Midlands Graduate School 2013

Coalgebras and Infinite Data Structures

Venanzio Capretta

Lecture 1: Introduction to infinite data

- Computers (and people) have finite memory
  They can't store infinite information
  There can't be infinite objects
  But we can represent them by finite information

- Circular structures:
  the stream (infinite list)
  [0, 0, 0, ...]
  can be represented by a circular linked list

The list \([0, 5, 2, 0, 5, 2, 0, 5, 2, ...]\)
can be represented by

- How do we represent non-cyclical lists?
  \([0, 1, 2, 3, ...]\)

We can specify an operation on the links:

Unfolding:

- In general:
  Give a program that generates an infinite list by steps.
  Such programs are called coalgebras
**Lazy Functional Programming**

*Haskell*

- Higher-order types:
  Functions can be given as input

- Recursive data:
  No distinction between finite and infinite
  Lists may be finite or go on forever

- Lazyness:
  Values are computed only when needed
  We can define an infinite object
  It will never be fully computed
  When (finite) parts of it are needed, they will be unfolded

**Type Theory and Proof Assistants**

*Coq* (Calculus of Inductive Constructions)

- Dependent types:
  The type of an object can depend on the value of another object

- Distinction between
  Inductive types (well-founded objects)
  Coinductive types (non-well-founded)

- Normalization:
  Every closed term reduces to canonical form (value)
  No lazyness, no divergence

- Curry-Howard correspondence
  Logic can be represented:
  Propositions $\Rightarrow$ Types
  Proofs $\Rightarrow$ Programs
Coinductive Types

They specify potentially non-well-founded structures.

Given by rules/constructors.

Streams

Let \( A \) be a type.

The type \( S_A \) of streams over \( A \) is defined by the rule

\[
\begin{align*}
\text{a: } & A \quad \text{s: } & S_A \\
\hline
\text{a<s: } & S_A
\end{align*}
\]

It says:

If \( a \) is an element of \( A \) and \( s \) is a (previously) constructed stream;

I can make a new stream \( a<s \).

Of course: \( s \) must have been constructed using the same rule from an \( a': A \) and \( s': S_A \). \( s = a'<s'! \)

Similarly for \( s' \).

Infinite regression.

How can I ever define a stream?

Recursive definitions:

\[
\begin{align*}
\text{allthree: } & S_N \\
\text{allthree = 3 < allthree}
\end{align*}
\]

we can use the stream we're defining in its own definition

Pattern matching:

Since every stream must have been constructed by the rule, we can assume that it is in constructor form

\[
\begin{align*}
\text{head: } & S_A \rightarrow A \\
\text{tail: } & S_A \rightarrow S_A
\end{align*}
\]

\[
\begin{align*}
\text{head (a<s)} & = a \\
\text{tail (a<s)} & = s
\end{align*}
\]

We write \( ^hS \) for \( \text{(head s)} \)

\( ^tS \) for \( \text{(tail s)} \)

Not all recursive definitions are sound:

Wrong stream: \( S_N \)

Wrong stream = 3 < ^t \text{wrongstream}
Productivity

A recursive definition is productive when it keeps generating new elements.

As we unfold the definition, longer and longer stretches of the result appear.

\( \Upsilon \) is productive, \( \Lambda \) is not productive.

Productivity is an informal notion.
How can we make it precise?

Productivity is undecidable
(Balestrieri/Sattler)
One way to guarantee productivity is **guardedness**

A recursive definition is guarded if the recursive calls occur only as direct arguments of constructors.

$S_A$ has a single constructor: $\lt$

**Guarded definition:**

$$S = \ldots \lt S$$

- this recursive call is guarded by the constructor
- no recursive calls

**Non-guarded equations:**

$$S = 3 \lt t_S$$

- not a direct argument of the constructor; the function tail is applied to it

$$S = h(t_S) \lt S$$

- not a direct argument, similarly

We allow the recursive calls to occur under several constructors:

$$S = 0 \lt 5 \lt 2 \lt S$$

The define object can be a function.
The recursive calls must be guarded but can be applied to any argument.

$$\text{from : } \mathbb{N} \longrightarrow S_N$$

$$\text{from } n = n \lt \text{ from } (n+1)$$

- argument of recursive call larger than original argument
- recursive call guarded by constructor

**Argument of recursive call guarded by three constructors**

$$\text{nat : } S_N$$

$$\text{nat = from 0}$$
More permissive guardedness:
Recursive calls may occur not as direct arguments of constructors but as arguments of operators as long as those operators produce an element of output for every element of the inputs.

The pointwise addition of streams is such an operator.

\[
(+) : S^N \rightarrow S^N \rightarrow S^N
\]

\[
(\langle x, xs \rangle + \langle y, ys \rangle) = \langle x+y, (xs+ys) \rangle
\]

\[\text{addition on streams}\]

\[\text{addition on numbers}\]

need only the first elements of its inputs \((x, y)\) to determine the first element of its output \((x+y)\)

So the following definition is OK:

\[
\text{nat} : S^N
\]

\[
\text{nat} = 0 \prec (\text{nat} + 1)
\]

\[\text{indirectly guarded by the constructor } \prec, \text{ through application of the preserving operator } +\]

Similarly:

\[
\text{fib} : S^N
\]

\[
\text{fib} = 0 \prec ((1 \prec \text{fib}) + \text{fib})
\]

\[\text{indirectly guarded.}\]

A further generalization introduces the notion of modulus of productivity, a numerical relation between size of the input needed to compute a given size of the output. Then deleting functions (evens, odds) can be used in combination with increasing functions (\(\times\)).
Coalgebras

A coalgebra (for $S_A$) is a pair $(X, \alpha)$ where

$X$ is a type (of states)

$\alpha : X \rightarrow A \times X$ (transition function)

Equivalently we can give the two components of $\alpha$:

$h_\alpha : X \rightarrow A$, $t_\alpha : X \rightarrow X$

Idea: given an initial state $x_0 : X$

the coalgebra produces an element of the output stream

$a_0 = h_\alpha x_0$

an a new state from which to continue the computation

$x_1 = t_\alpha x_0$

unfold : $X \rightarrow S_A$

$\text{unfold } x_0 = (h_\alpha x_0) \prec \text{unfold } (t_\alpha x_0)$

recursive call

guarded by constructor

Doing this for any coalgebra

$\text{unfold} : (X \rightarrow A) \rightarrow (X \rightarrow X) \rightarrow X \rightarrow S_A$

$\text{unfold } h \ t \ x = (h x) \prec (\text{unfold } h \ t \ (t x))$

Diagram:

There exists a unique function $\hat{\alpha}$

making the diagram commute:

$\langle \text{head, tail} \rangle \circ \hat{\alpha} = (\text{id}_A \times \hat{\alpha}) \circ \alpha$

i.e.: $\text{head } (\hat{\alpha} x) = h_\alpha x$

$\text{tail } (\hat{\alpha} x) = \hat{\alpha} (t_\alpha x)$

$\hat{\alpha}$ is the anamorphism

associated with the coalgebra $(X, \alpha)$
Examples:

- The function from is the anamorphism of the coalgebra
  \[ f_{\text{coalg}} : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \]
  \[ f_{\text{coalg}} \ n = \langle n, n+1 \rangle \]

- The Fibonacci sequence can be defined by an anamorphism:
  \[ \text{fib}_{\text{coalg}} : \mathbb{N}^2 \to \mathbb{N} \times \mathbb{N}^2 \]
  \[ \text{fib}_{\text{coalg}} \ <n, m> = \langle n, \langle m, n+m \rangle \rangle \]

\[ \text{fib}_{\text{from}} : \mathbb{N}^2 \to \mathbb{S}_\mathbb{N} \]

\[ \text{fib}_{\text{from}} = \text{fib}_{\text{coalg}} \]

\[ \text{fib} = \text{fib}_{\text{from}} \ <0, 1> \]

Exercise:
Define \( \mathcal{Y} \) using the anamorphism of a coalgebra.

Infinite Binary Trees

\( T_A \) is the Coinductive Type of binary trees with nodes labelled by elements of \( A \).

Rule:

\[ a : A \quad t_1 : T_A \quad t_2 : T_A \]

\[ \text{node}(a, t_1, t_2) : T_A \]

An element of \( T_A \) looks like this:

Notions of: Productivity, Guardedness, Coalgebra and Anamorphism can be given for \( T_A \).
Examples of guarded definitions:

- Tree with the same element on all nodes
  \[ \text{repeat} : A \rightarrow T_A \]
  \[ \text{repeat} a = \text{node } a \ (\text{repeat} a) \ (\text{repeat} a) \]

- Increasing the label with depth
  \[ \text{depthtree} : N \rightarrow T_N \]
  \[ \text{depthtree } n = \text{node } n \ (\text{depthtree } n+1) (\text{depthtree } n+1) \]

- Mirror image of a given tree
  \[ \text{mirror} : T_A \rightarrow T_A \]
  \[ \text{mirror } (\text{node } a \ t_1 \ t_2) = \text{node } a \ (\text{mirror } t_2) \ (\text{mirror } t_1) \]

Productivity for Trees:
A productive definition must produce the elements at a certain depth after a finite number of steps.

It is not enough that it keeps producing new parts of the structure.
It must be productive "in every direction".

Counterexample:
\[ \text{wrongtree} : N \rightarrow T_N \]
\[ \text{wrongtree } n = \text{node } n \ (\text{wrongtree } n+1) \]
\[ \ (\text{right } (\text{wrongtree } n)) \]

- \text{wrongtree } n
  - will produce elements on the left branches
    but not on the right
- Still, because Haskell is lazy
  \[ \text{espine } (\text{wrongtree } n) \]
  - is productive

Haskell programming with Binary Trees
Guardedness for trees

Recursive call must occur only as direct arguments of the node constructor:

\[ t = \text{node } x \ x \ t \ t \]

\[ \text{recursive calls guarded by constructor} \]

Function definition:

\[ tf : X \rightarrow T_A \]

\[ tf \ x = \text{node } a_x (tf x_1) (tf x_2) \]

Similarly to lists,
we can define more permissive notions of guardedness.

Coalgebras for Trees

A coalgebra (for \( T_A \)) is a pair \((X, \alpha)\)

\[ \alpha : X \rightarrow A \times X^2 \]

Equivalently:

\[ n_\alpha : X \rightarrow A \]

\[ \ell_\alpha : X \rightarrow X, \ r_\alpha : X \rightarrow X \]

unfold : \( X \rightarrow T_A \)

\[ \text{unfold } x = \text{node } (n_\alpha x) \ (\text{unfold } (\ell_\alpha x)) \ (\text{unfold } (r_\alpha x)) \]

Diagram:

\[ T_A \xrightarrow{\alpha} A \times T_A \]

\[ T_A \xrightarrow{\text{label, left, right}} A \times T_A \]

\[ \text{left } (\hat{\alpha} x) = \hat{\alpha} (\ell_\alpha x) \]

\[ \text{right } (\hat{\alpha} x) = \hat{\alpha} (r_\alpha x) \]