Reasoning about Codata

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. Programmers happily use induction to prove properties of recursive programs. To show properties of corecursive programs they employ coinduction, but perhaps less enthusiastically. Coinduction is often considered a rather low-level proof method, in particular, as it departs quite radically from equational reasoning. Corecursive programs are conveniently defined using recursion equations. Suitably restricted, these equations possess *unique solutions*. Uniqueness gives rise to a simple and attractive proof technique, which essentially brings equational reasoning to the coworld. We illustrate the approach using two major examples: streams and infinite binary trees. Both coinductive types exhibit a rich structure: they are applicative functors or idioms, and they can be seen as memo-tables or tabulations. We show that definitions and calculations benefit immensely from this additional structure.

1 Introduction

These lecture notes show how to use codata in modelling and programming and how to reason about codata, with the main focus on the latter. Codata is the dual of data, with an emphasis on observation rather than construction, and the indefinite rather than the finite.

Data is captured by inductive datatypes, whose elements can be constructed in a finite number of steps. Functional programming has been characterised as data-oriented programming: new datatypes are introduced with ease; elements of those types are analysed by recursive functions, conveniently defined by recursion equations; data constructors can be used on the right-hand side of equations to synthesise data and on the left-hand side to analyse data. Programmers happily use equational reasoning and induction to prove properties of recursive programs.

Dually, codata is captured by coinductive datatypes, whose elements can be deconstructed in a finite number of steps. Codata is synthesised using corecursive programs. To show properties of corecursive programs, programmers employ coinduction, but perhaps less enthusiastically. Coinduction is often considered a rather low-level proof method, especially, as it departs quite radically from equational reasoning. In these notes we introduce an alternative proof technique, based on unique fixed points, that remedies these problems. But we are skipping ahead.

Though data is dual to codata, it is not equally appreciated. For instance, in the seminal textbook on the "Algebra of Programming" [3] the authors denote a single paragraph to codata, remarking "We shall not have any use for such infinite data structures, however, and their discussion is therefore omitted." We hope to convince the reader that the notion of codata is equally valuable and that it has a lot to offer, both for the working programmer and for the working mathematician. For the programmer, it promises

– more elegant programs through a holistic or wholemeal approach,
– avoidance of case analysis,
– increased compositionality through separation of concerns.

For the mathematician, it promises

– more elegant proofs through a holistic or wholemeal approach,
– avoidance of index variables and subscripts,
– avoidance of case analysis and induction.

The simplest example of a coinductive type is the type of streams, where a stream is an infinite sequence of elements. In a lazy functional language, such as Haskell [31], streams are easy to define and many textbooks on Haskell reproduce the folklore examples of Fibonacci or Hamming numbers defined by recursion equations over streams. One has to be a bit careful in formulating a recursion equation, basically avoiding that the sequence defined swallows its own tail. However, if this care is exercised, the equation possesses a *unique solution*. Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are!

Let us illustrate the proof technique using a concrete example. Consider Figure 1, which displays a proof concerning a simple property of the Fibonacci numbers. The setting is very conventional, using a recurrence to define the Fibonacci numbers and an inductive proof to establish the property. The formalisation makes intensive use of the delimited Σ-notation. (Fourier introduced the notation in 1820, and it is reported to have taken the mathematical world by storm [14].) 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. In Figure 1, summation introduces the variable i, which is then used to index the Fibonacci sequence. Now, for comparison, let us re-develop the proof in a coinductive setting.

The Fibonacci sequence is defined by a set of recursion equations.

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

The definitions that make this work are introduced in Section 3. For the moment, it suffices to know that $\prec$ prepends an element to a stream and that the arithmetic operations are lifted point-wise to streams. Quite noticeable, index variables and subscripts are avoided by treating the sequence of Fibonacci numbers as a single entity.

The Fibonacci numbers are defined by the recurrence

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

The numbers satisfy a myriad of properties. For instance, if we add the first k Fibonacci numbers, we obtain $\mathcal{F}_{k+1} - 1$. Let us prove this simple fact. We show $\forall n \in \mathbb{N} \;.\; P(n)$, where P is given by

$$P(k) \quad :\Longleftrightarrow \quad \sum_{i=0}^{k-1} \mathcal{F}_i = \mathcal{F}_{k+1} - 1 \quad .$$

The proof proceeds by induction. *Basis:* $P(0)$.

$$\begin{aligned}& \sum_{i=0}^{-1} \mathcal{F}_i\\ =\;& \quad \{ \text{ empty sum } \}\\ & 0\\ =\;& \quad \{ \text{ arithmetic } \}\\ & 1 - 1\\ =\;& \quad \{ \text{ definition of } \mathcal{F}_1 \}\\ & \mathcal{F}_1 - 1\end{aligned}$$

Inductive step: $\forall n \in \mathbb{N} \;.\; P(n) \Longrightarrow P(n+1)$. Assume $P(n)$, then

$$\begin{aligned}& \sum_{i=0}^{n} \mathcal{F}_i\\ =\;& \quad \{ \text{ split sum } \}\\ & \left(\sum_{i=0}^{n-1} \mathcal{F}_i\right) + \mathcal{F}_n\\ =\;& \quad \{ \text{ ex hypothesi } P(n) \}\\ & \mathcal{F}_{n+1} - 1 + \mathcal{F}_n\\ =\;& \quad \{ \text{ arithmetic and definition of } \mathcal{F}_{n+2} \}\\ & \mathcal{F}_{n+2} - 1 \quad .\end{aligned}$$

Fig. 1. A famous recurrence and an inductive proof.

In the same spirit, summation is defined as a stream transformer or operator: it takes an input stream to the stream of its partial sums. Summation Σ is characterised by the following property.

$$\Sigma\ s = t \quad\Longleftrightarrow\quad t = 0 \prec s + t$$

The equivalence captures the fact that summation is the *unique solution* of the equation on the right-hand side.

The property of the Fibonacci numbers, adding the first k numbers yields $\mathcal{F}_{k+1} - 1$, is then captured by a simple stream equation: $\Sigma\ \mathit{fib} = \mathit{fib}' - 1$. Again, neither binders nor index variables are required. (By contrast, the corresponding statement in Figure 1 involves three binders: the universal quantifier introduces n, the abstraction defining the predicate P introduces k, and the delimited sum introduces i.) The proof is fairly straightforward. The characterisation of Σ leaves us with the task of showing $\mathit{fib}' - 1 = 0 \prec \mathit{fib} + \mathit{fib}' - 1$. We reason

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

The fairly voluminous, inductive argument in Figure 1 is replaced by a simple two-step calculation. It is the fact that summation is the *unique solution* of $\Sigma\ s = 0 \prec s + \Sigma\ s$ that makes the proof fly. In a nutshell, the proof method of unique fixed points brings equational reasoning to the coworld. Of course, it is by no means restricted to streams and can be used equally well to prove properties of infinite trees or the observational equivalence of instances of an abstract datatype.

Objectives The primary goal of these lecture notes is to familiarise you with the notion of codata. We shall make the ideas hinted above concrete using two major running examples: streams and infinite binary trees. At the end of the course, you should be able to capture sequences, iterative algorithms, infinite processes etc using recursion equations, and you should be able to prove properties using the unique fixed-point principle. Streams and infinite trees exhibit a rich structure: they are idioms and tabulations. We investigate these notions in considerable depth as they enable us to structure calculations more clearly.

Prerequisites We assume a basic knowledge of the functional programming language Haskell [31] — we shall make use of kinds, datatypes, type classes and lazy evaluation. Some knowledge of category theory is helpful, but not required.

Outline The rest of these notes are structured as follows. Section 2 reviews the notion of an applicative functor or idiom. Section 3 introduces the type of streams, our prime example of a coinductive datatype. Section 4 illustrates capturing recurrences using streams and investigates the relationship between streams and functions from the natural numbers. Section 5 applies the framework to finite calculus, the discrete counterpart of infinite calculus, where finite difference replaces the derivative and summation replaces integration. Section 6 introduces infinite trees, our second example of an inductive datatype, and discusses some applications. Both streams and infinite trees can be seen as tabulations or memo-tables. Section 7 investigates the notion of tabulation in more detail. Finally, Section 8 concludes. Related work is discussed at the end of each section, where appropriate.

2 Background: Idioms

Most definitions we encounter later on make use of operations lifted to streams or infinite trees. We obtain these liftings almost for free, as these datatypes are so-called *applicative functors* or *idioms* [27].

```
infixl 9 ⋄
class Idiom φ where
  pure :: α → φ α
  (⋄)  :: φ (α → β) → (φ α → φ β)
```

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 between structures. Consider as a simple example the *dual-core idiom*, which executes two programs in parallel.

```
data Pair α = Pair {outl :: α, outr :: α}
instance Idiom Pair where
  pure a = Pair a a
  u ⋄ v  = Pair ((outl u) (outl v)) ((outr u) (outr v))
```

The method *pure* duplicates its argument; idiomatic apply takes a pair of functions and a pair of arguments to a pair of results.

The type *Pair* can be seen as a very simple *container type*, which can accommodate exactly two elements. An alternative representation of a two-element container is a function from the Booleans: $Pair\ \alpha \cong Bool \to \alpha$. Generalising from *Bool*, we obtain the environment idiom '$\alpha \to$' — the type '$\alpha \to$' is actually a monad, but we shall not make use of the additional structure.

```
instance Idiom (α →) where
  pure a = λx → a
  f ⋄ g  = λx → (f x) (g x)
```

The idiom threads an environment, the argument x, through an idiomatic structure: *pure* discards the environment and $\diamond$ distributes it to its two arguments. Interestingly, *pure* is the combinator K and '$\diamond$' is the combinator S from combinatory logic [7]. The combinators were re-discovered in the 1970s to form the basis of an implementation technique for lazy functional languages [37].

Idioms abound, here are further examples of idioms and idiom transformers.

– The constant type constructor *Const A* with

$$Const\ \alpha\ \beta = \alpha$$

is an idiom if A is a monoid.
– The identity type constructor is an idiom.

$$Id\ \alpha = \alpha$$

– Idioms are closed under type composition.

$$(\phi \cdot \psi)\ \alpha = \phi\ (\psi\ \alpha)$$

– Idioms are closed under type pairing.

$$(\phi \mathbin{\dot{\times}} \psi)\ \alpha = (\phi\ \alpha, \psi\ \alpha)$$

The type constructor $\dot{\times}$ lifts pairing to parametric datatypes, type constructors of kind $\star \to \star$. The type *Pair* is isomorphic to $Id \mathbin{\dot{\times}} Id$.
– Every monad is an idiom — but not the other way round.

Exercise 1. Define suitable datatypes to represent the idioms and idiom transformers listed above. Then turn the types into instances of the *Idiom* class.

$$\begin{array}{l}
\mathbf{instance}\ (Monoid\ \alpha) \Rightarrow Idiom\ (Const\ \alpha)\\
\mathbf{instance}\ Idiom\ Id\\
\mathbf{instance}\ (Idiom\ \phi, Idiom\ \psi) \Rightarrow Idiom\ (\phi \cdot \psi)\\
\mathbf{instance}\ (Idiom\ \phi, Idiom\ \psi) \Rightarrow Idiom\ (\phi \mathbin{\dot{\times}} \psi)
\end{array}$$

(Section 6.2 defines the *Monoid* type class.) □

Using nested idiomatic applications, we can lift an arbitrary function pointwise to an idiomatic structure. Here are generic combinators for lifting unary and binary operations.

$$\begin{array}{ll}
map & :: (Idiom\ \phi) \Rightarrow (\alpha \to \beta) \to (\phi\ \alpha \to \phi\ \beta)\\
map\ f\ u & = pure\ f \diamond u\\
zip & :: (Idiom\ \phi) \Rightarrow (\alpha \to \beta \to \gamma) \to (\phi\ \alpha \to \phi\ \beta \to \phi\ \gamma)\\
zip\ g\ u\ v & = pure\ g \diamond u \diamond v
\end{array}$$

Using *zip* we can, for instance, lift pairing to idioms.

$$\begin{array}{l}
\mathbf{infixl}\ 6\ \star\\
(\star) :: (Idiom\ \phi) \Rightarrow \phi\ \alpha \to \phi\ \beta \to \phi\ (\alpha, \beta)\\
(\star) = zip\ (,)
\end{array}$$

The quizzical '(,)' is Haskell's pairing constructor.

For convenience and conciseness of notation, we lift the arithmetic operations to idioms. In Haskell, this is easily accomplished using the numeric type classes. Here is an excerpt of the code.[1]

```
instance (Idiom φ, Num α) ⇒ Num (φ α) where
  (+)            = zip (+)
  (−)            = zip (−)
  (∗ )           = zip (∗ )
  negate         = map negate  -- unary minus
  fromInteger i  = pure (fromInteger i)
```

We shall make intensive use of overloading, going beyond Haskell's predefined numeric classes. For instance, we also lift exponentiation u^v to idioms.

In these lecture notes, we mainly consider two idioms, streams and infinite trees. In both cases, the familiar arithmetic laws also hold for the lifted operators.

Speaking of laws, every instance of *Idiom* must satisfy four laws:

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

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

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

Every instance of Haskell's *Functor* class should satisfy these two laws (*map* is called *fmap* in *Functor*).

The interchange law allows us to swap pure and impure computations. This move possibly brings together pure computations, which can subsequently be merged using the homomorphism law. In fact, the idiom laws imply a normal form: every idiomatic expression can be rewritten into the form $\mathit{pure}\ f \diamond u_1 \diamond \cdots \diamond u_n$, a pure function applied to impure arguments. Put differently, applicative functors or idioms capture the notion of lifting: $\lambda u_1\ \cdots\ u_n \to \mathit{pure}\ f \diamond u_1 \diamond \cdots \diamond u_n$ is the lifted version of the $n$ary function f (assuming that f is curried). For instance, the environment idiom '$\alpha \to$' captures lifting operators to function spaces: $\mathit{zip}\ (+)\ f\ g = \mathit{pure}\ (+) \diamond f \diamond g = S\ (S\ (K\ (+))\ f)\ g = \lambda x \to f\ x + g\ x$.

Every structure comes equipped with structure-preserving maps; so do idioms: a polymorphic function $h :: \forall \alpha\ .\ \phi\ \alpha \to \psi\ \alpha$ is called an *idiom homomorphism* if and only if it preserves the idiomatic structure:

$$\begin{aligned}
&h\ (\mathit{pure}\ a) &&= \mathit{pure}\ a && (1)\\
&h\ (x \diamond y) &&= h\ x \diamond h\ y\ . && (2)
\end{aligned}$$

[1] 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.

The function $pure :: \forall\alpha \,.\, \alpha \to \phi\ \alpha$ itself is a homomorphism from the identity idiom Id to the idiom ϕ. Condition (2) for *pure* is equivalent to the homomorphism law, hence its name.

2.1 Summary and Related Work

Idioms capture the notion of lifting. Using *pure* and $\diamond$ we can, in particular, lift arithmetic operations point-wise to structures. The environment functor is the paradigmatic example of an idiom; it captures lifting operations point-wise to functions.

Categorically, idioms are *lax monoidal functors* [26] with strength. Programmatically, idioms arose as an interface for parsing combinators [34]. McBride and Paterson [27] introduced the notion to a wider audience. For the idioms we consider in these notes, the lifted operators satisfy the same properties as the 'unlifted' ones. This does, however, not hold in general [21].

3 Streams

A stream is an infinite sequence of elements. Here are some examples of (initial segments of) streams of natural numbers.

$$\begin{aligned}
&\langle 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, \ldots\rangle\\
&\langle 1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, \ldots\rangle\\
&\langle 0, 0, 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, \ldots\rangle\\
&\langle 0, 0, 2, 4, 8, 14, 24, 40, 66, 108, 176, 286, 464, 752, \ldots\rangle\\
&\langle 0, 1, 2, 6, 15, 40, 104, 273, 714, 1870, 4895, 12816, \ldots\rangle
\end{aligned}$$

Exercise 2. Describe the streams using natural language. □

Since Haskell is a lazy language, we can capture the type of streams as a datatype: $Stream\ \alpha$ is like Haskell's list data type $[\alpha]$, except that there is no base constructor so we cannot construct a finite stream. The *Stream* type is not an inductive type, but a *coinductive type*, whose semantics is given by a *final coalgebra* [1].

```
data Stream α = Cons {head :: α, tail :: Stream α}
infixr 5 ≺
(≺)     :: ∀α . α → Stream α → Stream α
a ≺ s = Cons a s
```

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.

Streams are an idiom, which means that we can effortlessly lift functions to streams:

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

Using this vocabulary we are already able to define the usual suspects: the natural numbers ($A001477$[2]), the factorial numbers ($A000142$), and the Fibonacci numbers ($A000045$).

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

Note that $\prec$ binds less tightly than $+$. For instance, $0 \prec \mathit{nat} + 1$ is grouped $0 \prec (\mathit{nat} + 1)$. The definitions capture invariants. For instance, incrementing the naturals by 1 and then prepending 0 yields again the naturals. Here is an attempt to visualise the invariant:

$$\begin{array}{cccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \cdots & & \mathit{nat} \\
 & + & + & + & + & + & + & + & + & + & + & \cdots & & + \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots & & 0 \prec 1 \\
\parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \cdots & & \parallel \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \cdots & & \mathit{nat} \ .
\end{array}$$

The table makes explicit that 1 in $\mathit{nat}+1$ is actually an infinite sequence of ones and that '+' zips two streams using addition.

The four 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

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

The table below visualises the definition.

$$\begin{array}{cccccccccccccc}
 & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & \cdots & & \mathit{fib} \\
 & + & + & + & + & + & + & + & + & + & + & \cdots & & + \\
0 & 1 & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & \cdots & & 0 \prec 1 \prec \mathit{fib} \\
\parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \parallel & \cdots & & \parallel \\
0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & \cdots & & \mathit{fib}
\end{array}$$

[2] Most if not all integer sequences defined in these lecture notes are recorded in Sloane's On-Line Encyclopedia of Integer Sequences [35]. Keys of the form *Annnnnn* refer to entries in that database.

The Fibonacci function is the folklore example of a function whose straightforward definition leads to a very inefficient program, see Exercise 9. By contrast, the stream definition, *fib*, does not suffer from this problem: to determine the nth element only $max\ \{n-1, 0\}$ additions are required.

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

$$
\begin{array}{l}
\gg\ \ fib \\
\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, .. \rangle \\
\gg\ \ nat * nat \\
\langle 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, .. \rangle \\
\gg\ \ fib'^2 - fib * fib'' \\
\langle 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, .. \rangle \\
\gg\ \ fib'^2 - fib * fib'' == (-1)^{nat} \\
True
\end{array}
$$

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 [22]. The session displays the actual output of the Haskell interpreter, generated automatically with lhs2TEX's active features.

Obviously, we cannot print out a sequence in full. The *Show* instance for *Stream* only displays the first n elements. Likewise, we cannot test two streams for equality: $==$ only checks whether the first n elements are equal. So, 'equality' is most useful for falsifying conjectures. For the purposes of these notes, n equals 15.

In Haskell, the same function can be defined in at least three different ways[3]. The same is true of sequences: here are three different variants of the stream of natural numbers — and there are more to come.

$$
\begin{array}{l}
nat = 0 \prec nat + 1 \\
nat = 0 \prec pure\ (+) \diamond nat \diamond pure\ 1 \\
nat = 0 \prec map\ (1+)\ nat
\end{array}
$$

The definitions can be shown equivalent using the idiom laws. As an example, the following calculation proves $nat + 1 = map\ (1+)\ nat$ — the most difficult part has been relegated to an exercise.

$$
\begin{array}{ll}
 & nat + 1 \\
= & \quad \{\ \text{definition of } + \text{ and } fromInteger\ \} \\
 & zip\ (+)\ nat\ (pure\ 1) \\
= & \quad \{\ \text{definition of } zip\ \} \\
 & pure\ (+) \diamond nat \diamond pure\ 1 \\
= & \quad \{\ \text{Exercise 3}\ \} \\
 & pure\ (+) \diamond pure\ 1 \diamond nat
\end{array}
$$

[3] See http://www.willamette.edu/~fruehr/haskell/evolution.html for an amusing illustration of this fact using the factorial function as an example.

$=$ { homomorphism law }

$\quad$ $\mathit{pure}\ (1+) \diamond \mathit{nat}$

$=$ { definition of map }

$\quad$ $\mathit{map}\ (1+)\ \mathit{nat}$

The proof exemplifies the typical style of reasoning: we transform the left-hand side into the right-hand side by repeatedly replacing equals by equals. The comments in curly braces justify the individual steps.

Exercise 3. Show $s + 1 = 1 + s$ using solely the idiom laws. (First, make sure that your understand why the laws are baptised ‘identity’, ‘composition’, ‘homomorphism’ and ‘interchange’. The text explains why.) Does lifted commutativity $s + t = t + s$ hold in every idiom? Conversely, what base-level identities can be lifted through any idiom? The paper “Lifting Operators and Laws” [21] answers these questions. □

3.1 Interleaving

Another important operator is *interleaving* of two streams.

infixr 5 $\curlyvee$
$(\curlyvee)\ \ :: \forall\alpha\ .\ \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha$
$s \curlyvee t = \mathit{head}\ s \prec t \curlyvee \mathit{tail}\ s$

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

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

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

Now that we have two, quite different 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 or of stream operators is one of our main activities in these lecture notes. However, we postpone a proof of $\mathit{nat} = \mathit{bin}$, until we have the prerequisites at hand.

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

The interleaving operator interacts nicely with lifting.

$$\mathit{pure}\ a \curlyvee \mathit{pure}\ a \quad = \mathit{pure}\ a$$
$$(s_1 \diamond s_2) \curlyvee (t_1 \diamond t_2) = (s_1 \curlyvee t_1) \diamond (s_2 \curlyvee t_2)$$

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

Property (3) 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.

$$\begin{array}{ccc} s_1 & \diamond & s_2 \\ & \curlyvee & \\ t_1 & \diamond & t_2 \end{array} = \begin{array}{ccc} s_1 & & s_2 \\ \curlyvee & \diamond & \curlyvee \\ t_1 & & t_2 \end{array} \qquad\qquad \frac{s_1 \mid s_2}{t_1 \mid t_2} = \begin{array}{c|c} \dfrac{s_1}{t_1} & \dfrac{s_2}{t_2} \end{array}$$

The two-dimensional arrangement is originally due to Hoare, the catchy name is due to Bird [4]. The geometrical interpretation can be further emphasised by writing the two operators $\mid$ and $-$, like on the right-hand side [11].

Exercise 4. Try to capture the sequences listed in the introduction to Section 3 using stream equations. For the latter two puzzles experiment a little with the Fibonacci sequences *fib* and *fib′*. *Hint:* Sloane's On-Line Encyclopedia of Integer Sequences lists most integer sequences one can think of. $\square$

Exercise 5. Turn the following verbal descriptions into streams.

1. The sequence of natural numbers divisible by 3.
2. The sequence of natural numbers *not* divisible by 3.
3. The sequence of cubes.
4. The sequence of all finite binary strings:

$$\langle [\,], [0], [1], [0,0], [1,0], [0,1], [1,1], [0,0,0], [1,0,0], [0,1,0], \ldots \rangle \ .$$

5. The bit-reversed positive numbers:

$$\langle 1, 2, 3, 4, 6, 5, 7, 8, 12, 10, 14, 9, 13, 11, 15, \ldots \rangle \ .$$

The order of all bits, except the most significant one, in the binary expansion of n is reversed. $\square$

3.2 Definitions and Proofs

Not every legal Haskell definition of type *Stream* τ actually defines a stream. Two simple counterexamples are $s_1 = \mathit{tail}\ s_1$ and $s_2 = \mathit{head}\ s_2 \prec \mathit{tail}\ s_2$. Both of them loop in Haskell; when viewed as stream equations they are ambiguous.[4] In fact, they admit infinitely many solutions: every constant stream is a solution of the first equation, every stream is a solution of the second one. This situation is undesirable from both a practical and a theoretical standpoint. Fortunately, it is not hard to restrict the *syntactic* form of equations so that they possess *unique solutions*. We insist that equations adhere to the following form:

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

[4] 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 $\perp$, $a_0 \prec \perp$, $a_0 \prec a_1 \prec \perp$ and so forth. We simply ignore this extra complication here.

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

For a formal account of these requirements, we refer the interested reader to the paper "Streams and Unique Fixed Points" [18], which contains a constructive proof that admissible equations indeed have unique solutions. Looking back, we find that the definitions we have encountered so far, including those of *pure*, $\diamond$ and $\curlyvee$, are admissible.

If $x = \varphi\ x$ is an admissible equation, we denote its unique solution by *fix* φ. (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.

$$\mathit{fix}\ \varphi = s \iff \varphi\ s = s$$

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

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

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

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

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

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

$$\begin{array}{ll} = & \{\text{ arithmetic }\} \\ & 0 \prec (2 * \mathit{nat} \curlyvee 2 * \mathit{nat} + 1) + 1 \end{array}$$

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

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

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

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 important 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*: $\mathit{fib}'^2 - \mathit{fib} * \mathit{fib}'' = (-1)^{\mathit{nat}}$.

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

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

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

The proof is straightforward involving only bureaucratic steps.

$$\begin{array}{ll}
 & f\ a\ b\ c\ d \\
= & \quad \{\ \text{definition of } f\ \} \\
 & (a \diamond b) \curlyvee (c \diamond d) \\
= & \quad \{\ \text{definition of } \diamond \text{ and definition of } \curlyvee\ \} \\
 & \mathit{head}\ a \diamond \mathit{head}\ b \prec (c \diamond d) \curlyvee (\mathit{tail}\ a \diamond \mathit{tail}\ b) \\
= & \quad \{\ \text{definition of } f\ \} \\
 & \mathit{head}\ a \diamond \mathit{head}\ b \prec f\ c\ d\ (\mathit{tail}\ a)\ (\mathit{tail}\ b) \\
\subset & \quad \{\ x\ s_1\ s_2\ t_1\ t_2 = \mathit{head}\ s_1 \diamond \mathit{head}\ s_2 \prec x\ t_1\ t_2\ (\mathit{tail}\ s_1)\ (\mathit{tail}\ s_2)\ \} \\
 & \mathit{head}\ a \diamond \mathit{head}\ b \prec g\ c\ d\ (\mathit{tail}\ a)\ (\mathit{tail}\ b) \\
= & \quad \{\ \text{definition of } g\ \} \\
 & \mathit{head}\ a \diamond \mathit{head}\ b \prec (c \curlyvee \mathit{tail}\ a) \diamond (d \curlyvee \mathit{tail}\ b) \\
= & \quad \{\ \text{definition of } \diamond \text{ and definition of } \curlyvee\ \} \\
 & (a \curlyvee c) \diamond (b \curlyvee d) \\
= & \quad \{\ \text{definition of } g\ \} \\
 & g\ a\ b\ c\ d
\end{array}$$

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.

Exercise 6. The parametric stream *from* is given by

$$\begin{array}{l}
\mathit{from}\ \ :: \mathit{Nat} \to \mathit{Stream\ Nat} \\
\mathit{from}\ n = n \prec \mathit{from}\ (n+1)\ .
\end{array}$$

Show that $\mathit{from}\ n + \mathit{pure}\ k = \mathit{from}\ (n + k)$ in at least two different ways. □

Exercise 7. Prove the other idiom laws using the unique fixed-point principle. □

3.3 Recursion and Iteration

The stream *nat* is constructed by repeatedly mapping a function over a stream. We can capture this recursion scheme using a combinator, which implements *recursive* or *top-down* constructions.

```
$recurse :: \forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to Stream\ \alpha)$
$recurse\ f\ a = s$
  **where** $s = a \prec map\ f\ s$
```

So, $nat = recurse\ (+1)\ 0$.

Alternatively, we can build a stream by repeatedly applying a given function to a given initial seed. The combinator *iterate* captures this *iterative* or *bottom-up* construction.

```
$iterate :: \forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to Stream\ \alpha)$
$iterate\ f\ a$         $= loop\ a$
  **where** $loop\ x = x \prec loop\ (f\ x)$
```

So, $iterate\ (+1)\ 0$ is yet another definition of the naturals. The type α can be seen as a type of states and the resulting stream as an enumeration of the state space. One could argue that *iterate* is more natural than *recurse*. This intuition is backed up by the fact that $map\ g \cdot iterate\ f$ is the *unfold* or *anamorphism* of the *Stream* codatatype. Very briefly, the unfold is characterised by the following *universal property*.

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

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

The functions *iterate* and *recurse* satisfy an important fusion law, which amounts to the free theorem of $\forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to Stream\ \alpha)$.

$$\begin{array}{c} map\ h \cdot recurse\ f_1 = recurse\ f_2 \cdot h \\ \Uparrow \\ h \cdot f_1 = f_2 \cdot h \\ \Downarrow \\ map\ h \cdot iterate\ f_1 = iterate\ f_2 \cdot h \end{array}$$

Here is a unique fixed-point proof of the first fusion law.

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

$$
\begin{aligned}
&= \quad \{ \text{ assumption: } h \cdot f_1 = f_2 \cdot h \ \} \\
&\quad h\ a \prec \mathit{iterate}\ f_2\ (f_2\ (h\ a)) \\
&= \quad \{ \text{ definition of } \mathit{iterate} \ \} \\
&\quad \mathit{iterate}\ f_2\ (h\ a)
\end{aligned}
$$

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

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

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

We show that $\mathit{iterate}\ f\ a$ is the unique solution of $x = a \prec \mathit{map}\ f\ x$.

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

Exercise 8. When are $\mathit{iterate}\ f\ a$ and $\mathit{iterate}\ g\ b$ equal? As a simple example, consider *iterate* (`["hi"]`++) [ ] and *iterate* (++`["hi"]`) [ ]. Can you find sufficient and necessary conditions? □

3.4 Summary and Related Work

The type of streams is a simple example of a coinductive datatype. The type has the structure of an idiom, which allows us to lift arbitrary functions to streams. Streams can be conveniently defined using recursion equations. Admissible equations have unique solutions, which is the basis of the unique fixed-point principle. For streams, recursive and iterative constructions coincide.

This section is based on the paper "Streams and Unique Fixed Points" [18], which in turns draws from Rutten's work on stream calculus [32, 33]. 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.) However, Rutten relies on coinduction as the main proof technique.

Various proof methods for corecursive programs are discussed by Gibbons and Hutton [13]. Interestingly, the technique of unique fixed points is not among them. Unique fixed-point proofs are closely related to the principle of *guarded induction* [6], which goes back to the work on process algebra [30]. 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,

$$nat = 1 \prec nat + 1$$

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

4 Application: Recurrences

A *recurrence* or recurrence relation is a set of equations that defines a sequence, a function from the natural numbers. It typically provides a boundary value and an equation for the general value in terms of earlier ones, see Figure 1 for an example. Using $\prec$ and $\curlyvee$ we can often capture a function from the natural numbers by a single equation. Though functions from the naturals and streams are in a one-to-one correspondence, a stream is usually easier to manipulate. Before we consider concrete examples, we first explore tabulation in more depth.

4.1 Tabulation

In Section 2 we have noted in passing by that $Pair$s are in a one-to-one correspondence to functions from the Booleans. Streams enjoy an analogous property, they are in a one-to-one correspondence to functions from the natural numbers:

$$Stream\ \alpha \cong Nat \to \alpha\ ,$$

where the inductive datatype Nat is given by the Pseudo-Haskell definition

$$\mathbf{data}\ Nat = 0 \mid Nat + 1\ .$$

(Strictly speaking, this defines the unary numbers or Peano numerals, which *represent* the natural numbers.) A stream can be seen as the tabulation of a function from the natural numbers. Conversely, a function of type $Nat \to \alpha$ can be implemented by looking up a memo-table. Here are the functions that witness the isomorphism.

$$\begin{aligned}
&tabulate :: \forall \alpha\ .\ (Nat \to \alpha) \to Stream\ \alpha \\
&tabulate\ f = f\ 0 \prec tabulate\ (f \cdot (+1))
\end{aligned}$$

$$\begin{aligned}
&lookup && :: \forall \alpha\ .\ Stream\ \alpha \to (Nat \to \alpha) \\
&lookup\ s\ 0 && = head\ s \\
&lookup\ s\ (n+1) && = lookup\ (tail\ s)\ n
\end{aligned}$$

The functions *lookup* and *tabulate* are mutually inverse

$$\begin{aligned} &\mathit{lookup} \cdot \mathit{tabulate} = \mathit{id} \\ &\mathit{tabulate} \cdot \mathit{lookup} = \mathit{id} \ , \end{aligned}$$

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 environment idiom $\tau \rightarrow$. The laws are somewhat easier to memorise, if we write them in a point-wise style.

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

A simple consequence of the first law is $\mathit{tabulate}\ f = \mathit{map}\ f\ (\mathit{tabulate}\ \mathit{id})$. Hence, *tabulate* is fully determined by the image of the identity, which is the stream of natural numbers (see below). So, one way of tabulating an arbitrary function is to map the function over the stream of natural numbers.

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. As an example, the recurrence below defines $\mathcal{T}_n$, the minimum number of moves to solve the Tower of Hanoï problem for n discs.

$$\begin{aligned} \mathcal{T}_0 &= 0 \\ \mathcal{T}_{n+1} &= 2 * \mathcal{T}_n + 1 \end{aligned}$$

It is not hard to see that the stream defined

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

implements the same sequence. In general, the recurrence $a_0 = k$ and $a_{n+1} = f(a_n)$ is captured by the stream equation $s = k \prec \mathit{map}\ f\ s$, or more succinctly by $\mathit{recurse}\ f\ k$. 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*.

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

Catamorphisms are dual to anamorphisms, enjoying a dual characterisation.

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

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

Now, tabulating *fold s z* gives *recurse s z* (hence the name of the combinator). The proof of this fact makes crucial use of *tabulate*'s naturality property.

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

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

$$\mathit{tabulate}\ (\mathit{fold}\ f\ k) = \mathit{recurse}\ f\ k = \mathit{iterate}\ f\ k$$

Using the reflection law, this furthermore implies that *nat* is the tabulation of the identity function:

$$\begin{aligned}
& \mathit{tabulate}\ \mathit{id} \\
= & \quad \{ \text{ reflection law: } \mathit{fold}\ (+1)\ 0 = \mathit{id} \ \} \\
& \mathit{tabulate}\ (\mathit{fold}\ (+1)\ 0) \\
= & \quad \{ \text{ see above } \} \\
& \mathit{recurse}\ (+1)\ 0 \ .
\end{aligned}$$

Exercise 9. The naîve implementation of the Fibonacci numbers is horribly inefficient.

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

But, can you make this more precise? For instance, how many additions are performed in order to compute $\mathcal{F}_n$, or, how many recursive calls are made? Express your findings as stream equations. Then try to relate the two streams to examples we have encountered so far. □

Exercise 10. Determine the number of binary strings of some given length that do not contain adjacent zeros. Again, first try to come up with a system of recursion equations and then try to relate the streams to known examples. □

We already know that *fib* tabulates the Fibonacci function $\mathcal{F}$. To sharpen our calculational skills let us try to derive the stream definition from the recurrence given in Exercise 9. The recurrence does not fit the simple scheme discussed above, so we have to start afresh. The calculations are effortless if we make use of the fact that *tabulate* is an idiom homomorphism between the environment idiom $Nat \rightarrow$ and *Stream*.

$$\begin{aligned} &\mathit{tabulate}\ (\mathit{pure}\ a) = \mathit{pure}\ a \\ &\mathit{tabulate}\ (x \diamond y) \quad = \mathit{tabulate}\ x \diamond \mathit{tabulate}\ y \end{aligned}$$

Since tabulation and look-up are inverses, this implies that *lookup* is an idiom homomorphism, as well.

$$\begin{aligned} &\mathit{lookup}\ (\mathit{pure}\ a) = \mathit{pure}\ a \\ &\mathit{lookup}\ (x \diamond y) \quad = \mathit{lookup}\ x \diamond \mathit{lookup}\ y \end{aligned}$$

Returning to the problem of tabulating $\mathcal{F}$, it is useful to rewrite the last equation of $\mathcal{F}$ in a point-free style: $\mathcal{F} \cdot (+2) = \mathcal{F} + \mathcal{F} \cdot (+1)$. The right-hand side makes use of addition lifted to the environment idiom.

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

The only non-trivial step is the second but last one, which uses $\mathit{tabulate}\ (f + g) = \mathit{tabulate}\ f + \mathit{tabulate}\ g$, 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. The resulting equation

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

is equivalent to the definitions given in Section 3.

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. The next exercise aims to illustrate this point by deriving an efficient *iterative* implementation of $\mathcal{F}$.

Exercise 11. Turn the Fibonacci sequence

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

into an iterative form: $\mathit{map}\ g\ (\mathit{iterate}\ f\ a) = \mathit{unfold}\ g\ f$. There are, at least, two approaches:

– Pair *fib* and *fib′*

$$fib \star fib' \ ,$$

where $(\star) = zip\ (,)$ turns a pair of streams into a stream of pairs, see Section 2.
– Use the fact that the tails of *fib* are linear combinations of *fib* and *fib′*.

$$i * fib + j * fib'$$

Hint: Express the tail of $i * fib + j * fib'$ as a linear combination of *fib* and *fib′* and then capture the corecursion using *unfold*.

Try to relate the two approaches. □

Exercise 12. Turn the equation

$$x = (a \prec map\ f\ x) + s$$

into an iterative form. *Hint:* You may find the function $tails = iterate\ tail$ useful. Try pairing x with $tails\ s$. As an aside, *tails* is the comultiplication of the comonad *Stream*. □

Exercise 13. Complete the proof that $tabulate\ f = map\ f\ nat$. □

4.2 Bit-fiddling

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 $s = k \prec map\ f\ s \curlyvee map\ g\ s$. We have already seen an instance of this scheme in Section 3.

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

Here, the parameters of the general scheme are instantiated by $k = 0$, $f\ n = 2 * n + 1$ and $g\ n = 2 * n + 2$. In other words, a is the identity and *bin* is its tabulation: $bin = tabulate\ id = nat$. For the positive numbers, we can derive a similar equation.

$$\begin{aligned} & bin + 1 \\ = & \quad \{ \text{ definition of } bin \ \} \\ & (0 \prec 2 * bin + 1 \curlyvee 2 * bin + 2) + 1 \\ = & \quad \{ \text{ abide law and arithmetic } \} \\ & 1 \prec 2 * (bin + 1) \curlyvee 2 * (bin + 1) + 1 \end{aligned}$$

We have calculated the definition below.

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

Since the equation has a unique solution, we know that $bin' = bin + 1 = nat + 1 = nat'$. The definition of bin' captures a well-known recipe for generating the positive numbers in binary: start with 1, then repeatedly shift the bits to the right (lsb first), placing a 0 or a 1 in the left-most, least significant position.

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

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

The most significant bit of 1 is 1, the most significant bit of both $2 * \mathit{bin}'$ and $2 * \mathit{bin}' + 1$ is $2 * \mathit{msb}$.

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

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

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

Exercise 14. Prove this claim. *Hint:* Let s be a solution of $x = x \curlyvee x + 1$. Show that $s - \mathit{pure}\ (\mathit{head}\ s)$ satisfies the definition of ones. □

Let us inspect the sequences.

$$\begin{aligned} &\gg\ \ \mathit{msb} \\ &\langle 1, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, .. \rangle \\ &\gg\ \ \mathit{bin}' - \mathit{msb} \\ &\langle 0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, .. \rangle \\ &\gg\ \ \mathit{ones} \\ &\langle 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, .. \rangle \end{aligned}$$

The sequence $\mathit{bin}' - \mathit{msb}$ ($A053645$) exhibits a nice pattern; it describes the distance to the largest power of two at most bin'. In binary, this amounts to removing the most significant bit.

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

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

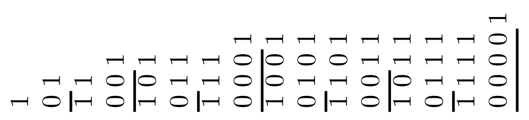
(The form of the equation does not quite meet the requirements. We allow ourselves some liberty, as a simple unfolding turns it into an admissible form: $\mathit{carry} = 0 \prec \mathit{carry} + 1 \curlyvee 0$. The unfolding works as long as the first argument of $\curlyvee$ is a sequence defined elsewhere.) Let us peek at some values.

$$\begin{aligned} &\gg\ \ \mathit{carry} \\ &\langle 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, .. \rangle \end{aligned}$$

The sequence gives the exponent of the largest power of two dividing bin', that is, the number of leading zeros in the binary representation (lsb first). In other words, it specifies the running time of the binary increment. The table below illustrates the relationship.

1
$\underline{0}\;1$
$1\;1$
$\underline{0\;0}\;1$
$1\;0\;1$
$\underline{0}\;1\;1$
$1\;1\;1$
$\underline{0\;0\;0}\;1$
$1\;0\;0\;1$
$\underline{0}\;1\;0\;1$
$1\;1\;0\;1$
$\underline{0\;0}\;1\;1$
$1\;0\;1\;1$
$\underline{0}\;1\;1\;1$
$1\;1\;1\;1$
$\underline{0\;0\;0\;0}\;1$

For emphasis, prefixes of zeros are underlined. There is also an intriguing connection to infinite binary trees. If we turn the table by 90° to the left, we can see the correspondence more clearly.



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 so-called *sideways tree*, an infinite tree, which has no root, but extends infinitely upwards.

Exercise 15. Prove that the sequence given by $a_0 = k$, $a_{2n+1} = f(a_n)$ and $a_{2n+2} = g(a_n)$ corresponds to the stream $s = k \prec map\ f\ s \curlyvee map\ g\ s$. *Hint:* Use $nat = bin$ and Exercise 13. $\square$

4.3 Summary and Related Work

A stream tabulates a function from the naturals. Tabulation and look-up are idiom isomorphisms between the environment idiom $Nat \rightarrow$ and $Stream$. Using $\prec$ and $\curlyvee$ we can capture 'unary' and 'binary' recurrences.

The section is also based on "Streams and Unique Fixed Points" [18].

5 Application: Finite Calculus

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

5.1 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.

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

Here are some examples ($A000079$, $A094267$, $A003215$, $A033428$).

$\gg\ \ \mathit{\Delta}\ 2^{nat}$
$\langle 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, ..\rangle$
$\gg\ \ \mathit{\Delta}\ \mathit{carry}$
$\langle 1, -1, 2, -2, 1, -1, 3, -3, 1, -1, 2, -2, 1, -1, 4, ..\rangle$
$\gg\ \ \mathit{\Delta}\ \mathit{nat}^3$
$\langle 1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, ..\rangle$
$\gg\ \ 3 * \mathit{nat}^2$
$\langle 0, 3, 12, 27, 48, 75, 108, 147, 192, 243, 300, 363, 432, 507, 588, ..\rangle$

Infinite calculus has an attractive rule for the derivative of a power: $(x^{n+1})\frac{d}{dx} = (n+1)x^n$. Unfortunately, the last two examples show that finite difference does not interact well with ordinary powers: $\mathit{\Delta}\ \mathit{nat}^3$ is by no means $3 * \mathit{nat}^2$. An alternative power that blends nicely with $\mathit{\Delta}$ is the *falling factorial power* defined

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

As usual, we lift the operator to streams: $s^{\underline{n}} = \mathit{map}\ (\lambda x \to x^{\underline{n}})\ s$. The new power satisfies $s * (s-1)^{\underline{n}} = s^{\underline{n+1}} = s^{\underline{n}} * (s - n)$. Hence, finite calculus has a handy rule to match the one for the derivative of a power.

$$\mathit{\Delta}\ (\mathit{nat}^{\underline{n+1}}) = (\mathit{pure}\ n + 1) * \mathit{nat}^{\underline{n}}$$

The proof is entirely straightforward.

$$\begin{aligned}&\quad \mathit{\Delta}\ (\mathit{nat}^{\underline{n+1}})\\ =&\quad \{\ \text{definition of } \mathit{\Delta}\ \}\\ &\quad \mathit{tail}\ (\mathit{nat}^{\underline{n+1}}) - \mathit{nat}^{\underline{n+1}}\\ =&\quad \{\ \text{definition of } \mathit{nat}\ \}\\ &\quad (\mathit{nat} + 1)^{\underline{n+1}} - \mathit{nat}^{\underline{n+1}}\\ =&\quad \{\ s * (s-1)^{\underline{n}} = s^{\underline{n+1}} = s^{\underline{n}} * (s-n)\ \}\\ &\quad (\mathit{nat} + 1) * \mathit{nat}^{\underline{n}} - \mathit{nat}^{\underline{n}} * (\mathit{nat} - \mathit{pure}\ n)\\ =&\quad \{\ \text{arithmetic}\ \}\\ &\quad (\mathit{pure}\ n + 1) * \mathit{nat}^{\underline{n}}\end{aligned}$$

Table 1. Converting between powers and falling factorial powers.

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

The following session shows that falling factorial powers behave as expected.

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

One can convert mechanically between powers and falling factorial powers using Stirling numbers [14]. The details are beyond the scope of these lecture notes. For reference, Table 1 displays the correspondence up to the fourth power.

Table 2 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 $\Delta\ 2^{nat} = 2^{nat}$. The product rule is similar to the product rule of infinite calculus except for an occurrence of *tail* on the right-hand side.

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

Exercise 16. The product rule $\Delta\ (s * t) = s * \Delta\ t + \Delta\ s * tail\ t$ is somewhat asymmetric. Can you find a symmetric variant? Prove it correct. □

5.2 Summation

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

$$\Delta\ (\Sigma\ s) = s$$

Table 2. Laws for finite difference (c and n are constant streams).

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

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

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 [8], turning the higher-order equation $\Sigma\ s = 0 \prec s + \Sigma\ s$ defining Σ into a first-order equation $t = 0 \prec s + t$ defining $t = \Sigma\ s$ with s fixed. The firstification of the definition enables sharing of computations as illustrated below.

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

Here are some applications of summation ($A004520$, $A000290$, $A011371$, $0 \prec A000330$ and $0 \prec A036799$).

$\gg\ \Sigma\,(0 \curlyvee 1)$
$\langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, ..\rangle$
$\gg\ \Sigma\,(2 * \mathit{nat} + 1)$
$\langle 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, ..\rangle$
$\gg\ \Sigma\ \mathit{carry}$
$\langle 0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, ..\rangle$
$\gg\ \Sigma\ \mathit{nat}^2$
$\langle 0, 0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506, 650, 819, ..\rangle$
$\gg\ \Sigma\,(\mathit{nat} * 2^{\mathit{nat}})$
$\langle 0, 0, 2, 10, 34, 98, 258, 642, 1538, 3586, 8194, 18434, 40962, 90114, ..\rangle$

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 $\Sigma\ s = t$. The step makes use of the fact that $\Sigma\ s$ is the unique solution of its defining equation. For example, $\Sigma\ 1 = \mathit{nat}$ as $\mathit{nat} = 0 \prec \mathit{nat} + 1$,

$\Sigma\ (2 * nat + 1) = nat^2$ as $nat^2 = 0 \prec nat^2 + 2 * nat + 1$, and $\Sigma\ (1 \prec \mathit{fib}) = \mathit{fib}$ as $\mathit{fib} = 0 \prec (1 \prec \mathit{fib}) + \mathit{fib}$. This is *summation by happenstance*.

Of course, if we already know the sum, we can use the definition to verify our conjecture. As an example, let us prove $\Sigma\ \mathit{fib}'^2 = \mathit{fib} * \mathit{fib}'$ — the elements of this sequence are known as the *golden rectangle numbers* (A001654).

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

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

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

$$t = \Delta\ s \iff \Sigma\ t = s - pure\ (head\ s)$$

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

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

For instance, $\Sigma\ 2^{nat} = 2^{nat} - 1$, since $2^{nat} = \Delta\ 2^{nat}$ and $head\ (2^{nat}) = 1$.

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

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

Table 3. Laws for summation (c and n are constant streams).

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

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.

Here is an alternative proof of $\Sigma\ \mathit{fib} = \mathit{fib}' - 1$ that uses some of the laws in Table 3.

$$\begin{aligned}
&\mathit{fib} = 0 \prec \mathit{fib} + (1 \prec \mathit{fib})\\
\iff &\quad \{\ \text{summation by happenstance}\ \}\\
&\Sigma\ (1 \prec \mathit{fib}) = \mathit{fib}\\
\iff &\quad \{\ \text{summation law}\ \}\\
&0 \prec 1 + \Sigma\ \mathit{fib} = \mathit{fib}\\
\implies &\quad \{\ s_1 = s_2 \implies \mathit{tail}\ s_1 = \mathit{tail}\ s_2\ \}\\
&1 + \Sigma\ \mathit{fib} = \mathit{fib}'\\
\iff &\quad \{\ \text{arithmetic}\ \}\\
&\Sigma\ \mathit{fib} = \mathit{fib}' - 1
\end{aligned}$$

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

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

Calculating the summation of a product, say, $\Sigma\ (\mathit{nat} * 2^{\mathit{nat}})$ 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 expression above, there is an obvious candidate: c^{nat}. Let us see how it goes.

$$\begin{aligned}
& \Sigma\ (nat * 2^{nat}) \\
= & \quad \{\ \Delta\ 2^{nat} = 2^{nat}\ \} \\
& \Sigma\ (nat * \Delta\ 2^{nat}) \\
= & \quad \{\ \text{summation by parts}\ \} \\
& nat * 2^{nat} - \Sigma\ (\Delta\ nat * tail\ 2^{nat}) \\
= & \quad \{\ \Delta\ nat = 1, \text{ and definition of } nat\ \} \\
& nat * 2^{nat} - 2 * \Sigma\ 2^{nat} \\
= & \quad \{\ \text{summation law}\ \} \\
& nat * 2^{nat} - 2 * (2^{nat} - 1) \\
= & \quad \{\ \text{arithmetic}\ \} \\
& (nat - 2) * 2^{nat} + 2
\end{aligned}$$

As a final example, let us tackle a sum that involves the interleaving operator: $\Sigma\ carry$ (A011371). The sum is important, as it determines the amortised running time of the binary increment. Let us experiment (A011371, A000120).

$\gg\ \Sigma\ carry$
$\langle 0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, ..\rangle$
$\gg\ nat - \Sigma\ carry$
$\langle 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, ..\rangle$
$\gg\ nat \geqslant \Sigma\ carry$
True

We observe that the sum is always at most *nat*, which would imply that the amortised running time, $\Sigma\ carry\ /\ nat$, is constant. This 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. 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.) Now, we can invoke the *summation by happenstance* rule.

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

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

Exercise 17. Derive the sum rule $\Sigma\ (s + t) = \Sigma\ s + \Sigma\ t$ from the sum rule $\Delta\ (s + t) = \Delta\ s + \Delta\ t$ using the Fundamental Theorem. □

Exercise 18. Work out $\Sigma\ nat^3$ using the summation laws and the correspondence between powers and falling factorial powers. □

Exercise 19. Here is an alternative definition of Σ

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

which uses a second-order fixed point. The code implements the naîve way of summing: the ith element is computed using i additions not reusing any previous results. Prove that the two definitions of Σ are equivalent. □

Exercise 20. Generalise the derivation of $\Sigma\ (nat * 2^{nat})$ to $\Sigma\ (nat * c^{nat})$, where c is a constant stream. □

5.3 Summary and Related Work

Finite calculus serves as an elegant application of corecursive definitions and the unique fixed-point principle. Index variables and subscripts are avoided by taking a holistic view treating sequences as a single entity.

Again, most of the material has been taken from "Streams and Unique Fixed Points" [18]. Two further corecursion schemes for stream-generating functions, scans and convolutions, are introduced in a recent paper [20]. The paper also presents a novel proof of Moessner's theorem. Scans generalise summation, convolution generalises the product of power series. Very briefly, a sequence of numbers, $a_0,\ a_1,\ a_2\ \ldots$, can be used to represent a power series, $a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots$, in some formal variable z. In fact, many papers on streams emphasise the 'power series' view of streams, most notably, [24, 28, 29]. Interestingly, the papers use lazy lists to represent streams, resulting in additional code to cover the empty list.

6 Infinite Trees

Streams are a lovely example of a coinductive datatype, but there is, of course, the danger of overspecialisation. To counteract this danger, we look at a second example in this section: infinite binary trees (trees for short). Trees are in many respects similar to streams, but, as we shall see, there are also some important differences. In a nutshell, streams relate to trees in the same way as unary numbers (Peano numerals) relate to binary numbers (bit strings).

Figure 2 displays the first five levels of an infinite binary tree that contains all the naturals. It is a *fractal* object, in the sense that parts of it are similar to the whole. The tree can be transformed into its left subtree by first doubling and then incrementing the elements (which is why the subtree contains exactly the odd numbers). To obtain the right subtree, we have to interchange the order of the two steps: the elements are first incremented and then doubled (which explains why the subtree contains exactly the even numbers greater than 0). This description can be nicely captured by a *corecursive* definition:

$$nat = Node\ 0\ ((2 * nat) + 1)\ (2 * (nat + 1))\ .$$

(We re-use some of the identifiers introduced in the previous sections to denote infinite trees. In case of ambiguity, we employ qualified names.) As to be expected, the operations are lifted point-wise to trees. Like streams, trees are an idiom. But we are skipping ahead.

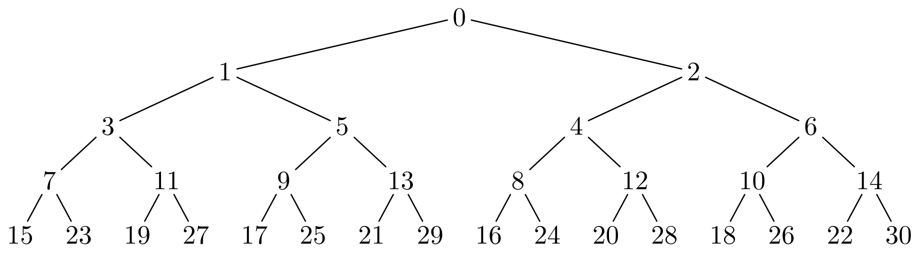


Fig. 2. The tree of natural numbers.

The type *Tree* α is a *coinductive datatype*. Its definition is similar to the standard textbook definition of binary trees, except that there is no base constructor, so we cannot build a finite tree.

$$\textbf{data}\ Tree\ \alpha = Node\ \{root :: \alpha, left :: Tree\ \alpha, right :: Tree\ \alpha\}$$

Trees are constructed using *Node*. They are destructed using *root*, which yields the label of the root node, and *left* and *right*, which return the left and the right subtree, respectively.

As mentioned above, trees are an idiom, which means that we can effortlessly lift functions to trees:

```
instance Idiom Tree where
  pure a = t where t = Node a t t
  t ⋄ u  = Node ((root t) (root u)) (left t ⋄ left u) (right t ⋄ right u) .
```

Recall that *pure*, *map* and *zip* and the arithmetic operations are overloaded to work with an arbitrary idiom. By virtue of the above instance declaration we can use them for infinite trees, as well. Here is variation of *nat* that captures a well-known recipe for generating the positive numbers: start with 1, then repeatedly double the number, adding 0 or 1 to the result.

$$pos = Node\ 1\ (2 * pos + 0)\ (2 * pos + 1)$$

6.1 Definitions and Proofs

As for streams, we can restrict the *syntactic* form of equations so that they possess *unique solutions*. As *admissible* equation is of the form

$$x\ x_1\ \ldots\ x_n = \mathit{Node}\ a\ l\ r\ \ ,$$

where x is an identifier of type $\tau_1 \to \cdots \to \tau_n \to \mathit{Tree}\ \tau$, a is a constant expression of type τ, and l and r are expressions of type $\mathit{Tree}\ \tau$ possibly referring to x or some other tree operator in the case of mutual recursion. The expressions may use $\mathit{root}\ x_i$, $\mathit{left}\ x_i$ or $\mathit{right}\ x_i$ provided x_i is of the right type. Apart from that, no other uses of the projection functions are permitted.

Admissible equations have unique solutions. Hence we can adopt the unique fixed-point principle to prove that two infinite trees are equal: if they satisfy the same recursion equation, then they are. The proof of $\mathit{nat} + 1 = \mathit{pos}$ below illustrates the principle: we show that $\mathit{nat} + 1$ satisfies the recursion equation of pos.

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

Like for streams, the familiar arithmetic laws also hold for the lifted operators.

Exercise 21. There are essentially two ways of generating an infinite tree that contains all bit strings (lists of zeros and ones).

$$\begin{aligned} \mathit{lbits} &= \mathit{Node}\ [\,]\ (\mathit{map}\ ([0]{+\!\!+})\ \mathit{lbits}\,)\ (\mathit{map}\ ([1]{+\!\!+})\ \mathit{lbits}\,) \\ \mathit{rbits} &= \mathit{Node}\ [\,]\ (\mathit{map}\ ({+\!\!+}[0])\ \mathit{rbits})\ (\mathit{map}\ ({+\!\!+}[1])\ \mathit{rbits}) \end{aligned}$$

Show that $\mathit{map}\ \mathit{reverse}\ \mathit{lbits} = \mathit{rbits}$ using the unique fixed-point principle. How are lbits and rbits related to nat and pos? □

6.2 Recursion and Iteration

The combinator *recurse* captures *recursive* or *top-down* tree constructions; the functions f and g are repeatedly mapped over the whole tree:

$$\begin{aligned} & \mathit{recurse} :: \forall \alpha\ .\ (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \mathit{Tree}\ \alpha) \\ & \mathit{recurse}\ f\ g\ a = t \\ & \quad \textbf{where}\ t \quad = \mathit{Node}\ a\ (\mathit{map}\ f\ t)\ (\mathit{map}\ g\ t)\ \ . \end{aligned}$$

Thus, an alternative definition of nat is $\mathit{recurse}\ (\lambda n \to 2{*}n{+}1)\ (\lambda n \to 2{*}n{+}2)\ 0$.

We can also construct a tree in an *iterative* or *bottom-up* fashion; the functions f and g are repeatedly applied to the given initial seed a:

$$\begin{aligned} & \mathit{iterate} :: \forall \alpha\ .\ (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \mathit{Tree}\ \alpha) \\ & \mathit{iterate}\ f\ g\ a = \mathit{loop}\ a \\ & \quad \textbf{where}\ \mathit{loop}\ x = \mathit{Node}\ x\ (\mathit{loop}\ (f\ x))\ (\mathit{loop}\ (g\ x))\ \ . \end{aligned}$$

The type α can be seen as a type of states and the infinite tree as an enumeration of the state space.

We have overloaded the names *recurse* and *iterate* to denote operations both on streams and on trees. The abuse of language is justified as both sets of operations satisfy similar laws. For instance, $map\ h \cdot iterate\ f\ g$ is the *unfold* of the *Tree* codatatype. Furthermore, both *recurse* and *iterate* satisfy a *fusion* law:

$$\begin{array}{c} map\ h \cdot recurse\ f_1\ g_1 = recurse\ f_2\ g_2 \cdot h \\ \Uparrow \\ h \cdot f_1 = f_2 \cdot h \wedge h \cdot g_1 = g_2 \cdot h \\ \Downarrow \\ map\ h \cdot iterate\ f_1\ g_1 = iterate\ f_2\ g_2 \cdot h\ . \end{array}$$

Exercise 22. Prove the fusion laws, and then use fusion to give an alternative proof of $map\ reverse\ lbits = rbits$. □

How are $recurse\ f\ g\ a$ and $iterate\ f\ g\ a$ related? Contrary to the situation for streams, they are certainly not equal. Consider Figure 3, which displays the trees $recurse\ ([0]+\!\!+)\ ([1]+\!\!+)\ [\,]$ and $iterate\ ([0]+\!\!+)\ ([1]+\!\!+)\ [\,]$. Since f and g are applied in different orders — inside out and outside in — each level of $recurse\ f\ g\ a$ is the *bit-reversal permutation* of the corresponding level of $iterate\ f\ g\ a$. For brevity's sake, one tree is called the *bit-reversal permutation tree* of the other. Exercises 21 and 22 explain the term bit-reversal permutation: a bit string can be seen as a path into an infinite tree — this is the central theme of Section 6.3 — following the reversed path leads to the permuted element.

Now, can we transform an instance of *recurse* into an instance of *iterate*? Yes, if the two functions are pre- or post-multiplications of elements of some given *monoid*. Let us introduce a suitable type class:

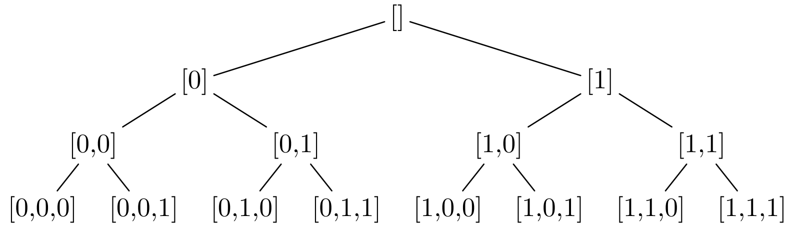
$$\begin{array}{l} \mathbf{infixr}\ 5\ \circ \\ \mathbf{class}\ Monoid\ \alpha\ \mathbf{where} \\ \quad \epsilon \quad :: \alpha \\ \quad (\circ) :: \alpha \to \alpha \to \alpha\ . \end{array}$$

The *recursion-iteration lemma* then states

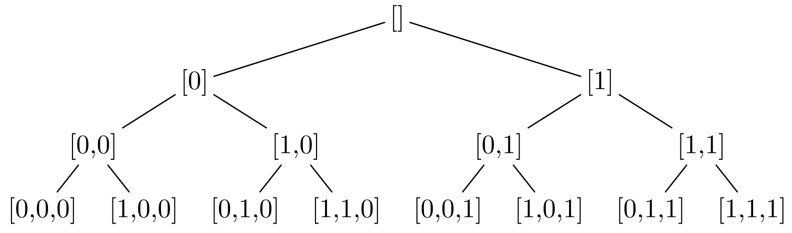
$$recurse\ (a\circ)\ (b\circ)\ \epsilon = iterate\ (\circ a)\ (\circ b)\ \epsilon\ , \qquad (3)$$

where a and b are elements of some monoid $(M, \circ, \epsilon)$. To establish the lemma, we show that $iterate\ (\circ a)\ (\circ b)\ \epsilon$ satisfies the defining equation of $recurse\ (a\circ)\ (b\circ)\ \epsilon$, that is $t = Node\ \epsilon\ (map\ (a\circ)\ t)\ (map\ (b\circ)\ t)$:

$$\begin{array}{ll} & iterate\ (\circ a)\ (\circ b)\ \epsilon \\ = & \quad \{\ \text{definition of}\ iterate\ \} \\ & Node\ \epsilon\ (iterate\ (\circ a)\ (\circ b)\ (\epsilon \circ a))\ (iterate\ (\circ a)\ (\circ b)\ (\epsilon \circ b)) \\ = & \quad \{\ \epsilon \circ x = x = x \circ \epsilon\ \} \\ & Node\ \epsilon\ (iterate\ (\circ a)\ (\circ b)\ (a \circ \epsilon))\ (iterate\ (\circ a)\ (\circ b)\ (b \circ \epsilon)) \end{array}$$



(a) *recurse* ([0]++) ([1]++) []



(b) *iterate* ([0]++) ([1]++) []

Fig. 3. A tree that contains all bit strings and its bit-reversal permutation tree.

$$
\begin{aligned}
= &\quad \{ \text{ fusion: } (x\circ) \cdot (\circ y) = (\circ y) \cdot (x\circ) \ \} \\
&\mathit{Node}\ \epsilon\ (\mathit{map}\ (a\circ)\ (\mathit{iterate}\ (\circ a)\ (\circ b)\ \epsilon))\ (\mathit{map}\ (b\circ)\ (\mathit{iterate}\ (\circ a)\ (\circ b)\ \epsilon))\ .
\end{aligned}
$$

As an example, *recurse* ([0]++) ([1]++) [] = *iterate* (++[0]) (++[1]) []; both expressions construct the infinite tree of all bit strings, shown in Figure 3 (a).

At first sight, it seems that the applicability of the lemma is somewhat hampered by the requirement on the form of the two arguments. However, since *endomorphisms*, functions of type $\tau \to \tau$ for some τ, form a monoid, we can easily rewrite an arbitrary instance of *recurse* into the required form ($\diamond$ is function application below, the 'apply' of the identity idiom):

$$
\begin{aligned}
&\mathit{recurse}\ f\ g\ a \\
= &\quad \{ \text{ identity } \} \\
&\mathit{recurse}\ f\ g\ ((\diamond a)\ \mathit{id}) \\
= &\quad \{ \text{ fusion: } (\diamond x) \cdot (f\ \cdot) = f \cdot (\diamond x) \ \} \\
&\mathit{map}\ (\diamond a)\ (\mathit{recurse}\ (f\ \cdot)\ (g\ \cdot)\ \mathit{id}) \\
= &\quad \{ \text{ definition of } \mathit{map} \ \} \\
&\mathit{pure}\ (\diamond a) \diamond \mathit{recurse}\ (f\ \cdot)\ (g\ \cdot)\ \mathit{id} \\
= &\quad \{ \text{ interchange law } \} \\
&\mathit{recurse}\ (f\ \cdot)\ (g\ \cdot)\ \mathit{id} \diamond \mathit{pure}\ a
\end{aligned}
$$

$$\begin{aligned} &= \quad \{ \text{recursion-iteration lemma} \} \\ &\quad \mathit{iterate}\ (\cdot\ f)\ (\cdot\ g)\ \mathit{id} \diamond \mathit{pure}\ a \ . \end{aligned}$$

(Note that we cannot 'un-fuse' the final expression.) This transformation turns a recursive construction into an iterative one, where functions serve as the internal state. One could argue the resulting construction is not really iterative (after all, the functions involved create a chain of closures). However, often we can provide a concrete representation of these functions, for instance, as a matrix, see the paper "The Bird tree" [19] for an example along these lines.

6.3 Tabulation

Like streams, infinite trees are a tabulation: they are in a one-to-one correspondence to functions from the binary numbers:

$$\mathit{Tree}\ \alpha \cong \mathit{Bin} \to \alpha \ ,$$

where the datatype *Bin* is given by

$$\mathbf{data}\ \mathit{Bin} = \mathit{Nil} \mid \mathit{One}\ \mathit{Bin} \mid \mathit{Two}\ \mathit{Bin} \ .$$

(The type is isomorphic to the type of lists of bits, that we have used in the previous section. For the purposes of this section, a tailor-made datatype is preferable.) A tree can be seen as the tabulation of a function from the binary numbers. Conversely, a function of type $\mathit{Bin} \to \alpha$ can be implemented by looking up a memo-table. Here are the functions that witness the isomorphism.

$$\begin{aligned} &\mathit{tabulate} :: \forall \alpha\ .\ (\mathit{Bin} \to \alpha) \to \mathit{Tree}\ \alpha \\ &\mathit{tabulate}\ f = \mathit{Node}\ (f\ \mathit{Nil})\ (\mathit{tabulate}\ (f \cdot \mathit{One}))\ (\mathit{tabulate}\ (f \cdot \mathit{Two})) \end{aligned}$$

$$\begin{aligned} &\mathit{lookup} && :: \forall \alpha\ .\ \mathit{Tree}\ \alpha \to (\mathit{Bin} \to \alpha) \\ &\mathit{lookup}\ t\ \mathit{Nil} && = \mathit{root}\ t \\ &\mathit{lookup}\ t\ (\mathit{One}\ b) && = \mathit{lookup}\ (\mathit{left}\ \ t)\ b \\ &\mathit{lookup}\ t\ (\mathit{Two}\ b) && = \mathit{lookup}\ (\mathit{right}\ t)\ b \end{aligned}$$

Again, we have overloaded the names to also denote operations on trees. (Exercise 24 asks you to capture the overloading using type classes.) This is justified as the new functions satisfy exactly the same properties as the old ones: they are mutually inverse and they are natural in the value type α. Tabulating the identity yields the infinite tree of binary numbers:

$$\begin{aligned} &\quad \mathit{tabulate}\ \mathit{id} \\ &= \quad \{ \text{definition of } \mathit{tabulate} \} \\ &\quad \mathit{Node}\ \mathit{Nil}\ (\mathit{tabulate}\ \mathit{One})\ (\mathit{tabulate}\ \mathit{Two}) \\ &= \quad \{ \text{naturality of } \mathit{tabulate} \} \\ &\quad \mathit{Node}\ \mathit{Nil}\ (\mathit{map}\ \mathit{One}\ (\mathit{tabulate}\ \mathit{id}))\ (\mathit{map}\ \mathit{Two}\ (\mathit{tabulate}\ \mathit{id})) \ . \end{aligned}$$

Consequently, $\mathit{tabulate}\ \mathit{id} = \mathit{bin}$ where *bin* is given by

$$\mathit{bin} = \mathit{Node}\ \mathit{Nil}\ (\mathit{map}\ \mathit{One}\ \mathit{bin})\ (\mathit{map}\ \mathit{Two}\ \mathit{bin}) \ .$$

Modulo the representation of binary numbers, *bin* is equivalent to *nat*, *lbits* and *rbits*.

In Section 4.1 we have discussed at length how to tabulate functions. For variety, we consider the opposite problem here, namely, how to turn an infinite tree into a recursive or iterative algorithm. To this end, we require the *fold* or *catamorphism* for the inductive datatype *Bin*.

$$\begin{aligned}
&\mathit{fold} :: \forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to (\mathit{Bin} \to \alpha)\\
&\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\ \mathit{Nil} &&= \mathit{nil}\\
&\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\ (\mathit{One}\ b) &&= \mathit{one}\ (\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\ b)\\
&\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\ (\mathit{Two}\ b) &&= \mathit{two}\ (\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\ b)
\end{aligned}$$

The naming of identifiers makes explicit that a fold replaces constructors by functions. Like for streams, tabulation relates *fold* to *recurse*. Conversely, untabulating a recursive construction yields a fold.

$$\begin{aligned}
\mathit{tabulate}\ (\mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}) &= \mathit{recurse}\ \mathit{one}\ \mathit{two}\ \mathit{nil}\\
\mathit{lookup}\ (\mathit{recurse}\ \mathit{one}\ \mathit{two}\ \mathit{nil}) &= \mathit{fold}\ \mathit{one}\ \mathit{two}\ \mathit{nil}
\end{aligned}$$

As an example, let us derive a recursive algorithm for *fast exponentiation*. Let c be a constant. We seek an efficient implementation of $(\mathit{pure}\ c)^{\mathit{nat}}$. Let us calculate.

$$\begin{aligned}
&\quad (\mathit{pure}\ c)^{\mathit{nat}}\\
&= \quad \{\ \text{definition of } \mathit{nat}\ \}\\
&\quad \mathit{Node}\ c^0\ (\mathit{pure}\ c)^{(2*\mathit{nat})+1}\ (\mathit{pure}\ c)^{2*(\mathit{nat}+1)}\\
&= \quad \{\ \text{laws of exponentials}\ \}\\
&\quad \mathit{Node}\ 1\ (((\mathit{pure}\ c)^{\mathit{nat}})^2 * \mathit{pure}\ c)\ ((\mathit{pure}\ c)^{\mathit{nat}} * \mathit{pure}\ c)^2
\end{aligned}$$

Consequently, $(\mathit{pure}\ c)^{\mathit{nat}} = \mathit{recurse}\ (\lambda x \to x^2 * c)\ (\lambda x \to (x * c)^2)\ 1$ or equivalently $\mathit{lookup}\ (\mathit{pure}\ c)^{\mathit{nat}} = \mathit{fold}\ (\lambda x \to x^2 * c)\ (\lambda x \to (x * c)^2)\ 1$. The derivation can be readily generalised to an arbitrary monoid. For instance, 2×2 matrices with matrix multiplication form a monoid, so the program can be used to calculate the Fibonacci numbers in logarithmic time:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} \mathcal{F}_{n-1} & \mathcal{F}_n \\ \mathcal{F}_n & \mathcal{F}_{n+1} \end{pmatrix}\ ,$$

with $\mathcal{F}_{-1} = 1$.

Can we also derive an iterative algorithm for fast exponentiation? There are, at least, two choices: we can use the recursion-iteration lemma (see the previous section) and go higher-order, or we can use the bit-reversal permutation lemma (introduced below) and do some bit-fiddling.

Very briefly, the first approach yields

$$\begin{aligned}
&\quad \mathit{lookup}\ (\mathit{recurse}\ f\ g\ a)\ n\\
&= \quad \{\ \text{Section 6.2}\ \}
\end{aligned}$$

$$\begin{aligned}
& \mathit{lookup}\ (\mathit{iterate}\ (\cdot\ f)\ (\cdot\ g)\ \mathit{id} \diamond \mathit{pure}\ a)\ n \\
= \quad & \{\ \mathit{lookup} \text{ is an idiom homomorphism }\} \\
& (\mathit{lookup}\ (\mathit{iterate}\ (\cdot\ f)\ (\cdot\ g)\ \mathit{id}) \diamond \mathit{lookup}\ (\mathit{pure}\ a))\ n \\
= \quad & \{\ \text{environment idiom }\} \\
& \mathit{lookup}\ (\mathit{iterate}\ (\cdot\ f)\ (\cdot\ g)\ \mathit{id})\ n\ a \ .
\end{aligned}$$

We build up an infinite tree of functions, look-up the function at position n and then apply it to a.

For the second approach, recall that *recurse* $f\ g\ a$ is the bit-reversal permutation tree of *iterate* $f\ g\ a$. One way to formulate this relationship is via *lookup*:

$$\mathit{lookup}\ (\mathit{recurse}\ f\ g\ a) = \mathit{lookup}\ (\mathit{iterate}\ f\ g\ a) \cdot \mathit{reverse}\ , \tag{4}$$

where *reverse* mirrors a binary number. The proof of Equation (4), dubbed the *bit-reversal permutation lemma*, proceeds smoothly if we turn *Bin* into an instance of *Num*, *Enum* and *Monoid*. Then *tabulate* can be written more perspicuously as

$$\mathit{tabulate}\ f = \mathit{Node}\ (f\ \epsilon)\ (\mathit{tabulate}\ (f \cdot (1\circ)))\ (\mathit{tabulate}\ (f \cdot (2\circ)))\ .$$

Equation (4) calls for an inductive proof. We can circumvent induction by applying *tabulate* to both sides of the equation. Let $h = \mathit{lookup}\ (\mathit{iterate}\ f\ g\ a)$, we show that $\mathit{tabulate}\ (h \cdot \mathit{reverse})$ satisfies the recursion equation of *recurse* $f\ g\ a$.

$$\begin{aligned}
& \mathit{tabulate}\ (h \cdot \mathit{reverse}) \\
= \quad & \{\ \text{definition of } \mathit{tabulate} \text{ and } (h \cdot \mathit{reverse})\ \epsilon = a\ \} \\
& \mathit{Node}\ a\ (\mathit{tabulate}\ (h \cdot \mathit{reverse} \cdot (1\circ)))\ (\mathit{tabulate}\ (h \cdot \mathit{reverse} \cdot (2\circ))) \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& \mathit{Node}\ a\ (\mathit{tabulate}\ (h \cdot (\circ 1) \cdot \mathit{reverse}))\ (\mathit{tabulate}\ (h \cdot (\circ 2) \cdot \mathit{reverse})) \\
= \quad & \{\ \text{proof obligation }\} \\
& \mathit{Node}\ a\ (\mathit{tabulate}\ (f \cdot h \cdot \mathit{reverse}))\ (\mathit{tabulate}\ (g \cdot h \cdot \mathit{reverse})) \\
= \quad & \{\ \text{naturality of tabulate }\} \\
& \mathit{Node}\ a\ (\mathit{map}\ f\ (\mathit{tabulate}\ (h \cdot \mathit{reverse})))\ (\mathit{map}\ g\ (\mathit{tabulate}\ (h \cdot \mathit{reverse})))
\end{aligned}$$

It remains to discard the proof obligations $h \cdot (\circ 1) = f \cdot h$ and $h \cdot (\circ 2) = g \cdot h$, which capture the fact that the most significant bit determines the function applied in the last iteration. Again, to avoid an inductive proof we show the equivalent $\mathit{tabulate}\ (h \cdot (\circ 1)) = \mathit{map}\ f\ (\mathit{iterate}\ f\ g\ a)$. Let $k = \mathit{lookup} \cdot \mathit{iterate}\ f\ g$, then

$$\begin{aligned}
& \mathit{tabulate}\ (k\ a \cdot (\circ 1)) \\
= \quad & \{\ \text{definition of tabulate and } (k\ a \cdot (\circ 1))\ \epsilon = f\ a\ \} \\
& \mathit{Node}\ (f\ a)\ (\mathit{tabulate}\ (k\ a \cdot (\circ 1) \cdot (1\circ)))\ (\mathit{tabulate}\ (k\ a \cdot (\circ 1) \cdot (2\circ)))
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ monoids: } (x\circ)\cdot(\circ y) = (\circ y)\cdot(x\circ) \ \}\\
&\quad \mathit{Node}\ (f\ a)\ \epsilon)\ (\mathit{tabulate}\ (k\ a\cdot(1\circ)\cdot(\circ 1)))\ (\mathit{tabulate}\ (k\ a\cdot(2\circ)\cdot(\circ 1)))\\
&= \quad \{ \text{ definition of } k \ \}\\
&\quad \mathit{Node}\ (f\ a)\ (\mathit{tabulate}\ (k\ (f\ a)\cdot(\circ 1)))\ (\mathit{tabulate}\ (k\ (g\ a)\cdot(\circ 1)))\\
&\subset \quad \{ \ x\ a = \mathit{Node}\ (f\ a)\ (x\ (f\ a))\ (x\ (g\ a)) \text{ has a unique solution } \}\\
&\quad \mathit{Node}\ (f\ a)\ (\mathit{map}\ f\ (\mathit{iterate}\ f\ g\ (f\ a)))\ (\mathit{map}\ f\ (\mathit{iterate}\ f\ g\ (g\ a)))\\
&= \quad \{ \text{ definition of } \mathit{map} \text{ and } \mathit{iterate} \ \}\\
&\quad \mathit{map}\ f\ (\mathit{iterate}\ f\ g\ a) \ .
\end{aligned}
$$

The proof of $h \cdot (\circ 2) = g \cdot h$ proceeds analogously.

It remains to deforest the intermediate data structure created by *iterate*. If we 'un-tabulate' *iterate*, setting $\mathit{loop}\ f\ g = \mathit{lookup} \cdot \mathit{iterate}\ f\ g$, we obtain an iterative or *tail-recursive* function, which can be seen as the counterpart of *foldl* for binary numbers.

$$
\begin{aligned}
&\mathit{loop} && :: \forall\alpha\ .\ (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to (\mathit{Bin} \to \alpha)\\
&\mathit{loop}\ f\ g\ a\ \mathit{Nil} && = a\\
&\mathit{loop}\ f\ g\ a\ (\mathit{One}\ b) && = \mathit{loop}\ f\ g\ (f\ a)\ b\\
&\mathit{loop}\ f\ g\ a\ (\mathit{Two}\ b) && = \mathit{loop}\ f\ g\ (g\ a)\ b
\end{aligned}
$$

To summarise, we have derived two iterative algorithms for fast exponentiation:

$$
\begin{aligned}
&\mathit{power}\ c\ n = \mathit{loop}\ (\cdot\ (\lambda x \to x^2 * c))\ (\cdot\ (\lambda x \to (x * c)^2))\ \mathit{id}\ n\ 1\\
&\mathit{power}\ c\ n = \mathit{loop}\ (\lambda x \to x^2 * c)\ (\lambda x \to (x * c)^2)\ 1\ (\mathit{reverse}\ n) \ .
\end{aligned}
$$

The latter function is called the square-and-multiply algorithm or binary exponentiation. In fact, it corresponds to a variant known as the *Montgomery powering ladder*. (Exponentiation is used in most public-key crypto systems. The algorithm above is less vulnerable to attacks, since in each step a squaring and a multiplication is performed.)

Exercise 23. The datatype *Bin* implements the 1-2 *number system*, a variant of the binary system, which uses the digits $\{1, 2\}$, rather than $\{0, 1\}$. (A distinct advantage of this number system is that each natural number has a unique representation.) The functions

$$
\begin{aligned}
&\mathit{toNat} :: \mathit{Bin} \to \mathit{Nat}\\
&\mathit{toNat} = \mathit{fold}\ (\lambda n \to 2 * n + 1)\ (\lambda n \to 2 * n + 2)\ 0\\
&\mathit{toBin} :: \mathit{Nat} \to \mathit{Bin}\\
&\mathit{toBin} = \mathit{fold}\ \mathit{succ}\ \mathit{zero}
\end{aligned}
$$

convert between unary numbers and these binary numbers. Implement $\mathit{zero} :: \mathit{Bin}$ and $\mathit{succ} :: \mathit{Bin} \to \mathit{Bin}$. Show that *toNat* and *toBin* are inverses. □

Exercise 24. Capture *lookup* and *tabulate* using a type class. Since two types are involved, the type of keys and the type of tables, you need either multi-parameter type classes or type families. □

6.4 Infinite Trees and Sequences

The type of natural numbers is isomorphic to the type of binary numbers: $\mathit{Nat} \cong \mathit{Bin}$. This implies that the type of streams is isomorphic to the type of infinite binary trees:

$$\mathit{Stream}\ \alpha \cong \mathit{Tree}\ \alpha\ .$$

We obtain the *canonical isomorphism* for converting a stream into a tree and vice versa by following the aforementioned chain of isomorphisms. Let $\mathit{toNat} :: \mathit{Bin} \to \mathit{Nat}$ and $\mathit{toBin} :: \mathit{Nat} \to \mathit{Bin}$ be the isomorphisms witnessing $\mathit{Nat} \cong \mathit{Bin}$, see Exercise 23. Then $\mathit{stream} :: \mathit{Tree}\ \alpha \to \mathit{Stream}\ \alpha$ and $\mathit{tree} :: \mathit{Stream}\ \alpha \to \mathit{Tree}\ \alpha$ are given by the following diagram.

$$\begin{tikzcd}
\mathit{Stream}\ \alpha \arrow[r, shift left, tail, "\mathit{lookup}"] \arrow[d, shift right, tail, "\mathit{tree}"'] & \mathit{Nat} \to \alpha \arrow[l, shift left, tail, "\mathit{tabulate}"] \arrow[d, shift left, tail, "\mathit{toNat} \to \mathit{id}"] \\
\mathit{Tree}\ \alpha \arrow[r, shift left, tail, "\mathit{lookup}"] \arrow[u, shift right, tail, "\mathit{stream}"'] & \mathit{Bin} \to \alpha \arrow[l, shift left, tail, "\mathit{tabulate}"] \arrow[u, shift left, tail, "\mathit{toBin} \to \mathit{id}"]
\end{tikzcd} \qquad (5)$$

Here, $f \to g$ is the mapping function of the function space type constructor defined $(f \to g)\ h = g \cdot h \cdot f$.

The interactive session below shows that *stream* converts the tree of natural numbers, see Figure 2, into the stream of natural numbers.

$$\begin{aligned}
&\gg\ \ \mathit{stream}\ (\mathit{recurse}\ (\lambda n \to 2 * n + 1)\ (\lambda n \to 2 * n + 2)\ 0) \\
&\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ..\rangle \\
&\gg\ \ \mathit{stream}\ (\mathit{iterate}\ (\lambda n \to 2 * n + 1)\ (\lambda n \to 2 * n + 2)\ 0) \\
&\langle 0, 1, 2, 3, 5, 4, 6, 7, 11, 9, 13, 8, 12, 10, 14, ..\rangle
\end{aligned}$$

It is important to note that *stream* does *not* list the elements level-wise from left to right, rather, it involves a bit-reversal permutation. Hence, streaming the iterative construction yields the permuted list of naturals ($0 \prec A081241$).

For calculational purposes, it is useful to derive versions of *stream* and *tree* that do not involve number systems. For streaming, the idea is to define functions that mimic the projection functions *head* and *tail*. Clearly, *root* is the counterpart of *head*, the counterpart of *tail* is *chop* given by

$$\begin{aligned}
&\mathit{chop} \quad :: \forall \alpha\ .\ \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha \\
&\mathit{chop}\ t = \mathit{Node}\ (\mathit{root}\ (\mathit{left}\ t))\ (\mathit{right}\ t)\ (\mathit{chop}\ (\mathit{left}\ t))\ \ .
\end{aligned}$$

The name indicates that it chops off the root of a given tree, interleaving the two subtrees. (The definition is reminiscent of $\curlyvee$, this is not a coincidence, see below.) The projection functions are related by

$$\mathit{root} = \mathit{head} \cdot \mathit{stream} \tag{6}$$

$$\mathit{stream} \cdot \mathit{chop} = \mathit{tail} \cdot \mathit{stream}\ \ . \tag{7}$$

In other words, *stream* is a so-called *representation changer* [23]. Given these prerequisites, it is a simple exercise to derive *stream*.

$$\begin{aligned}
& \mathit{stream}\ t \\
= \quad & \{ \text{ extensionality: } s = \mathit{head}\ s \prec \mathit{tail}\ s \ \} \\
& \mathit{head}\ (\mathit{stream}\ t) \prec \mathit{tail}\ (\mathit{stream}\ s) \\
= \quad & \{ \ \mathit{stream} \text{ is a representation changer: (6) and (7) } \} \\
& \mathit{root}\ t \prec \mathit{stream}\ (\mathit{chop}\ s)
\end{aligned}$$

We obtain

$$\begin{aligned}
& \mathit{stream} && :: \forall\alpha\ .\ \mathit{Tree}\ \alpha \to \mathit{Stream}\ \alpha \\
& \mathit{stream}\ t && = \mathit{root}\ t \prec \mathit{stream}\ (\mathit{chop}\ t)\ \ .
\end{aligned}$$

Conversely, for *tree* we define functions that mimic the projection functions *root*, *left* and *right*. The counterparts of *left* and *right* are $\mathit{even} \cdot \mathit{tail}$ and $\mathit{odd} \cdot \mathit{tail}$, respectively, where *even* and *odd* are given by

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

Exercise 25. Formulate laws that capture the fact that *head* is the counterpart of *root* etc. Use the laws to derive an implementation of *tree*. (The resulting equation is displayed below.) □

The isomorphism *tree* is then given by

$$\begin{aligned}
& \mathit{tree} && :: \forall\alpha\ .\ \mathit{Stream}\ \alpha \to \mathit{Tree}\ \alpha \\
& \mathit{tree}\ s && = \mathit{Node}\ (\mathit{head}\ s)\ (\mathit{tree}\ (\mathit{even}\ (\mathit{tail}\ s)))\ (\mathit{tree}\ (\mathit{odd}\ (\mathit{tail}\ s)))\ \ .
\end{aligned}$$

Exercise 26. Both *stream* and *tree* are given as unfolds or anamorphisms — they *construct* a stream from a tree and vice versa. In Haskell, inductive datatypes and coinductive types coincide [12]. For that reason, we can also define the isomorphisms as folds or catamorphisms — these variants *deconstruct* a tree to form a stream and vice versa.

$$\begin{aligned}
& \mathit{stream}'\ ({\sim}(\mathit{Node}\ a\ l\ r)) && = \mathit{node}\ a\ (\mathit{stream}'\ l)\ (\mathit{stream}'\ r) \\
& \mathit{tree}'\ ({\sim}(\mathit{Cons}\ a\ s)) && = \mathit{cons}\ a\ (\mathit{tree}'\ s)
\end{aligned}$$

(The twiddles on the left-hand side delay pattern matching for increased laziness.) Define the helper functions *node* and *cons*. □

The two functions *stream* and *tree* satisfy a variety of properties: they are mutually inverse, they are natural in the element type and, most importantly, they are idiom homomorphisms. If you have solved Exercise 26, then you know that constructing a node corresponds roughly to interleaving two streams.

$$\mathit{stream}\ (\mathit{Node}\ a\ l\ r) = a \prec \mathit{stream}\ l \curlyvee \mathit{stream}\ r \tag{8}$$

$$\mathit{tree}\ (a \prec l \curlyvee r) = \mathit{Node}\ a\ (\mathit{tree}\ l)\ (\mathit{tree}\ r) \tag{9}$$

Finally, the stream of natural numbers corresponds to the tree of natural numbers. The proof is straightforward: we show that *tree Stream.nat* satisfies the recursion equation of *Tree.nat*.

$$\begin{aligned}
& \mathit{tree}\ \mathit{nat} \\
= & \quad \{ \text{ property of } \mathit{nat} \text{ and definition of } \curlyvee \ \} \\
& \mathit{tree}\ (0 \prec 2 * \mathit{nat} + 1 \curlyvee 2 * (\mathit{nat} + 1)) \\
= & \quad \{ \text{ property of } \mathit{tree}\ (9) \ \} \\
& \mathit{Node}\ 0\ (\mathit{tree}\ (2 * \mathit{nat} + 1))\ (\mathit{tree}\ (2 * (\mathit{nat} + 1))) \\
= & \quad \{ \ \mathit{tree} \text{ is an idiom homomorphism } \} \\
& \mathit{Node}\ 0\ (2 * \mathit{tree}\ \mathit{nat} + 1)\ (2 * (\mathit{tree}\ \mathit{nat} + 1))
\end{aligned}$$

We have noted in the introduction to this section that streams relate to trees in the same way as unary numbers relate to binary numbers. A stream corresponds to a function from the natural numbers. Looking up the stream has, at best, a linear running time — if each element of the sequence is constructed in constant time. A tree corresponds to a function from the binary numbers. Looking up the tree has, at best, a logarithmic running time. Consequently, transforming a stream into a tree possibly transforms a linear into a logarithmic algorithm. In a sense, we have already seen an example along those lines: fast exponentiation. In the previous section we have derived an efficient implementation of *Tree.lookup* $((\mathit{pure}\ c)^{\mathit{Tree.nat}})$. It remains to make the transition from streams to trees explicit.

$$\begin{aligned}
& \mathit{Stream.lookup}\ ((\mathit{pure}\ c)^{\mathit{Stream.nat}}) \\
= & \quad \{ \text{ isomorphism: } \mathit{stream} \cdot \mathit{tree} = \mathit{id} \ \} \\
& \mathit{Stream.lookup}\ (\mathit{stream}\ (\mathit{tree}\ ((\mathit{pure}\ c)^{\mathit{Stream.nat}}))) \\
= & \quad \{ \ \mathit{Stream.lookup} \cdot \mathit{stream} = (\mathit{toBin} \rightarrow \mathit{id}) \cdot \mathit{Tree.lookup}\ (5) \ \} \\
& \mathit{Tree.lookup}\ (\mathit{tree}\ ((\mathit{pure}\ c)^{\mathit{Stream.nat}})) \cdot \mathit{toBin} \\
= & \quad \{ \ \mathit{tree} \text{ is an idiom homomorphism } \} \\
& \mathit{Tree.lookup}\ ((\mathit{pure}\ c)^{\mathit{tree}\ \mathit{Stream.nat}}) \cdot \mathit{toBin} \\
= & \quad \{ \ \mathit{tree}\ \mathit{Stream.nat} = \mathit{Tree.nat} \ \} \\
& \mathit{Tree.lookup}\ ((\mathit{pure}\ c)^{\mathit{Tree.nat}}) \cdot \mathit{toBin}
\end{aligned}$$

The example nicely demonstrates separation of concerns: a program is factored into a corecursive part that constructs codata and a recursive part that inspects the codata, taking care of termination.

The central step in the above derivation is the use of *tree*, which transforms a stream to a tree. Perhaps surprisingly, the opposite transformation is equally useful. If we view an infinite binary tree as a state space, then *stream* enumerates this space. The next section considers such an example.

Exercise 27. Is *chop* an idiom homomorphism? □

6.5 Application: Enumerating the Positive Rationals

This section is organised as a set of exercises around a common theme: enumerating the positive rationals. The challenge is to set things up so that every positive rational occurs *exactly once*. This side condition rules out the naïve approach, generating all possible combinations of numerators and denominators, as the resulting enumeration will contain infinitely many copies of every positive rational.

There are, in fact, several ways to enumerate the positive rationals without duplicates. Probably the oldest method was discovered in the 1850s by the German mathematician Stern and independently a few years later by the French clockmaker Brocot. It is deceptively simple: Start with the two 'boundary rationals' ${}^0/_1$ and ${}^1/_0$, which are not included in the enumeration, and then repeatedly insert the so-called *mediant* ${}^{a+b}/_{c+d}$ between two adjacent rationals ${}^a/_c$ and ${}^b/_d$.

Since the number of inserted rationals doubles with every step, the process can be pictured by an infinite binary tree, the so-called Stern-Brocot tree, see Figure 4. Its root is labelled with the first inserted mediant: ${}^{0+1}/_{1+0} = {}^1/_1$.

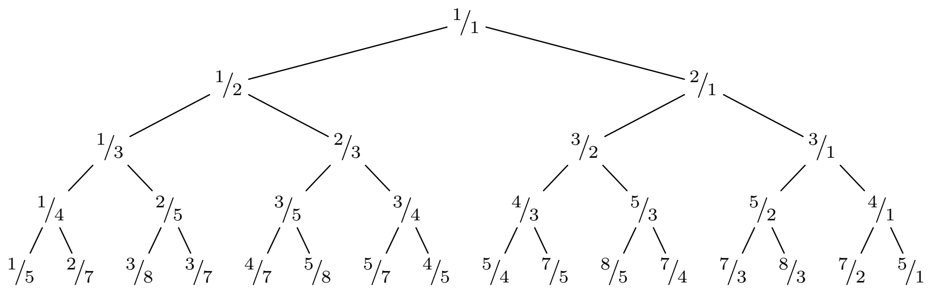


Fig. 4. Stern-Brocot tree.

Exercise 28. (*Turn the informal description into a program*) If we represent an inserted rational ${}^{a+b}/_{c+d}$ by the matrix $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$, then its left and right descendant can be determined as follows.

$$\begin{pmatrix} a & a+b \\ c & c+d \end{pmatrix} \ \hookleftarrow\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \ \mapsto\ \begin{pmatrix} a+b & b \\ c+d & d \end{pmatrix}$$

Phrase the transformations as matrix multiplications and then define the Stern-Brocot tree as an *unfold*, a *map* after an *iterate*. □

Exercise 29. (*Turn the iterative form into a recursive form*) Show that the iterative formulation is equivalent to the following recursive definition.

$$\begin{aligned} &\mathit{stern} :: \mathit{Tree\ Rational} \\ &\mathit{stern} = \mathit{Node}\ 1\ (1\ /\ (1\ /\ \mathit{stern} + 1))\ (\mathit{stern} + 1) \end{aligned}$$

The definition makes explicit that the right subtree is the 'successor' of the entire tree, see Figure 4. *Hint:* Use fusion and the recursion-iteration lemma. □

Exercise 30. (*Relate the Stern-Brocot tree to Dijkstra's fusc sequence*) In one of his EWDs [9], Dijkstra introduced the following function, also known as Stern's diatomic sequence

$$\begin{aligned} \mathcal{S}_1 &= 1 \\ \mathcal{S}_{2*n} &= \mathcal{S}_n \\ \mathcal{S}_{2*n+1} &= \mathcal{S}_n + \mathcal{S}_{n+1} \ , \end{aligned}$$

which is a strange variant of *fib*.

Tabulate the function: $\mathit{fusc} = \mathit{tabulate}\ \mathcal{S}$. *Hint:* You may find it helpful to use the function *chop* that serves as the counterpart of *tail*.

Show that $\mathit{stern} = \mathit{fusc} \div \mathit{fusc}'$, where $\div$ constructs a rational from two integers and $\mathit{fusc}' = \mathit{chop}\ \mathit{fusc}$. □

Exercise 31. (*Turn the recursive form of fusc into an iterative one*) Turn the trees

$$\begin{aligned} \mathit{num} &= \mathit{Node}\ 1\ \mathit{num}\ (\mathit{num} + \mathit{den}) \\ \mathit{den} &= \mathit{Node}\ 1\ (\mathit{num} + \mathit{den})\ \mathit{den} \end{aligned}$$

into an iterative form (*num* and *den* are more telling names for *fusc* and fusc').

There are, at least, two approaches:

- Pair *num* and *den*

  $$\mathit{num} \star \mathit{den} \ ,$$

  where $(\star) = \mathit{zip}\ (,)$ turns a pair of trees into a tree of pairs.
- Use the fact that the subtrees of *num* are linear combinations of *num* and *den*.

  $$i * \mathit{num} + j * \mathit{den}$$

  (Dijkstra [10] uses a similar approach to show that $\mathit{fusc} + \mathit{fusc}' = \mathit{brp}\ (\mathit{fusc} + \mathit{fusc}')$, where *brp* transforms a tree to its bit-reversal permutation.)

Try to relate the two approaches, see also Exercise 11. □

Exercise 32. (*Show that the rationals are in their lowest common form*) In Exercise 30 we have shown that $\mathit{stern} = \mathit{num} \div \mathit{den}$. This fact does *not*, however, imply that $\mathit{map}\ \mathit{numerator}\ \mathit{stern} = \mathit{num}$ and $\mathit{map}\ \mathit{denominator}\ \mathit{stern} = \mathit{den}$. (Why?) In order to prove the latter two equations, we have to show that the rationals $\mathit{num} \div \mathit{den}$ are in their lowest common form, that is, the greatest common divisor of *num* and *den* is 1:

$$\mathit{num} \triangledown \mathit{den} = 1 \ ,$$

where $\triangledown$ denotes the greatest common divisor lifted to trees. □

Exercise 33. (*Show that the Stern-Brocot tree contains every rational at most once*) Again, there are, at least, two approaches. One can show that *stern* is a search-tree using the following fact about mediants: if ${}^{a}/_{c} \leqslant {}^{b}/_{d}$, then

$${}^{a}/_{c} \leqslant {}^{a+b}/_{c+d} \leqslant {}^{b}/_{d} \ .$$

Alternatively, one can show that *lookup stern* is injective by demonstrating that it has a left-inverse (g is the left-inverse of f iff $g \cdot f = id$). Rational numbers are in a one-to-one correspondence to bit strings. The following instrumented version of the greatest common divisor

```
$a \blacktriangledown b = \mathbf{case}\ compare\ a\ b\ \mathbf{of}$
            $LT \rightarrow 0 : (a \blacktriangledown (b - a))$
            $EQ \rightarrow [\,]$
            $GT \rightarrow 1 : ((a - b) \blacktriangledown b)$ ,
```

maps two positive numbers to a bit string. We claim that this defines the required left-inverse. Establish the result by showing

$$num \blacktriangledown den = tabulate\ id \ .$$

Why is this sufficient? □

Exercise 34. (*Show that the Stern-Brocot tree contains every rational at least once*) Show that *lookup stern* is surjective by demonstrating that it has a right-inverse (g is the right-inverse of f iff $f \cdot g = id$). □

Exercise 35. (*Linearise the Stern-Brocot tree*) Turn *stream stern* into an iterative form. In other words, enumerate the rationals!

1. As a first step, linearise *den*. You have to express *chop den* in terms of *den* and possibly *num*. To this end show that $chop\ den = num + den - 2 * x$ where x is the unique solution of $x = Node\ 0\ num\ x$.
2. Show that the unique solution of $x = Node\ 0\ num\ x$ equals $num\ \mathbf{mod}\ den$.
3. Using the results of the two previous items, linearise *num* and *den*, defining $snum = stream\ num$ and $sden = stream\ den$.
4. Turn $snum \star sden$ into an iterative form.
5. *Polishing up:* Use the formula

   $$1\,/\,(\lfloor n \div d \rfloor + 1 - \{n \div d\}) = d \div (n + d - 2 * (n\ \mathbf{mod}\ d))$$

   to turn the result of the previous item into the following amazingly short program for enumerating the rationals.

   ```
   $rationals = iterate\ next\ 1$
     $\mathbf{where}\ next\ r = 1\,/\,(\lfloor r \rfloor + 1 - \{r\})$
   ```

   Here, $\lfloor r \rfloor$ denotes the integral part of r and $\{r\}$ its fractional part, such that $r = \lfloor r \rfloor + \{r\}$. □

6.6 Summary and Related Work

The type of infinite binary trees is another example of a coinductive datatype. Like streams, infinite trees form an idiom. Trees can be defined using recursion equations; admissible equations have unique solutions. Unlike streams, recursive and iterative constructions do not coincide: one tree is the bit-reversal permutation tree of the other. A tree tabulates a function from the binary numbers. Tabulation and look-up are idiom isomorphisms between the environment idiom $Bin \rightarrow$ and $Tree$.

The section is loosely based on the paper "The Bird tree" [19], which introduces an alternative scheme for enumerating the positive rationals. It also develops an almost loopless algorithm for enumerating the elements of the infinite tree $recurse\ (a\circ)\ (b\circ)\ \epsilon$, where a and b are elements of some given group.

7 Tabulation

We have repeatedly stressed the fact that a stream can be seen as a tabulation of a function from the unary numbers and that a tree tabulates a function from the binary numbers.

$$Nat \rightarrow \gamma \cong Stream\ \gamma$$
$$Bin \rightarrow \gamma \cong Tree\ \gamma$$

In this section we look at this relationship from a more principled perspective and show, among other things, that the two isomorphisms are based on the laws of exponentials.

As a warm-up exercise, consider tabulating a function from a non-recursive datatype. Probably every textbook on computer architecture includes truth tables for the logical connectives.

$$(\wedge) :: (Bool, Bool) \rightarrow Bool$$

False	*False*
False	*True*

A function from a pair of Booleans can be represented by a two-by-two table. Expressed in terms of type constructors we have

$$(Bool, Bool) \rightarrow Bool \cong ((Bool, Bool), (Bool, Bool)) \ .$$

The relationship becomes more perspicuous, if we use mathematical notation for the types: $(Bool, Bool)$ corresponds to $(1+1) \times (1+1)$ where 1 is a one-element type, $+$ is disjoint union and $\times$ denotes the cartesian product — called $()$, *Either* and $(,)$ in Haskell. Rephrasing the above isomorphism in terms of the 'arithmetic types' we obtain

$$(1+1) \times (1+1) \rightarrow Bool \cong (Bool \times Bool) \times (Bool \times Bool) \ . \tag{10}$$

If we furthermore write the function space $K \rightarrow V$ as an exponential V^K — the type K is mnemonic for *key type* and V for *value type* — we realise that tabulation rests on the well-known *laws of exponentials*.

$$X^0 \cong 1 \qquad X^1 \cong X \qquad X^{A+B} \cong X^A \times X^B \qquad X^{A\times B} \cong (X^B)^A$$

A straightforward application of these laws proves the correspondence above, namely that we can tabulate a function from a pair of Booleans using a two-by-two table.

$$\begin{array}{ll} & \mathit{Bool}^{(1+1)\times(1+1)} \\ = & \{\ X^{A\times B} \cong (X^B)^A\ \} \\ & (\mathit{Bool}^{1+1})^{1+1} \\ = & \{\ X^{A+B} \cong X^A \times X^B\ \} \\ & (\mathit{Bool}^1 \times \mathit{Bool}^1)^1 \times (\mathit{Bool}^1 \times \mathit{Bool}^1)^1 \\ = & \{\ X^1 \cong X\ \} \\ & (\mathit{Bool} \times \mathit{Bool}) \times (\mathit{Bool} \times \mathit{Bool}) \end{array}$$

The derivation holds for every return type, so Equation (10) can, in fact, be generalised to an isomorphism between two type constructors

$$\Lambda\ V\ .\ (1+1)\times(1+1) \to V \cong \Lambda\ V\ .\ (V \times V) \times (V \times V)\ ,$$

or equivalently, in a 'point-free style',

$$(1+1)\times(1+1) \to\ \cong (\mathit{Id} \mathbin{\dot{\times}} \mathit{Id}) \mathbin{\dot{\times}} (\mathit{Id} \mathbin{\dot{\times}} \mathit{Id})\ .$$

This is an isomorphism between two type constructors of kind $\star \to \star$. On the left-hand side, the two-argument type constructor '$\to$' is written without its second argument, so $\mathit{Bool} \times \mathit{Bool} \to$ has kind $\star \to \star$. On the right-hand side, we use the identity type constructor of kind $\star \to \star$ and the lifted product, which sends two type constructors of kind $\star \to \star$ to another type constructor of this kind. Using the types introduced in Section 2, the laws of exponentials can be rephrased as follows:

$$\begin{array}{llll} 0 \to \gamma & \cong 1 & \qquad 0 \to & \cong \mathit{Const}\ 1 \\ 1 \to \gamma & \cong \gamma & \qquad 1 \to & \cong \mathit{Id} \\ (\alpha+\beta) \to \gamma & \cong (\alpha \to \gamma) \times (\beta \to \gamma) & \qquad (\alpha+\beta) \to & \cong (\alpha \to) \mathbin{\dot{\times}} (\beta \to) \\ (\alpha\times\beta) \to \gamma & \cong \alpha \to (\beta \to \gamma) & \qquad (\alpha\times\beta) \to & \cong (\alpha \to) \cdot (\beta \to)\ . \end{array}$$

The constructors on the right-hand side are container types. To represent a function from the empty type, we use an empty container; to represent a function from the one-element type, we use a one-element container; to represent a function from a disjoint union, we use a pair of containers; and finally, to represent a function from a pair, we use nested containers. The last law captures *currying*: a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument. The law underlies, for instance, representations of two-dimensional arrays as arrays of arrays in the programming languages C or Java.

As an intermediate summary, tabulation is defined by induction on the structure of the key type; the construction is, however, parametric in the return type. Looking back at Section 2, we notice that all the container types involved have the structure of an idiom. Moreover, tabulation preserves the idiomatic structure of the environment idiom: one can show that datatype-generic versions of *tabulate* and *lookup* are idiom isomorphisms between the environment idiom and memo-tables. The proof is beyond the scope of these lecture notes.

Turning to recursive datatypes, we note that a function from a recursive type is tabulated by a recursive container type. Actually, we can be more precise than that: a function from an *inductive* type is tabulated by a *coinductive* container type. And indeed, both *Nat* and *Bin* are inductive types and both *Stream* and *Tree* are coinductive types. (In Haskell, inductive and coinductive types coincide, but it is useful to maintain the distinction.) Writing $\mu\ \alpha\ .\ \tau$ for an inductive type and $\nu\ \alpha\ .\ \tau$ for a coinductive type, the isomorphisms for streams and infinite trees can be written

$$(\mu\ \alpha\ .\ 1+\alpha) \rightarrow \quad \cong \nu\ \beta\ .\ \mathit{Id} \mathbin{\dot\times} \beta$$
$$(\mu\ \alpha\ .\ 1+\alpha+\alpha) \rightarrow \cong \nu\ \beta\ .\ \mathit{Id} \mathbin{\dot\times} \beta \mathbin{\dot\times} \beta\ .$$

The notation nicely makes the structure of the key and the corresponding container type explicit. In terms of constructors and destructors: 0 and +1 correspond to *head* and *tail*; *Nil*, *One* and *Two* correspond to *root*, *left* and *right*.

Table 4 extends the correspondence between key and container types to parametric types and types with embedded recursive types. To reduce clutter, we abbreviate $(\alpha_1, \ldots, \alpha_n)$ by $\boldsymbol{\alpha}$. It is understood that for each definition of K in the left column, T is defined by the corresponding entry in the right column.

Table 4. Tabulation: types of keys $K(\boldsymbol{\alpha})$ and tables $T(\boldsymbol{\beta})$.

$K(\boldsymbol{\alpha}) = \alpha_i$	$T(\boldsymbol{\beta}) = \beta_i$
$K(\boldsymbol{\alpha}) = 0$	$T(\boldsymbol{\beta}) = \mathit{Const}\ 1$
$K(\boldsymbol{\alpha}) = 1$	$T(\boldsymbol{\beta}) = \mathit{Id}$
$K(\boldsymbol{\alpha}) = K_1(\boldsymbol{\alpha}) + K_2(\boldsymbol{\alpha})$	$T(\boldsymbol{\beta}) = T_1(\boldsymbol{\beta}) \mathbin{\dot\times} T_2(\boldsymbol{\beta})$
$K(\boldsymbol{\alpha}) = K_1(\boldsymbol{\alpha}) \times K_2(\boldsymbol{\alpha})$	$T(\boldsymbol{\beta}) = T_1(\boldsymbol{\beta}) \cdot T_2(\boldsymbol{\beta})$
$K(\boldsymbol{\alpha}) = \mu\ \alpha\ .\ K_1(\boldsymbol{\alpha}, \alpha)$	$T(\boldsymbol{\beta}) = \nu\ \beta\ .\ T_1(\boldsymbol{\beta}, \beta)$

Without proof we state the following

Theorem 1 (Tabulation). *Let $K(\boldsymbol{\alpha})$ and $T(\boldsymbol{\beta})$ be defined as in Table 4. Then*

$$K(\tau_1, \ldots, \tau_n) \rightarrow \cong T(\tau_1 \rightarrow, \ldots, \tau_n \rightarrow)\ .$$

for all types $\tau_1,\ \ldots,\ \tau_n$. □

Note that the type $T(\boldsymbol{\beta})$ of memo-tables contains only products, no sums, hence the terms table and tabulation. All the examples of tabulation we have seen before are instances of this scheme. For variety, let us discuss two further examples.

We have primarily considered functions from the natural numbers, what about the integers? Well, if integers are represented by

data *Int* = *Neg Nat* | *Zero* | *Pos Nat* , then

$$\textbf{data}\ \mathit{Tape}\ \alpha = \mathit{Window}\ \{\mathit{neg} :: \mathit{Stream}\ \alpha, \mathit{zero} :: \alpha, \mathit{pos} :: \mathit{Stream}\ \alpha\}\ .$$

is a suitable container type. A container of type *Tape* α can be seen as a tape that extends infinitely to the left and infinitely to the right, with the *zero* component marking the current position. Phrased in terms of the arithmetic type constructors, the two types are related by

$$\mathit{Nat} + 1 + \mathit{Nat} \to \cong \mathit{Stream}\ \dot{\times}\ \mathit{Id}\ \dot{\times}\ \mathit{Stream}\ .$$

If the key type involves products, then the container type is nested accordingly. For instance to represent a function from a pair of natural numbers, we use a stream of streams.

$$\mathit{Nat} \times \mathit{Nat} \to \cong \mathit{Stream} \cdot \mathit{Stream}$$

As we have noted before, the isomorphism above also underlies the usual encoding of two-dimensional arrays in C or Java — an array is a finite map of type $\{0, \ldots, n-1\} \to$ where n is the size.

Let us conclude the section with a brief discussion of proof techniques. If we want to establish a property of a function from the naturals, we have, at least, two choices. The standard approach is to use induction and case analysis, see Figure 1. A less conventional approach, favoured in these lecture notes, is to rephrase the function and the property in terms of streams and to use coinduction or, preferably, the unique fixed-point principle. Theorem 1 explains why the eschewed case-analysis disappears, it is replaced by a proof about pairs.

case analysis	$K_1(\boldsymbol{\alpha}) + K_2(\boldsymbol{\alpha})$	$T_1(\boldsymbol{\beta})\ \dot{\times}\ T_2(\boldsymbol{\beta})$	pairs
pairs	$K_1(\boldsymbol{\alpha}) \times K_2(\boldsymbol{\alpha})$	$T_1(\boldsymbol{\beta}) \cdot T_2(\boldsymbol{\beta})$	nested proofs
induction	$\mu\ \alpha\ .\ K_1(\boldsymbol{\alpha}, \alpha)$	$\nu\ \beta\ .\ T_1(\boldsymbol{\beta}, \beta)$	coinduction

The laws of exponentials eliminate sums and consequently proofs by case analysis. This is why the unique fixed-point proof in Section 1 is so much more attractive than the inductive proof. To establish the equality of two pairs, we simply have to show that the corresponding elements are equal.

However, all that glitters is not gold. If the key type involves products, then we have to deal with a nested container type, which is often less manageable. Of course, tabulation establishes an isomorphism, which also allows us to transfer proofs from one setting to the other. So in principle, we can port an inductive proof to the coinductive setting. Conversely, even if we do not use streams or other coinductive types directly, we may profit from the widened perspective.

Overall, tabulation is a very valuable tool in the arsenal of techniques for program derivation and verification and it certainly deserves to be better known.

7.1 Summary and Related Work

Tabulation is based on the laws of exponentials. A function from an inductive type is tabulated by a coinductive type. Memo-tables are basically products, hence the name. In particular, they do not contain sums, which explains why the proofs in these lecture notes do without case analysis.

Finite versions of memo-tables are known as *tries* or *digital search trees*. Knuth [25] attributes the idea of a trie to Thue [36]. Connelly and Morris [5] formalised the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation. The author gave a datatype-generic or polytypic definition of tries and memo-tables using type-indexed datatypes [16, 17]. The insight that a function from an inductive type is tabulated by a coinductive type is due to Altenkirch [2]. If the trie structures are deforested, we obtain linear algorithms for sorting and grouping [15]. Like tries, these algorithms do not depend on an ordering relation, but use the structure of the elements to organise the working.

8 Conclusion

I hope you have enjoyed the journey. By and large, coinductive datatypes and corecursive programs are under-appreciated. We have demonstrated that they nicely support a holistic or wholemeal approach to programming and proving. A stream enables us to treat an infinite sequence of elements as a single entity. Likewise, a tree captures an infinite binary process.

Streams and trees can be conveniently defined using recursion equations. Admissible equations have unique solutions, which is the basis of the unique fixed-point principle. Both coinductive types have additional structure that can be put to good use. The idiomatic structure allows us to lift operations, which is a notational convenience not to be underestimated. Definitions and calculations benefit from the fact that streams and trees are memo-tables and that look-up and tabulation are idiom homomorphisms.

References

1. Aczel, P., Mendler, N.: A final coalgebra theorem. In: Pitt, D., Rydeheard, D., Dybjer, P., Poigné, A. (eds.) Category Theory and Computer Science (Manchester). Lecture Notes in Computer Science, vol. 389, pp. 357–365. Springer-Verlag, Berlin (1989)
2. Altenkirch, T.: Representations of first order function types as terminal coalgebras. In: Typed Lambda Calculi and Applications, TLCA 2001. Lecture Notes in Computer Science, vol. 2044, pp. 62–78. Springer-Verlag (2001)
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
4. Bird, R.: An introduction to the theory of lists. In: Broy, M. (ed.) Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design, Marktoberdorf, Germany. pp. 5–42. Springer-Verlag (1987)
5. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science 5(3), 381–418 (September 1995)
6. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen,

The Netherlands, May 24-–28, 1993, Selected Papers, Lecture Notes in Computer Science, vol. 806, pp. 62–78. Springer-Verlag (1994)

7. Curry, H., Feys, R.: Combinatory Logic, Volume 1. North-Holland, Amsterdam New York Oxford (1958)
8. Danvy, O.: An extensional characterization of lambda-lifting and lambda-dropping. In: Middeldorp, A., Sato, T. (eds.) 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan. Lecture Notes in Computer Science, vol. 1722, pp. 241–250. Springer-Verlag (November 1999)
9. Dijkstra, E.W.: EWD570: An exercise for Dr.R.M.Burstall (May 1976), the manuscript was published as pages 215–216 of Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982. ISBN 0–387–90652–5.
10. Dijkstra, E.W.: EWD578: More about the function "fusc" (a sequel to EWD570) (May 1976), the manuscript was published as pages 230–232 of Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982. ISBN 0–387–90652–5.
11. Fokkinga, M.M.: Law and Order in Algorithmics. Ph.D. thesis, University of Twente (February 1992)
12. Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Tech. Rep. CS-R9104, Centre of Mathematics and Computer Science, CWI, Amsterdam (January 1991)
13. Gibbons, J., Hutton, G.: Proof methods for corecursive programs. Fundamenta Informaticae (XX), 1–14 (2005)
14. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete mathematics. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edn. (1994)
15. Henglein, F.: Generic discrimination: Sorting and partitioning unshared data in linear time. In: Thiemann, P. (ed.) Proceedings of the 13th ACM Sigplan International Conference on Functional Programming (ICFP'08), September 22–24, 2008, Victoria, BC, Canada. pp. 91–102. ACM, New York, NY, USA (September 2008)
16. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming 10(4), 327–351 (July 2000)
17. Hinze, R.: Memo functions, polytypically! In: Jeuring, J. (ed.) Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal. pp. 17–32 (July 2000), the proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19
18. Hinze, R.: Functional Pearl: Streams and unique fixed points. In: Thiemann, P. (ed.) Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08). pp. 189–200. ACM Press (September 2008)
19. Hinze, R.: Functional Pearl: The Bird tree. J. Functional Programming 19(5), 491–508 (September 2009)
20. Hinze, R.: Scans and convolutions—a calculational proof of Moessner's theorem. In: Scholz, S.B. (ed.) 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. Lecture Notes in Computer Science, vol. 5836. Springer-Verlag (2009)
21. Hinze, R.: Lifting operators and laws (2010), available from `http://www.comlab.ox.ac.uk/ralf.hinze/Lifting.pdf`
22. Hinze, R., Löh, A.: Guide2lhs2tex (for version 1.13) (February 2008), `http://people.cs.uu.nl/andres/lhs2tex/`
23. Hutton, G., Meijer, E.: Functional Pearl: Back to basics: Deriving representation changers functionally. J. Functional Programming 6(1), 181–188 (January 1996)

24. Karczmarczuk, J.: Generating power of lazy semantics. Theoretical Computer Science (187), 203–219 (1997)
25. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing Company, 2nd edn. (1998)
26. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, Springer-Verlag, Berlin, 2nd edn. (1998)
27. McBride, C., Paterson, R.: Functional Pearl: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)
28. McIlroy, M.D.: Power series, power serious. J. Functional Programming 3(9), 325–337 (May 1999)
29. McIlroy, M.D.: The music of streams. Information Processing Letters (77), 189–195 (2001)
30. Milner, R.: Communication and Concurrency. International Series in Computer Science, Prentice Hall International (1989)
31. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
32. Rutten, J.: Fundamental study: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. Theoretical Computer Science 308, 1–53 (2003)
33. Rutten, J.: A coinductive calculus of streams. Math. Struct. in Comp. Science 15, 93–147 (2005)
34. Röjemo, N.: Garbage collection, and memory efficiency, in lazy functional languages. Ph.D. thesis, Chalmers University of Technology (1995)
35. Sloane, N.J.A.: The on-line encyclopedia of integer sequences [online] (2009), available at: `http://www.research.att.com/~njas/sequences/`
36. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse 1, 1–67 (1912), reprinted in Thue's "Selected Mathematical Papers" (Oslo: Universitetsforlaget, 1977), 413–477
37. Turner, D.: A new implementation technique for applicative languages. Software - Practice and Experience 9, 31–49 (1979)