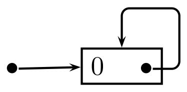
Chapter 8

Streams

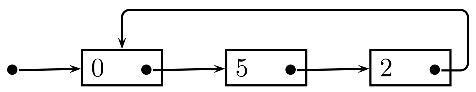
Computers and people have finite amounts of memory. They can't store infinite information. Therefore, it is impossible for us to represent an object with infinite parts in our mind or in digital storage. However, we can represent such object by a finite amount of information.

Let's start by considering *streams*, that is, infinite sequences of values. While elements of a finite list can be completely stored in memory, obviously a stream cannot.

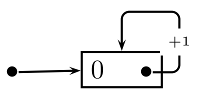
Sometimes we can represent a stream by a circular structure. For example the stream that repeat the zero element, $0 \lhd 0 \lhd 0 \lhd \cdots$ can be represented by a circular linked list:



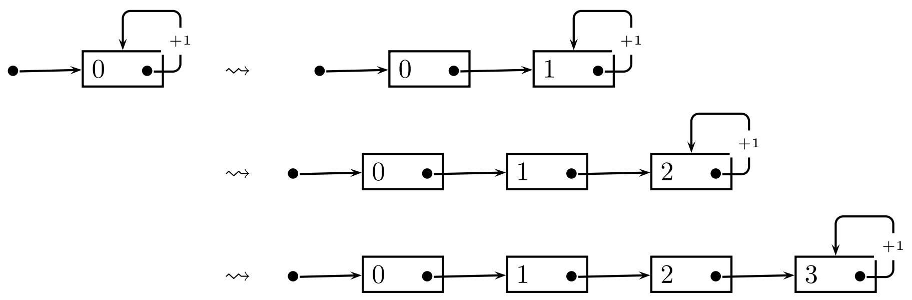
The periodic stream $0 \lhd 5 \lhd 2 \lhd 0 \lhd 5 \lhd 2 \lhd 0 \lhd 5 \lhd 2 \lhd \cdots$ is represented by



But how do we represent non-cyclic streams, for example the stream of all natural numbers, $0 \lhd 1 \lhd 2 \lhd 3 \lhd \cdots$? We can specify an operation that will be performed at each unfolding to change the state:



Unfolding this circular list will progressively increase the element value:



TO DO [Further example: Fibonacci stream, done with two pointers. Expand the previous explanations, saying explicitely how the linked lists work]

In general, we can generate streams by giving computational processes that produce one elemente at a time, while changing their state at every step. Such processes are called *colagebras*

TO DO [(Handwritten notes INF1/2) Somewhere say that we work with Haskell and Coq (and Agda?) and explain the theoretical basis for these languages/systems.]

Streams are an example of a *coinductive type*. Coinductive types are sets of objects with a potentially infinite structure. They are specified by a list of *constructors* and their types, with the understanding that we can apply them an infinite number of times to build an object.

Specifically, the type of streams (infinite sequences) over a given type A is defined by:

$$\begin{array}{l}\mathsf{codata}\ \mathsf{Stream}_A : \mathsf{Set}\\ \quad (\triangleleft) : A \to \mathsf{Stream}_A \to \mathsf{Stream}_A.\end{array}$$

This says that Stream_A is a set of elements, each one $\alpha : \mathsf{Stream}_A$ must have the form $\alpha = a_0 \triangleleft \alpha_1$, with a head $a_0 : A$ and a tail $\alpha_1 : \mathsf{Stream}_A$. In turn, also α_1 must have the same form, $\alpha_1 = a_1 \triangleleft \alpha_2$, and again for α_2 and so on. This process never ends, so an element has an infinitely unfolding structure. We have an infinite regression. So how can we ever define some element of Stream_A in practice? We will develop some formalisms to specify such infinite objects by finite definitions.

The dual notion is that of *inductive type*, whose elements have a well-founded structure, meaning that only a finite consecutive sequence of constructors can occur. For example, the type of lists over A is defined by:

$$\begin{array}{l}\mathsf{data}\ \mathsf{List}_A : \mathsf{Set}\\ \quad \mathsf{nil} : \mathsf{List}_A\\ \quad (::) : A \to \mathsf{List}_A \to \mathsf{List}_A.\end{array}$$

The difference here is that the recursive constructor $(::)$ can be applied only a finite number of times before we use the base constructor nil. A list is explicitely defined by giving all its structure, for example $7 :: 9 :: 0 :: \mathsf{nil}$.

This is obviously not possible for streams. We can only build a stream by a recursive definition. For example, the stream consisting in an infinite sequence of copies of the number 3, that is $3 \triangleleft 3 \triangleleft 3 \triangleleft \cdots$, can be given by

$$\begin{aligned}&\mathsf{inf}_3 : \mathsf{Stream}_{\mathbb{N}}\\ &\mathsf{inf}_3 = 3 \triangleleft \mathsf{inf}_3.\end{aligned}$$

In general we can specify a constant stream, a stream that repeats the same given element:

$$\begin{aligned}&\mathsf{inf} : A \to \mathsf{Stream}_A\\ &\mathsf{inf}_a = a \triangleleft (\mathsf{inf}_a).\end{aligned}$$

(Later we will simply denote the infinite stream inf_a by $\bar{a}$.)

We must impose restrictions on which recursive definitions are allowed. In the definition above, we use inf_3 in its own definition, that is, we assume that it already exists to define it. This is justified because we made sure that the unfolding of the definition generates at least one constructor $(3 \triangleleft\ )$, so we can continue to unfold it to generate the whole sequence:

$$\begin{aligned}\mathsf{inf}_3 &= 3 \triangleleft \mathsf{inf}_3 && \text{by definition/unfolding}\\ &= 3 \triangleleft (3 \triangleleft \mathsf{inf}_3) && \text{unfolding of } \mathsf{inf}_3\\ &= 3 \triangleleft 3 \triangleleft (3 \triangleleft \mathsf{inf}_3) && \text{unfolding of } \mathsf{inf}_3\\ &= 3 \triangleleft 3 \triangleleft 3 \triangleleft \cdots && \text{and so on.}\end{aligned}$$

Similarly, the stream of all natural numbers can be defined like this:

$$\begin{aligned}&\mathsf{nats} : \mathsf{Stream}_{\mathbb{N}}\\ &\mathsf{nats} = \mathsf{natsFrom}\, 0\\[1em] &\mathsf{natsFrom} : \mathbb{N} \to \mathsf{Stream}_{\mathbb{N}}\\ &\mathsf{natsFrom}\, n = n \triangleleft \mathsf{natsFrom}\, (n+1).\end{aligned}$$

Pattern Matching Since we know that every element must have a constructor form, we can define functions on the type of streams by pattern matching, that is, by specifying how they work on elements in constructor form. For example, here are the definitions of the head and tail functions:

$$\begin{aligned}&\mathsf{head} : \mathsf{Stream}_A \to A && \mathsf{tail} : \mathsf{Stream}_A \to \mathsf{Stream}_A\\ &\mathsf{head}\, (a \triangleleft \alpha) = a && \mathsf{tail}\, (a \triangleleft \alpha) = \alpha\end{aligned}$$

We'll use the notation ${}^{\mathsf{h}}\alpha$ for $(\mathsf{head}\, \alpha)$ and ${}^{\mathsf{t}}\alpha$ for $(\mathsf{tail}\, \alpha)$.

Not all recursive definitions are sound, even if we make sure to produce a head element at the first unfolding. Thake for example

$$\begin{aligned}&\mathsf{wrongStr} : \mathsf{Stream}_{\mathbb{N}}\\ &\mathsf{wrongStr} = 3 \triangleleft {}^{\mathsf{t}}\mathsf{wrongStr}.\end{aligned}$$

This definition produces a first element 3, but then get stuck, because its tail is defined to be equal to its tail, without any additional information that would help to generate it. If we try to unfold the definition, we get into a non-productive infinite loop:

$$\begin{array}{rll}
\mathsf{wrongStr} & = 3 \lhd {}^{\mathsf{t}}\mathsf{wrongStr} & \text{by definition/unfolding} \\
& = 3 \lhd {}^{\mathsf{t}}(3 \lhd {}^{\mathsf{t}}\mathsf{wrongStr}) & \text{unfolding of } \mathsf{wrongStr} \\
& = 3 \lhd {}^{\mathsf{t}}\mathsf{wrongStr} & \text{definition of } \mathsf{tail} \\
& = \cdots & \text{repeat the previous two steps.}
\end{array}$$

The following is a simultaneous definition of two functions evens and odds that extract the elements of a stream in positions with even and odd indices, respectively.

$$\begin{array}{l}
\mathsf{evens} : \mathsf{Stream}_A \to \mathsf{Stream}_A \\
\mathsf{evens}\,(a_0 \lhd \alpha_1) = a_0 \lhd (\mathsf{odds}\,\alpha_1) \\
\\
\mathsf{odds} : \mathsf{Stream}_A \to \mathsf{Stream}_A \\
\mathsf{odds}\,(a_0 \lhd \alpha_1) = \mathsf{evens}\,\alpha_1
\end{array}$$

They are mutually recursive; unfolding them generates a result sequence that produces one element for every two elements of the input:

$$\begin{array}{rll}
\mathsf{evens}\,\mathsf{nats} & = \mathsf{evens}\,(\mathsf{natsFrom}\,0) & \text{definiton of } \mathsf{nats} \\
& = \mathsf{evens}\,(0 \lhd (\mathsf{natsFrom}\,1)) & \text{definiton/unfolding of } \mathsf{natsFrom} \\
& = 0 \lhd (\mathsf{odds}(\mathsf{natsFrom}\,1)) & \text{definition of } \mathsf{evens} \\
& = 0 \lhd (\mathsf{odds}(1 \lhd\ (\mathsf{natsFrom}\,2))) & \text{unfolding of } \mathsf{natsFrom} \\
& = 0 \lhd (\mathsf{evens}\,(\mathsf{natsFrom}\,2)) & \text{definition of } \mathsf{odds} \\
& = 0 \lhd (\mathsf{evens}\,(2 \lhd (\mathsf{natsFrom}\,3))) & \text{unfolding of } \mathsf{natsFrom} \\
& = 0 \lhd 2 \lhd (\mathsf{odds}\,(\mathsf{natsFrom}\,3)) & \text{definition of } \mathsf{evens} \\
& = \cdots & \text{and so on.}
\end{array}$$

The inverse operation consists in zipping two streams together, interleaving their elements.

$$\begin{array}{l}
(\ltimes) : \mathsf{Stream}_A \to \mathsf{Stream}_A \to \mathsf{Stream}_A \\
(a \lhd \alpha) \ltimes \beta = a \lhd (\beta \ltimes \alpha).
\end{array}$$

Proposition 16 *The interleave function* $(\ltimes)$ *is the inverse of the pair* $(\mathsf{evens}, \mathsf{odds})$*, that is, for all streams* $\alpha, \alpha_1, \alpha_2 : \mathsf{Stream}_A$*,*

$$\begin{array}{l}
(\mathsf{evens}\,\alpha) \ltimes (\mathsf{odds}\,\alpha) = \alpha \\
\mathsf{evens}\,(\alpha_1 \ltimes \alpha_2) = \alpha_1 \\
\mathsf{odds}\,(\alpha_1 \ltimes \alpha_2) = \alpha_2.
\end{array}$$

Proof. **TO DO** [Just an informal argument here, with a remark that after proving the first steps we recurse on the proof itself. Promise to make this proof method formally precise when we introduce bisimulation.] □

In the cases that we looked at, it is quite easy to see when the definition is sound and produces a whole infinite sequence by unfolding, and when it is not sound, getting stuck at some point. But it is not always obvious.

Let's look at two slightly more compolicated function, that look very similar but have very different unfolding behaviour.

$$
\begin{aligned}
&\psi : \mathsf{Stream}_A \to \mathsf{Stream}_A\\
&\psi\,\alpha = {}^{\mathsf{h}}\alpha \lhd \mathsf{evens}\,(\psi\,(\mathsf{odds}\,{}^{\mathsf{t}}\alpha)) \ltimes \mathsf{odds}\,(\psi\,(\mathsf{evens}\,{}^{\mathsf{t}}\alpha))
\end{aligned}
$$

Let's try it on the stream of natural numbers: **TO DO** [all wrong, to be redone]

$$
\begin{aligned}
\psi\,\mathsf{nats} &= {}^{\mathsf{h}}\mathsf{nats} \lhd \;\; \mathsf{evens}\,(\psi\,(\mathsf{odds}\,{}^{\mathsf{t}}\mathsf{nats})) && \text{definition/unfolding of } \psi\\
&\qquad\qquad \ltimes \mathsf{odds}\,(\psi\,(\mathsf{evens}\,{}^{\mathsf{t}}\mathsf{nats}))\\
&= 0 \lhd \;\; \mathsf{evens}\,(\psi\,(2 \lhd 4 \lhd 6 \lhd 8 \lhd 10 \lhd \cdots)) && \text{definition of } \mathsf{nats}, \mathsf{odds}, \mathsf{evens}\\
&\qquad \ltimes \mathsf{odds}\,(\psi\,(1 \lhd 3 \lhd 5 \lhd 7 \lhd 9 \lhd \cdots))\\
&= 0 \lhd \;\; \mathsf{evens}\,(2 \lhd \;\; \mathsf{evens}\,(\psi\,(6 \lhd 10 \lhd \cdots)) && \text{definition of } \psi\\
&\qquad\qquad\qquad \ltimes \mathsf{odds}\,(\psi\,(4 \lhd 8 \lhd \cdots)))\\
&\qquad \ltimes \mathsf{odds}\,(1 \lhd \;\; \mathsf{evens}\,(\psi\,(5 \lhd 9 \lhd \cdots))\\
&\qquad\qquad\qquad \ltimes \mathsf{odds}\,(\psi\,(3 \lhd 7 \lhd \cdots)))\\
&= 0 \lhd 2 \lhd \;\; \mathsf{odds}\,(1 \lhd \;\; \mathsf{evens}\,(\psi\,(5 \lhd 9 \lhd \cdots)) && \text{definition of } \mathsf{evens}, \ltimes\\
&\qquad\qquad\qquad \ltimes \mathsf{odds}\,(\psi\,(3 \lhd 7 \lhd \cdots)))\\
&\qquad\quad \ltimes \mathsf{odds}\,(\;\; \mathsf{evens}\,(\psi\,(6 \lhd 10 \lhd \cdots))\\
&\qquad\qquad\qquad \ltimes \mathsf{odds}\,(\psi\,(4 \lhd 8 \lhd \cdots)))\\
&= \;\; 0 \lhd 2 \lhd 5 \lhd 12 \lhd 25 \lhd 52 \lhd 105 \lhd 212 \lhd 425 \lhd 852 \lhd \cdots
\end{aligned}
$$

The unfolding of $(\psi\,\mathsf{nats})$ becomes very complex quickly and its size grows exponentially. It is not easy to compute by hand even the first few elements of the result. This is a good moment for you to try to implement some stream functions in your favorite programming language and verify the calculation above. I have done it using Haskell: it is a lazy functional language, which makes computations with streams and other infinite structures easy.

In any case, experimentation seems to show that the function ψ behaves productively: it correctly generates an output stream for every input.

Now look at the following function definition $\rotatebox{180}{$\psi$}$ (pronounced *"isp"*), which is very similar to ψ, we only swapped the roles of evens and odds.

$$
\begin{aligned}
&\rotatebox{180}{ψ} : \mathsf{Stream}_A \to \mathsf{Stream}_A\\
&\rotatebox{180}{ψ}\,\alpha = {}^{\mathsf{h}}\alpha \lhd \mathsf{odds}\,(\rotatebox{180}{ψ}\,(\mathsf{evens}\,{}^{\mathsf{t}}\alpha)) \ltimes \mathsf{evens}\,(\rotatebox{180}{ψ}\,(\mathsf{odds}\,{}^{\mathsf{t}}\alpha))
\end{aligned}
$$

When we try to compute it, we get a problem:

$$\begin{array}{lll}
\phi\,\mathsf{nats} & = {}^{\mathsf{h}}\mathsf{nats} \triangleleft \ \ \mathsf{odds}\,(\phi\,(\mathsf{evens}\,{}^{\mathsf{t}}\mathsf{nats})) & \text{definition/unfolding of } \phi \\
& \qquad\qquad \ltimes\ \mathsf{evens}\,(\phi\,(\mathsf{odds}\,{}^{\mathsf{t}}\mathsf{nats})) & \\
& = 0 \triangleleft \ \ \mathsf{odds}\,(\phi\,(1 \triangleleft 3 \triangleleft 5 \triangleleft 7 \triangleleft 9 \triangleleft \cdots)) & \text{definition of } \mathsf{nats}, \mathsf{evens}, \mathsf{odds} \\
& \qquad \ltimes\ \mathsf{evens}\,(\phi\,(2 \triangleleft 4 \triangleleft 6 \triangleleft 8 \triangleleft 10 \triangleleft \cdots)) & \\
& = 0 \triangleleft \ \ \mathsf{odds}\,(1 \triangleleft \ \ \mathsf{odds}\,(\phi\,(3 \triangleleft 7 \triangleleft \cdots)) & \text{definition of } \phi \\
& \qquad\qquad\qquad \ltimes\ \mathsf{evens}\,(\phi\,(5 \triangleleft 9 \triangleleft \cdots))) & \\
& \qquad \ltimes\ \mathsf{evens}\,(2 \triangleleft \ \ \mathsf{odds}\,(\phi\,(4 \triangleleft 8 \triangleleft \cdots)) & \\
& \qquad\qquad\qquad\quad \ltimes\ \mathsf{evens}\,(\phi\,(6 \triangleleft 10 \triangleleft \cdots))) & \\
& = 0 \triangleleft \ \ \mathsf{evens}\,(\ \ \mathsf{odds}\,(\phi\,(3 \triangleleft 7 \triangleleft \cdots)) & \text{definition of } \mathsf{evens}, \mathsf{odds} \\
& \qquad\qquad \ltimes\ \mathsf{evens}\,(\phi\,(5 \triangleleft 9 \triangleleft \cdots))) & \\
& \qquad \ltimes\ 2 \triangleleft \mathsf{odds}\,(\ \ \mathsf{odds}\,(\phi\,(4 \triangleleft 8 \triangleleft \cdots)) & \\
& \qquad\qquad\qquad \ltimes\ \mathsf{evens}\,(\phi\,(6 \triangleleft 10 \triangleleft \cdots))) & \\
& = 0 \triangleleft \cdots & \text{stuck!}
\end{array}$$

No more unfoldings of ϕ will generate a second element of the output stream. The computation gets stuck: the expression becomes larger and larger but never produces any output after the head element.

Can we pinpoint a clear reason why ψ seems to work fine while ϕ hangs without producing a result? In the next section we investigate characterizations of (classes of) recursive definitions that are correct, in the sense that they define a total function that will always generate a result when run.

Productivity and Guardedness

Let's start by stating clearly what the problem is. We say that a recursive definition is *productive* if every unfolding generates a new part of the output structure. For example, natsFrom is productive because when we unfold it once on any input number n, we always get a constructor with a head element:

$$\mathsf{natsFrom}\,n \rightsquigarrow n \triangleleft \mathsf{natsFrom}\,(n+1).$$

Clearly every successive unfolding will generate a new element and the whole output sequence can be generated progressively.

On the other hand, wrongStr is not productive because only its first unfolding produces an element, while successive unfoldings fall into a vicious circle:

$$\mathsf{wrongStr} \rightsquigarrow 3 \triangleleft {}^{\mathsf{t}}\mathsf{wrongStr} \rightsquigarrow 3 \triangleleft {}^{\mathsf{t}}(3 \triangleleft {}^{\mathsf{t}}\mathsf{wrongStr}) \rightsquigarrow 3 \triangleleft {}^{\mathsf{t}}\mathsf{wrongStr}.$$

Some definitions are not directly productive, there are some unfoldings that do not generate any element; but successive unfoldings sooner or later result in some output. What matter is that a sufficient number of unfolding always successfully produces something. A simple example is the odds function:

$$\begin{array}{l}
\mathsf{odds}\,(a_0 \triangleleft a_1 \triangleleft a_2 \triangleleft a_3 \triangleleft a_4 \triangleleft \cdots) \rightsquigarrow \mathsf{evens}\,(a_1 \triangleleft a_2 \triangleleft a_3 \triangleleft a_4 \triangleleft \cdots) \\
\rightsquigarrow a_1 \triangleleft \mathsf{odds}\,(a_2 \triangleleft a_3 \triangleleft a_4 \triangleleft \cdots) \rightsquigarrow a_1 \triangleleft \mathsf{evens}\,(a_3 \triangleleft a_4 \triangleleft \cdots) \\
\rightsquigarrow a_1 \triangleleft a_3 \triangleleft \mathsf{odds}\,(a_4 \triangleleft \cdots).
\end{array}$$

Even if the first (and third) unfolding of odds doesn't produce any output, the definition produces an element every two unfoldings anyway. Therefore it is safe and can be said to be productive in an extended sense. In fact, such definitions can always be refactored so that they are directly productive. For example we could define odds in this alternative way:

$$\begin{array}{l} \mathsf{odds} : \mathsf{Stream}_A \to \mathsf{Stream}_A \\ \mathsf{odds}\,\alpha = {}^{\mathsf{ht}}\alpha \lhd \mathsf{odds}\,{}^{\mathsf{tt}}\alpha. \end{array}$$

The definition of ψ is productive and can be put in such a form, although with some technical effort (we'll have a look at it later). On the other hand $\rotatebox[origin=c]{180}{ψ}$ is hopelessly unproductive.

So far we used productivity as an informal notion. We aim to give a formal definition of it, a way to measure exactly how well a definition performs in generating a stream in output. We could assign a number (call it a *modulus of productivity*) characterizing how much input is produced by each unfolding. For example, natsFrom could have a modulus of 1, saying that it produces one output element for each unfolding; odds could have modulus 1/2, saying that it produces one output element for each two unfolding. In fact we may want to be a bit more precise and distinguish between odds and evens: the latter is a bit more productive, because it generates an element already at the first unfolding. This small difference can be vital, as seen in the example of ψ versus $\rotatebox[origin=c]{180}{ψ}$.

TO DO [Define formally productivity and modulus of productivity. Prove it for some of the examples. Later show that productive definitions can always be rewritten as a coalgebra.]

Undecidability of Productivity

It is undecidable whether a recursive definition on streams is productive.

If we consider streams of Boolean values, we can see a stream as the tape of a Turing machine. It is possible to encode the standard operations of Turing machines as recursive equations, then reducing the halting problem to the problem of productivity.

TO DO [Put more detail about Rosu's proof.]

A stronger undecidability result comes from considering a very restricted class of polymorphic stream equations. Consider functions that map streams to streams, independently of what type the elements of the streams have. These are *polymorphic* functions in the sense that they work in the same way independently of the type of streams. All they can do is reorder the elements of the streams. We can introduce a very simple formalism in which only the operations of head and tail are allowed. So we can have a system of equations on several input streams, where the right-hand side of each equation is an expression in the original streams, recursive calls to the functions, head and tail. It turns out that even with this restricted formalism, productivity is undecidable.

TO DO [Write (simplified version of) the proof by Sattler and Balestrieri.]

Guardedness

One syntactic way to ensure productivity is *guardedness*

We say that a recursive definition is *guarded* if the recursive calls occur only as direct arguments of constructors. The type Stream_A has a single constructor $\triangleleft$. The first argument is an element of A (the head), so there cannot be any recursive call in that position. The second argument is a stream, and this is the only place in which a recursive call can occur.

So a guarded defintion of a single stream must have the form:

$$\alpha = \cdots \triangleleft \alpha.$$

The $\cdots$ in the head position can be filled only by a term that doesn't depend on α. Our definition of inf_3 is guarded in this sense.

The following equations are not guarded: even if the recursive calls occur inside an argument of the constructor, they are not *direct* arguments, but occur after the interpositions of other functions:

$$\mathsf{wrongStr} = 3 \triangleleft {}^{\mathsf{t}}\mathsf{wrongStr}, \qquad \alpha = {}^{\mathsf{ht}}\alpha \triangleleft \alpha.$$

In the first equation, guardedness is spoiled by the application of the tail function. We have already seen earlier that this definition is not productive, because it never manages to generate any part of the tail. In the second, the first recursive call is also not guarded by the application of the head and tail functions. This definition never manages to produce anything.

Definitions of single streams can be guarded only if they are constant. However, functions that produce stream are more flexible: we're allowed to recursively apply the function with a different input, as long as it is guarded. One simple example is the natsFrom function, which has the equation

$$\mathsf{natsFrom}\, n = n \triangleleft \mathsf{natsFrom}\, (n+1).$$

The recursive call changes the input to $n+1$ instead of n, but it occurs directly as argument of the constructor $\triangleleft$. Therefore it is guarded and thus productive.

Notice the difference between this guardedness criterion for stream definition and the criteria for recursive definitions by recursion on inductive types. In that case, we allow the recursive calls to appear anywhere, but they must be applied to inputs that are structureally smaller that the original one.

The strict guardedness condition can be difficult to meet in some cases. We can allow a more permissive criterion for guardedness: recursive calls may occur under several constructors, not just one. This helps with the definition of ciclic streams, for example

$$\alpha = 0 \triangleleft 5 \triangleleft 2 \triangleleft \alpha.$$

The recursive call to α occurs under three applications of the constructor $\triangleleft$. This satisfies the extended guardedness rule and the definition can be accepted and is productive.

In general, the intervetion of external functions on the left-hand side of a recursive defintion may spoil guardedness, as we've seen in the examples involving

tail. However, that does not need to be always the case. If the intervening function preserves the productivity behaviour of its arguments, it can be accepted.

For example, let us define the point-wise addition of two streams of natural numbers:

$$\begin{aligned}&(\oplus) : \mathsf{Stream}_{\mathbb{N}} \to \mathsf{Stream}_{\mathbb{N}} \to \mathsf{Stream}_{\mathbb{N}}\\&(a \triangleleft \alpha) \oplus (b \triangleleft \beta) = (a + b) \triangleleft (\alpha \oplus \beta).\end{aligned}$$

This is guarded in a direct way: the occurrence of $\oplus$ on the right-hand side is immediately in the argument position of $\triangleleft$. We can also observe that $\oplus$ preserves the productivity behaviour of its arguments: it produces an element in output for every element in each of the inputs.

So, we can use it in another recursive definition, like this alternative way of defining the streams of natural numbers:

$$\begin{aligned}&\mathsf{nats} : \mathsf{Stream}_{\mathbb{N}}\\&\mathsf{nats} = 0 \triangleleft (\mathsf{nats} \oplus \bar{1}).\end{aligned}$$

Here the recursive call to nats on the right-hand side is not directly guarded, because it is first combined with $\triangleleft 1$ using the point-wise addition operation $\oplus$. However, since $\oplus$ preserves the productivity properties of its arguments, we are guaranteed that it will generate an output element for every element of its argument. We will say that the occurrence of nats is *indirectly guarded* by the constructor $\triangleleft$ through application of $\oplus$.

Another example of a sound recursive definition that respects this extended guardedness criterion is this compact specification of the stream of Fibonacci numbers:

$$\begin{aligned}&\mathsf{fibs} : \mathsf{Stream}_{\mathbb{N}}\\&\mathsf{fibs} = 0 \triangleleft ((1 \triangleleft \mathsf{fibs}) \oplus \mathsf{fibs})\end{aligned}$$

Guardedness, in all its forms, is an attempt to find a concrete decision procedure that allows us (or an algorithm) to determine syntactically if an equation is productive. Since we know that productivity is undecidable, even if we refine further the notion of guardedness, we will never be able to capture all productive definitions.

If, as we suggested earlier, we introduce a *modulus of productivity*, a measurement of how much output a function generates in function of how much input it consumes, we may be even more subtle in determine what equations can be considered guarded.

TO DO [Precise definition of guardedness and proof that it implies productivity.]

Coalgebras

The notion of guardedness is a syntactic property of recursive equations. It ensures that definitions are productive, but it is not complete: there will always be some productive definitions that are not captured even by a very sophisticated extension of the guardedness condition.

Let us move to a more abstract level and use some concept from category theory. We present a way of generating streams that is complete with respect to productivity and much simpler to understand and apply than guardedness. It is not a syntactic check, but an algebraic structure. A format to specify functions that return streams as output.

Imagine a stream producing process: it is a little machine with an internal state; at every stage of its computation it will generate one element of the output stream and change its state. Seen an an algebraic object, such a process is called a *coalgebra*.

Definition 17 *A* coalgebra *(for streams over a type A) is a pair $\langle X, \sigma\rangle$ where*

$$\begin{array}{ll} X \text{ is a type} & \textit{(of states of the process)} \\ \sigma : X \to A \times X & \textit{(the process itself)} \end{array}$$

Equivalently, σ can be given by two separate components:

$$\mathsf{h}_\sigma : X \to A, \qquad \mathsf{t}_\sigma : X \to X.$$

The idea is that, starting with an initial state x_0, the coalgebra goes through a sequence of steps in which it generates the next element of the output stream using h_σ and changes the state using t_σ.

$$\begin{tikzcd}
x_0 \arrow[r, "\mathsf{t}_\sigma"] \arrow[d, dotted, "\mathsf{h}_\sigma"] & x_1 \arrow[r, "\mathsf{t}_\sigma"] \arrow[d, dotted, "\mathsf{h}_\sigma"] & x_2 \arrow[r, "\mathsf{t}_\sigma"] \arrow[d, dotted, "\mathsf{h}_\sigma"] & \cdots \\
a_0 & a_1 & a_2 & \cdots
\end{tikzcd}$$

Unfolding a coalgebra generates a stream. We can express this with a guarded equation:

$$\begin{array}{l} \mathsf{unfold}_\sigma : X \to \mathsf{Stream}_A \\ \mathsf{unfold}_\sigma\, x_0 = (\mathsf{h}_\sigma\, x_0) \lhd \mathsf{unfold}_\sigma\, (\mathsf{t}_\sigma\, x_0). \end{array}$$

In general, we have a single function producing the stream generated by any coalgebra (given as its two components):

$$\begin{array}{l} \mathsf{unfold} : (X \to A) \to (X \to X) \to X \to \mathsf{Stream}_A \\ \mathsf{unfold}\, h\, t\, x_0 = (h\, x_0) \lhd \mathsf{unfold}\, h\, t\, (t\, x_0). \end{array}$$

This equations are guarded and therefore productive. We can formulate this in algebraic and diagrammatic terms.

Theorem 18 *For every coalgebra $\langle X, \sigma\rangle$, there exists a unique function (called* anamorphism*) $\widehat{\sigma} : X \to \mathsf{Stream}_A$ such that the following diagram commutes:*

$$\begin{tikzcd}
\mathsf{Stream}_A \arrow[rr, "\langle \mathsf{head}, \mathsf{tail}\rangle"] & & A \times \mathsf{Stream}_A \\
X \arrow[u, dotted, "\widehat{\sigma}"] \arrow[rr, "\sigma", "\langle \mathsf{h}_\sigma, \mathsf{t}_\sigma\rangle"'] & & A \times X \arrow[u, "\mathsf{id}_A \times \widehat{\sigma}"']
\end{tikzcd}$$

$\langle \mathsf{head}, \mathsf{tail}\rangle \circ \widehat{\sigma} = (\mathsf{id}_A \times \widehat{\sigma}) \circ \sigma$;
equivalently, for every $x : X$,
$\mathsf{head}\, (\widehat{\sigma}\, x) = \mathsf{h}_\sigma\, x$
$\mathsf{tail}\, (\widehat{\sigma}\, x) = \widehat{\sigma}\, (\mathsf{t}_\sigma\, x)$.

The anamorphism $\widehat{\sigma}$ is the function computed by the program unfold_σ.

Examples of coalgebraic presentations of functions that we have seen previously are:

- The function $\mathsf{natsFrom} : \mathbb{N} \to \mathsf{Stream}_\mathbb{N}$ generating the stream of numbers starting from a ginven initial point is definable as the anamorphism of the following coalgebra:

$$\begin{array}{l} \mathsf{fromCoalg} : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \\ \mathsf{fromCoalg}\, n = \langle n, n+1 \rangle \\ \\ \mathsf{natsFrom} = \widehat{\mathsf{fromCoalg}}. \end{array}$$

- The Fibonacci stream is the application of an anamorphism two a pair of initial values (with different initial values we get other Fibonacci-like streams):

$$\begin{array}{l} \mathsf{fibCoalg} : \mathbb{N}^2 \to \mathbb{N} \times \mathbb{N}^2 \\ \mathsf{fibCoalg}\, \langle n, m \rangle = \langle n, \langle m, n+m \rangle \rangle \\ \\ \mathsf{fib}_{\mathsf{from}} : \mathbb{N}^2 \to \mathsf{Stream}_\mathbb{N} \\ \mathsf{fib}_{\mathsf{from}} = \widehat{\mathsf{fibCoalg}} \\ \\ \mathsf{fibs} = \mathsf{fib}_{\mathsf{from}}\, \langle 0, 1 \rangle. \end{array}$$

Of course, this does not get around the undecidability problem, it just puts the burden on the user to

Exercise 19 *Try to define the function ψ using the anamorphism of a coalgebra. So far we just experimented with it: we implemented in our favourite programming language and run it on the stream of natural numbers. It seems to be productive, but it is not obviously guarded. If we can redefine it as an anamorphism, we would have the mathematical certainty that it is total. However, it is not completely obvious how to do it. The secret is to invent an appropriate state set X on which to define a coalgebra. In this case, X cannot just be the intended domain of ψ, Stream_A, but should be a larger space that includes it. Finding the right one is a tricky exercise, but it is instructive to illuminate how this can be done in general.*

History and Literature

The undecidability of productivity of recursive stream equations on Booleans was shown by Rosu [2]

The undecidability of pure stream equations was proven by Sattler and Balestrieri [3].