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

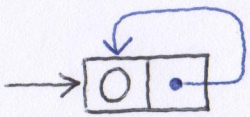
  But we can represent them by finite information

- Circular structures:

  the stream (infinite list)

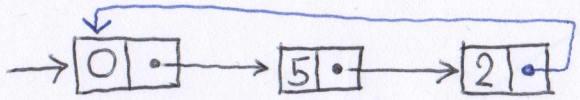
  [0, 0, 0, ...]

  can be represented by a circular linked list



The list [0, 5, 2, 0, 5, 2, 0, 5, 2, ...]

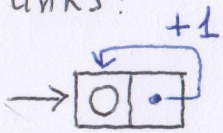
can be represented by



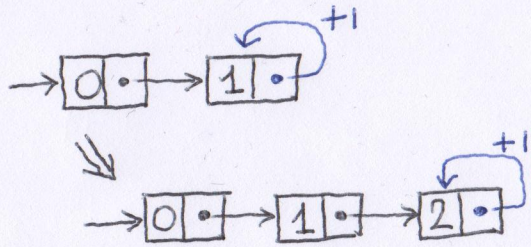
- How do we represent non-cyclical lists?

  [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 $\$_A$ of streams over A is defined by the rule

$$\frac{a : A \qquad s : \$_A}{a \prec s : \$_A}$$

It says:

If a is an element of A and

s is a (previously) constructed stream;

I can make a new stream $a \prec s$.

Of course: s must have been constructed using the same rule from an $a' : A$ and $s' : \$_A : s = a' \prec s'$. Similarly for s'.

Infinite regression.

How can I ever define a stream?

Recursive definitions:

$allthree : \$_{\mathbb{N}}$

$allthree = 3 \prec allthree$

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

$head : \$_A \longrightarrow A \qquad tail : \$_A \longrightarrow \$_A$

$head\ (a \prec s) = a \qquad tail\ (a \prec s) = s$

We write ^{h}s for $(head\ s)$

^{t}s for $(tail\ s)$

Not all recursive definitions are sound:

$wrongstream : \$_{\mathbb{N}}$

$wrongstream = 3 \prec {}^{t}wrongstream$

• Haskell Programming with Streams

When does a recursive definition generate all elements of a stream?

We've seen two very similar definitions:

$$\Upsilon : \$_A \longrightarrow \$_A$$

$$\Upsilon\ s = {}^h s \prec \text{evens}(\Upsilon(\text{odds}\ {}^t s)) \bowtie \text{odds}(\Upsilon(\text{evens}\ {}^t s))$$

This seens to be OK: given a stream s it generates a stream Υs

$$\perp : \$_A \longrightarrow \$_A$$

$$\perp\ s = {}^h s \prec \text{odds}(\perp(\text{evens}\ {}^t s)) \bowtie \text{evens}(\perp(\text{odds}\ {}^t s))$$

This doesn't work: $\perp s$ gets stuck

What's the difference?

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.

Υ is productive, $\perp$ 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.

$\$_A$ has a single constructor: $<$

Guarded definition:

$$s = \ldots < s$$

↑ no recursive calls

↑ this recursive call is guarded by the constructor

Non-guarded equations:

$$s = 3 < {}^t s$$

↑ not a *direct* argument of the constructor: the function tail is applied to it

$$s = {}^h({}^t s) < s$$

↑ not a direct argument, similarly

We allow the recursive calls to occur under several constructors:

$$s = 0 < 5 < 2 < s$$

↑ recursive call guarded by three constructors

The define object can be a function.

The recursive calls must be guarded but can be applied to any argument.

from : $\mathbb{N} \longrightarrow \$_{\mathbb{N}}$

from n = n < from (n+1)

↑ recursive call guarded by constructor

↑ argument of recursive call larger than original argument

OK

nat : $\$_{\mathbb{N}}$

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.

$$(+) : \$_{\mathbb{N}} \longrightarrow \$_{\mathbb{N}} \longrightarrow \$_{\mathbb{N}}$$

$$(x < xs) + (y < ys) = (x+y) < (xs + ys)$$

($+$ on streams; addition on numbers)

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

So the following definition is OK:

$$nat : \$_{\mathbb{N}}$$

$$nat = 0 < (nat + 1)$$

(1: stream of 1s.)

(nat: indirectly guarded by the constructor $<$, throug application of the preserving operator $+$)

Similarly:

$$fib : \$_{\mathbb{N}}$$

$$fib = 0 < ((1 < fib) + fib)$$

(fib: indirectly guarded.)

A further generalization introduces the notion of modulus of productivity, a numerical relation between size of the input needed to compute a give size of the output.

Then deleting functions (evens, odds) can be used in combination with increasing functions ($\nearrow$).

A more abstract notion of productive definitions

Coalgebras

A coalgebra (for $\$_A$) is a pair (X, α) where

X is a type (of states)

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

Equivalently we can give the two components of α:

$$h_\alpha : X \longrightarrow A, \quad t_\alpha : X \longrightarrow X$$

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

$$x_1 = t_\alpha\, x_0$$

$$\text{unfold} : X \longrightarrow \$_A$$

$$\text{unfold}\ x_0 = (h_\alpha\, x_0) :< \text{unfold}\ (t_\alpha\, x_0)$$

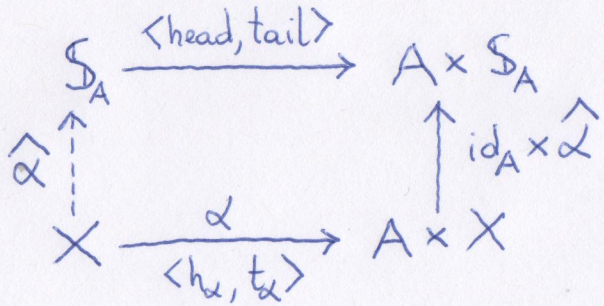
↑ recursive call guarded by constructor

Doing this for any coalgebra

$$\text{unfold} : (X \to A) \to (X \to X) \to X \to \$_A$$

$$\text{unfold}\ h\ t\ x = (h\, x) :< (\text{unfold}\ h\ t\ (t\, x))$$

Diagram:



There exists a unique function $\hat{\alpha}$ making the diagram commute:

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

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, α)

Examples:

- The function from is the anamorphism of the coalgebra

$$fcoalg : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$$

$$fcoalg\ n = \langle n, n+1 \rangle$$

- The Fibonacci sequence can be defined by an anamorphism:

$$fibcoalg : \mathbb{N}^2 \longrightarrow \mathbb{N} \times \mathbb{N}^2$$

$$fibcoalg\ \langle n, m \rangle = \langle n, \langle m, n+m \rangle \rangle$$

$$fibfrom : \mathbb{N}^2 \longrightarrow \$_{\mathbb{N}}$$

$$fibfrom = \overbrace{fibcoalg}$$

$$fib = fibfrom\ \langle 0, 1 \rangle$$

Exercise:

Define Υ using the anamorphism of a coalgebra.

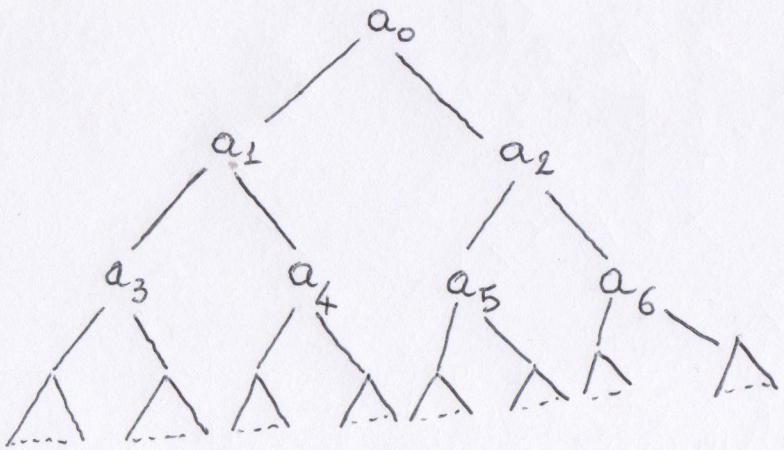
Infinite Binary Trees

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

Rule:

$$\frac{a : A \quad t_1 : T_A \quad t_2 : T_A}{node(a, t_1, t_2) : T_A}$$

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

repeat : $A \longrightarrow T_A$

repeat a = node a (repeat a) (repeat a)

- Increasing the label with depth

depthtree : $\mathbb{N} \longrightarrow T_{\mathbb{N}}$

depthtree n = node n (depthtree n+1) (depthtree n+1)

- Mirror image of a given tree

mirror : $T_A \longrightarrow T_A$

mirror (node x t_1 t_2) = node x (mirror t_2) (mirror t_1)

- Haskell programming with Binary Trees

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:

wrongtree : $\mathbb{N} \longrightarrow T_{\mathbb{N}}$

wrongtree n = node n (wrongtree n+1) (right (wrongtree n))

- wrongtree n will produce elements on the left branches but not on the right

- Still, because Haskell is lazy

espine (wrongtree n) is productive

Guardedness for trees

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

$t = \text{node}\ x\ t\ t$

recursive calls guarded by constructor

Function definition:

$tf : X \longrightarrow T_A$

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

guarded

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

Coalgebras for Trees

A coalgebra (for T_A) is a pair (X, α)

X is a type

$\alpha : X \longrightarrow A \times X^2$

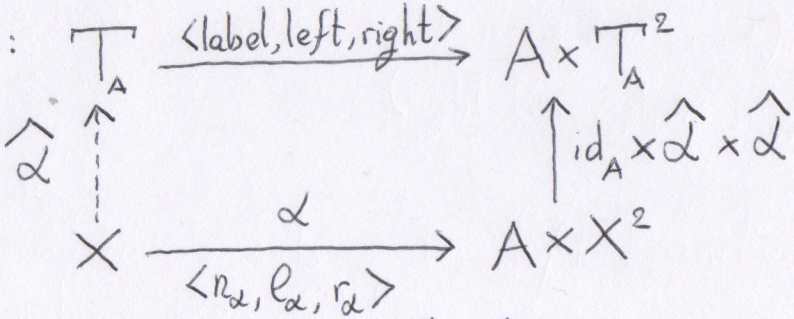
Equivalently: $n_\alpha : X \longrightarrow A$

$\ell_\alpha : X \longrightarrow X$, $r_\alpha : X \longrightarrow X$

$unfold : X \longrightarrow T_A$

$unfold\ x = \text{node}\ (n_\alpha x)\ (unfold\ (\ell x))\ (unfold\ (r x))$

Diagram:



$\langle label, left, right \rangle \circ \hat{\alpha} = (id_A \times \hat{\alpha} \times \hat{\alpha}) \circ \langle n_\alpha, \ell_\alpha, r_\alpha \rangle$

i.e.: $label\ (\hat{\alpha}\ x) = n_\alpha x$

$left\ (\hat{\alpha}\ x) = \hat{\alpha}\ (\ell_\alpha\ x)$

$right\ (\hat{\alpha}\ x) = \hat{\alpha}\ (r_\alpha\ x)$.