroy, M.D.: The music of streams. Information Processing Letters (77) (2001) 189–195
22. Rutten, J.: Fundamental study — Behavioural differential equations: a coinductive calculus of streams, automata, and power series. Theoretical Computer Science (308) (2003) 1–53
23. Rutten, J.: A coinductive calculus of streams. Math. Struct. in Comp. Science (15) (2005) 93–147

A Proofs

Most of the proofs have been relegated to this appendix as not to disturb the flow. For conciseness, we abbreviate $\mathit{head}\, s$ by s_0 and $\mathit{tail}\, s$ by s'. Furthermore, s_1 is shorthand for $\mathit{head}\,(\mathit{tail}\, s)$ and s'' for $\mathit{tail}\,(\mathit{tail}\, s)$ and so forth.

Several proofs establish the equality of two streams by showing that they satisfy the same recursion equation. These proofs are laid out as follows.

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

The symbol $\subset$ is meant to suggest a link connecting the upper and the lower part; the recursion equation is given within the curly braces (below we omit the "has a unique solution" blurb for reasons of space). When reading $\subset$-proofs, it is easiest to start at both ends working towards the link. Each part follows a typical pattern: starting with e we unfold the definitions obtaining $e_1 \prec e_2$; then we try to express e_2 in terms of e.

Scan functor fusion We show that $scanr\,(\circledast)\,e\,(pure\,h \diamond s)$ satisfies $x = e \prec s \oplus x$, the recursion equation of $scanr\,(\oplus)\,e\,s$.

$$
\begin{aligned}
& scanr\,(\circledast)\,e\,(pure\,h \diamond s) \\
= \quad & \{\ \text{definition of } scanr\ \} \\
& e \prec (pure\,h \diamond s) \circledast scanr\,(\circledast)\,e\,(pure\,h \diamond s) \\
= \quad & \{\ \text{assumption: } h\,a \circledast b = a \oplus b\ \} \\
& e \prec s \oplus scanr\,(\circledast)\,e\,(pure\,h \diamond s)
\end{aligned}
$$

Scan flip The straightforward proof is left as an exercise to the reader.

Equality of left and right convolution We have noted in Section 4 that there are at least three different ways to define a convolution (here we fix the two operators $\cdot$ and $+$).

$$
\begin{aligned}
s \ltimes t &= s_0 \cdot t_0 \prec map\,(\cdot\, t_0)\,s' + s \ltimes t' \\
s \bowtie t &= s_0 \cdot t_0 \prec s_1 \cdot t_0 + s_0 \cdot t_1 \prec map\,(\cdot\, t_0)\,s'' + s' \bowtie t' + map\,(s_0 \cdot)\,t'' \\
s \rtimes t &= s_0 \cdot t_0 \prec s' \rtimes t + map\,(s_0 \cdot)\,t'
\end{aligned}
$$

In general they yield different results. However, if $+$ is associative, then $\ltimes = \bowtie = \rtimes$. To establish this equality, we first show the *shifting lemma*:

$$map\,(\cdot\, t_0)\,s' + s \bowtie t' = s' \bowtie t + map\,(s_0 \cdot)\,t' \ .$$

Let $f\,s\,t = map\,(\cdot t_0)\,s' + s \bowtie t'$ and $g\,s\,t = s' \bowtie t + map\,(s_0 \cdot)\,t'$, then

$$
\begin{aligned}
& f\,s\,t \\
= \quad & \{\ \text{definition of } f\ \} \\
& map\,(\cdot\, t_0)\,s' + s \bowtie t' \\
= \quad & \{\ \text{definition of } \bowtie \text{ and } +\ \} \\
& s_1 \cdot t_0 + s_0 \cdot t_1 \prec s_2 \cdot t_0 + (s_1 \cdot t_1 + s_0 \cdot t_2) \\
& \quad \prec map\,(\cdot\, t_0)\,s''' + (map\,(\cdot\, t_1)\,s'' + s' \bowtie t'' + map\,(s_0 \cdot)\,t''') \\
= \quad & \{\ + \text{ is associative and definition of } f\ \} \\
& s_1 \cdot t_0 + s_0 \cdot t_1 \prec s_2 \cdot t_0 + s_1 \cdot t_1 + s_0 \cdot t_2 \\
& \quad \prec map\,(\cdot\, t_0)\,s''' + f\,s'\,t' + map\,(s_0 \cdot)\,t''' \\
\subset \quad & \{\ x\,s\,t = \cdots \prec map\,(\cdot t_0)\,s''' + x\,s'\,t' + map\,(s_0 \cdot)\,t'''\ \} \\
& s_1 \cdot t_0 + s_0 \cdot t_1 \prec s_2 \cdot t_0 + s_1 \cdot t_1 + s_0 \cdot t_2 \\
& \quad \prec map\,(\cdot\, t_0)\,s''' + g\,s'\,t' + map\,(s_0 \cdot)\,t''' \\
= \quad & \{\ + \text{ is associative and definition of } g\ \} \\
& s_1 \cdot t_0 + s_0 \cdot t_1 \prec (s_2 \cdot t_0 + s_1 \cdot t_1) + s_0 \cdot t_2 \\
& \quad \prec (map\,(\cdot\, t_0)\,s''' + s'' \bowtie t' + map\,(s_1 \cdot)\,t'') + map\,(s_0 \cdot)\,t''' \\
= \quad & \{\ \text{definition of } \bowtie \text{ and } +\ \}
\end{aligned}
$$

$$
\begin{array}{ll}
& s' \bowtie t + map\,(s_0\,\cdot)\,t' \\
= & \quad \{ \text{ definition of } g \ \} \\
& g\ s\ t\ .
\end{array}
$$

Next we show that $\bowtie$ satisfies the recursion equation of $\rtimes$.

$$
\begin{array}{ll}
& s \bowtie t \\
= & \quad \{ \text{ definition of } \bowtie \ \} \\
& s_0 \cdot t_0\ \prec\ s_1 \cdot t_0 + s_0 \cdot t_1\ \prec\ map\,(\,\cdot\, t_0)\,s'' + s' \bowtie t' + map\,(s_0\,\cdot)\,t'' \\
= & \quad \{ \text{ shifting lemma } \} \\
& s_0 \cdot t_0\ \prec\ s_1 \cdot t_0 + s_0 \cdot t_1\ \prec\ s'' \bowtie t + map\,(s_1\,\cdot)\,t' + map\,(s_0\,\cdot)\,t'' \\
\subset & \quad \{\ x\ s\ t = \cdots \prec \cdots \prec x\ s''\ t + map\,(s_1\cdot)\,t' + map\,(s_0\cdot)\,t''\ \} \\
& s_0 \cdot t_0\ \prec\ s_1 \cdot t_0 + s_0 \cdot t_1\ \prec\ s'' \rtimes t + map\,(s_1\,\cdot)\,t' + map\,(s_0\,\cdot)\,t'' \\
= & \quad \{ \text{ definition of } \rtimes \text{ and } + \ \} \\
& s_0 \cdot t_0\ \prec\ s' \rtimes t + map\,(s_0\,\cdot)\,t' \\
= & \quad \{ \text{ definition of } \rtimes \ \} \\
& s \rtimes t
\end{array}
$$

An analogous argument shows that $\bowtie$ satisfies the recursion equation of $\ltimes$, which completes the proof.

Convolution fusion We show that $s \ltimes t = pure\,h \diamond convolute\,(\circledast)\,(\oplus)\,s\,t$ satisfies $x\ s\ t = s_0 \circledast t_0 \prec (s' \circledast pure\ t_0) \boxplus x\ s\ t'$, the defining equation of $convolute\,(\circledast)\,(\boxplus)\,s\,t$.

$$
\begin{array}{ll}
& pure\,h \diamond (s \ltimes t) \\
= & \quad \{ \text{ definition of } \ltimes \text{ and } \diamond \ \} \\
& h\,(s_0 \circledast t_0)\ \prec\ pure\,h \diamond (s' \circledast pure\ t_0 \oplus s \ltimes t') \\
= & \quad \{ \text{ assumption: } h\,(c_1 \oplus c_2) = h\ c_1 \boxplus h\ c_2 \text{ and } h\,(a \circledast b) = a \circledast b \ \} \\
& (s_0 \circledast t_0)\ \prec\ s' \circledast pure\ t_0 \boxplus pure\,h \diamond (s \ltimes t')
\end{array}
$$

Convolution functor fusion We demonstrate that the left convolution $s \ltimes t = convolute\,(\circledast)\,(\oplus)\,(pure\,h \diamond s)\,(pure\,k \diamond t)$ satisfies $x\ s\ t = s_0 \circledast t_0 \prec (s' \circledast pure\ t_0) \oplus x\ s\ t'$, the recursion equation of $convolute\,(\circledast)\,(\oplus)\,s\,t$.

$$
\begin{array}{ll}
& (pure\,h \diamond s) \ltimes (pure\,k \diamond t) \\
= & \quad \{ \text{ definition of } \ltimes \text{ and } \diamond \ \} \\
& (h\ s_0 \circledast k\ t_0)\ \prec\ (pure\,h \diamond s' \circledast pure\,(k\ t_0)) \oplus ((pure\,h \diamond s) \ltimes (pure\,k \diamond t')) \\
= & \quad \{ \text{ idiom laws and assumption: } h\ a \circledast k\ b = a \circledast b \ \} \\
& (s_0 \circledast t_0)\ \prec\ (s' \circledast pure\ t_0) \oplus ((pure\,h \diamond s) \ltimes (pure\,k \diamond t'))
\end{array}
$$

Convolution flip Again, we leave the straightforward proof as an exercise.

Convolution distributivity 1 This law is, in fact, a simple application of the two fusion laws.

$$
\begin{aligned}
& (c \circledast s) \bowtie t \\
= \quad & \{ \text{ definition of } \bowtie \} \\
& \mathit{convolute}\ (\circledast)\ (\oplus)\ (c \circledast s)\ t \\
= \quad & \{ \text{ functor fusion: } h\ a = c \circledast a \text{ and } k = \mathit{id} \} \\
& \mathit{convolute}\ (\lambda a\ b \to (c \circledast a) \circledast b)\ (\oplus)\ s\ t \\
= \quad & \{ \text{ fusion: } h\ x = c \circledast x,\ c \circledast (c_1 \oplus c_2) = (c \circledast c_1) \oplus (c \circledast c_2) \text{ and} \\
& \quad\ c \circledast (a \circledast b) = (c \circledast a) \circledast b \text{ by assumption } \} \\
& c \circledast \mathit{convolute}\ (\circledast)\ (\oplus)\ s\ t \\
= \quad & \{ \text{ definition of } \bowtie \} \\
& c \circledast (s \bowtie t)
\end{aligned}
$$

Convolution distributivity 2 This law can be shown using the unique-fixed-point principle.

$$
\begin{aligned}
& (s \boxplus t) \bowtie u \\
= \quad & \{ \text{ definition of } \bowtie \text{ and } \boxplus \} \\
& (s_0 \boxplus t_0) \circledast u_0 \ \prec\ ((s' \boxplus t') \circledast \mathit{pure}\ u_0) \oplus (s \boxplus t) \bowtie u' \\
= \quad & \{ \text{ assumption: } (a_1 \boxplus a_2) \circledast b = (a_1 \circledast b) \oplus (a_2 \circledast b) \} \\
& (s_0 \circledast u_0 \oplus t_0 \circledast u_0) \ \prec\ (s' \circledast \mathit{pure}\ u_0 \oplus t' \circledast \mathit{pure}\ u_0) \oplus (s \boxplus t) \bowtie u' \\
\subset \quad & \{ \ x\ s\ t\ u = \cdots \prec (s' \circledast \mathit{pure}\ u_0 \oplus t' \circledast \mathit{pure}\ u_0) \oplus x\ s\ t\ u' \ \} \\
& (s_0 \circledast u_0 \oplus t_0 \circledast u_0) \ \prec\ (s' \circledast \mathit{pure}\ u_0 \oplus t' \circledast \mathit{pure}\ u_0) \oplus (s \bowtie u') \oplus (t \bowtie u') \\
= \quad & \{ \text{ assumption: } \oplus \text{ is associative and commutative } \} \\
& (s_0 \circledast u_0 \oplus t_0 \circledast u_0) \ \prec\ (s' \circledast \mathit{pure}\ u_0 \oplus s \bowtie u') \oplus (t' \circledast \mathit{pure}\ u_0 \oplus t \bowtie u') \\
= \quad & \{ \text{ definition of } \oplus \} \\
& (s_0 \circledast u_0 \ \prec\ s' \circledast \mathit{pure}\ u_0 \oplus s \bowtie u') \oplus (t_0 \circledast u_0 \ \prec\ t' \circledast \mathit{pure}\ u_0 \oplus t \bowtie u') \\
= \quad & \{ \text{ definition of } \bowtie \} \\
& (s \bowtie u) \oplus (t \bowtie u)
\end{aligned}
$$

Convolution distributivity 3 Let $t = \mathit{scanr}\ (\boxplus)\ n\ s$, we show that $t \bowtie u$ satisfies $x = e \prec (s \bowtie u) \oplus x$, the recursion equation of $\mathit{scanr}\ (\oplus)\ e\ (s \bowtie u)$.

$$
\begin{aligned}
& t \bowtie u \\
= \quad & \{ \text{ definition of } \mathit{scanr} \text{ and } \bowtie \} \\
& n \circledast u_0 \ \prec\ (s \boxplus t) \bowtie u \oplus \mathit{pure}\ n \circledast u' \\
= \quad & \{ \text{ assumption: } n \circledast b = e \text{ and } a \oplus e = a \ \} \\
& e \ \prec\ (s \boxplus t) \bowtie u \\
= \quad & \{ \text{ convolution distributivity 2 } \} \\
& e \ \prec\ (s \bowtie u) \oplus (t \bowtie u)
\end{aligned}
$$