MGS 2013: Coalgebras and Infinite Data Structures

Venanzio Capretta

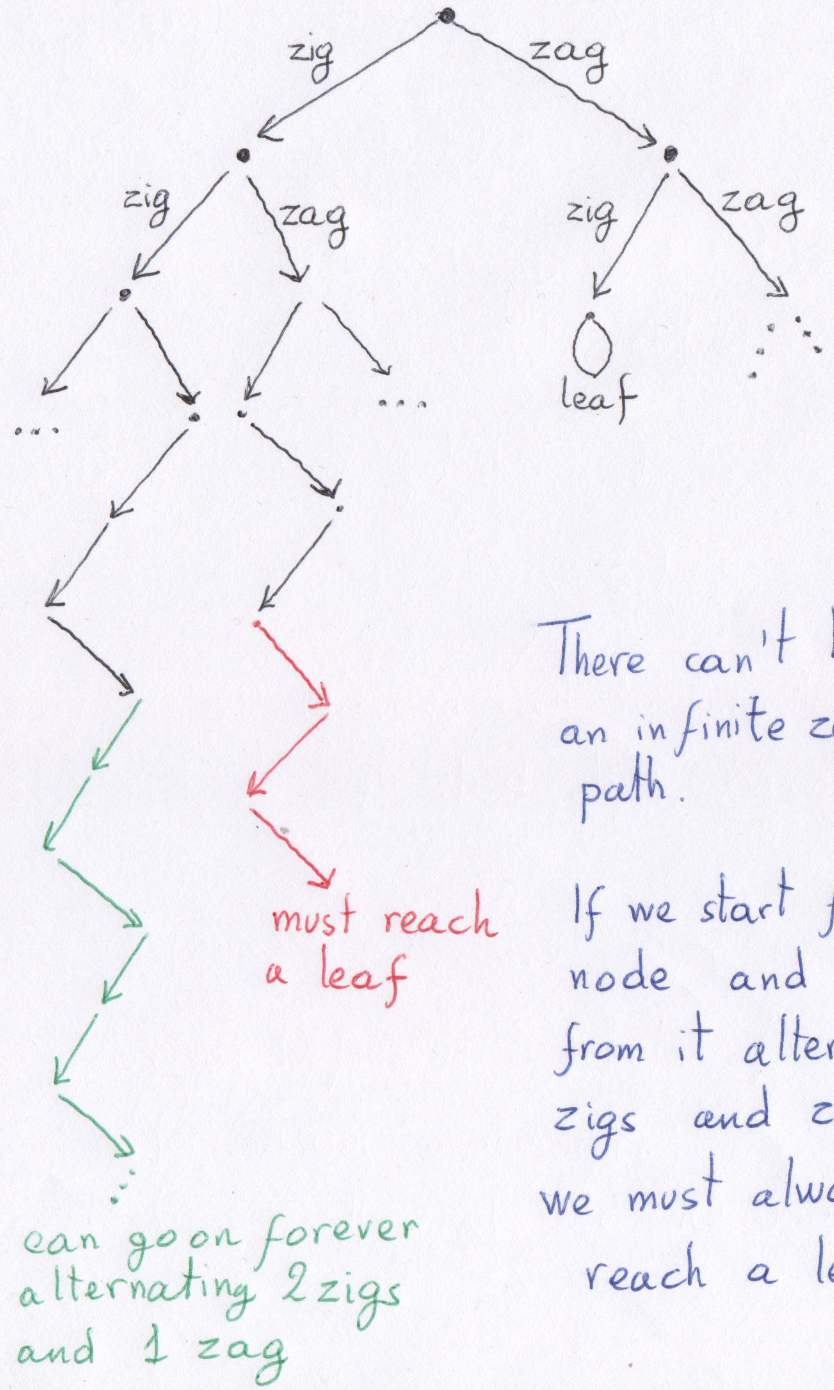
Lecture 4: Wander Types

We want to define a type of binary trees called ZigZag.

Trees are not necessarily well-founded there can be infinite paths.

But we impose a restriction:

There can't be an infinite path that keep alternating left and right turns



There can't be an infinite zigzaggin path.

If we start from any node and descend from it alternating zigs and zags we must always reach a leaf.

We can realize this type by simultaneously defining it together with two functions computing the number of consecutive zigzags.

Wander Type ZigZag : Set

  Leaf : ZigZag

  node : ZigZag $\longrightarrow$ ZigZag $\longrightarrow$ ZigZag

zigs : ZigZag $\longrightarrow \mathbb{N}$

zags : ZigZag $\longrightarrow \mathbb{N}$

  zigs leaf = 0

  zags leaf = 0

  zigs (node t_1 t_2) = zags t_1 + 1

  zags (node t_1 t_2) = zigs t_2 + 1

The type ZigZag must be interpreted coinductively: the tree may not be wellfounded.

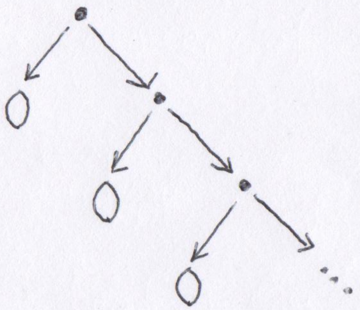
If the functions zigs and zags were defined after the type, they would be incorrect: no guarantee of result.

They are defined simultaneously: only trees for which zigs and zags are well-defined belong to ZigZag.

Examples:

- $t_1 = \text{node leaf } t_1$

  is a well-defined zigzag tree



$\text{zigs } t_1 = \text{zags leaf} + 1 = 1$

$\text{zags } t_1 = \text{zigs } t_1 + 1 = 2$

- $t_2 = \text{node } t_2\ t_2$

  is not well-defined:

  $\text{zigs } t_2 = \text{zags } t_2 + 1 = \text{zigs } t_2 + 2$

  $\text{zags } t_2 = \text{zigs } t_2 + 1$

  this has no solution in the natural numbers.

  ($\text{zigs } t_2$) and ($\text{zags } t_2$) are undefined

  So t_2 is not a well-defined zigzag tree

And yet the equation for t_2 satisfies the guardedness condition

We need a stronger notion:

Guardedness by values:
the constructor guard must also specify the values of the functions.

(But we will continue informally for now.)

- Example of mutual definition

  $t_3 = \text{node } t_3\ t_4 \qquad t_4 = \text{node leaf } t_3$

  $\text{zigs } t_3 = \text{zags } t_3 + 1 = 3$

  $\text{zags } t_3 = \text{zigs } t_4 + 1 = 2$

  $\text{zigs } t_4 = \text{zags leaf} + 1 = 1$

  $\text{zags } t_4 = \text{zigs } t_3 + 1 = 4$

Draw the trees t_3 and t_4 and verify that to avoid leaves, the zags alway come in twos

Wander Types are a coinductive version of

simultaneous inductive/recursive definitions

They first appeared in Per Martin-Löf's universes à la Tarski.

Universe: A type whose elements are (codes for) types

For example: define a universe containing
the empty type, the unit type,
the natural numbers
and closed under sums, products and function types

Inductive U: Set

$zr : U$

$un : U$

$nt : U$

$pr : U \to U \to U$

$sm : U \to U \to U$

$fn : U \to U \to U$

U is a set of codes.

Decoding function by large elimination:

$El : U \longrightarrow Set$ the result is a set

$El\ zr = \emptyset$

$El\ un = \{\bullet\}$

$El\ (pr\ t_1\ t_2) = (El\ t_1) \times (El\ t_2)$

$El\ (sm\ t_1\ t_2) = (El\ t_1) + (El\ t_2)$

$El\ (fn\ t_1\ t_2) = (El\ t_1) \longrightarrow (El\ t_2)$

U is a normal simple inductive type

El is defined by (large) recursion on it

Exercise: What if we made U coinductive, instead of inductive.

Then we could leave nt out and define it by guarded recursion:

$$nt = sm\ un\ nt$$

We could also define, for $a : U$ the code of the type of streams:

$$st_a = pr\ a\ st_a$$

Can you see where the problem with a coinductive universe is?

This universe contains only non-dependent types. We want to add dependent sums and products.

$A : \text{Set}$ $\quad$ $B : A \longrightarrow \text{Set}$

dependent type:

Ba is a set for every $a : A$

Dependent Product:

$$\prod x : A.\ Bx$$

its elements are functions that map each $a : A$ to an element of Ba

Dependent Sum:

$$\sum x : A.\ Bx$$

its elements are pairs $\langle a, b \rangle$ with $a : A$, $b : Ba$

We now try to extend U with these two type constructors:

Inductive U : Set

⋮

pi : $(a : U) \to (El\,a \to U) \to U$

sg : $(a : U) \to (El\,a \to U) \to U$

$El : U \longrightarrow Set$

⋮

$El\,(pi\ a\ b) = \Pi x : El\,a.\ El\,(b\,x)$

$El\,(sg\ a\ b) = \Sigma x : El\,a.\ El\,(b\,x)$

We used E in the type of the constructors pi and sg of U before it is defined

U and El are simultaneously mutually defined

Eric Palmgren
generalized the construction to a type operator creating a universe closed under given type formers.

P. Dybjer
formulated the general notion of induction/recursion with syntactic check guaranteeing soundness.

Dybjer/Setzer
a type of codes for acceptable inductive/recursive definitions.

Other examples:

- Catarina Coquand: Freshness Lists (lists without repetitions)

  $\mathrm{List}_A : \mathrm{Set}$

  $\quad \mathrm{nil} : \mathrm{List}_A$

  $\quad \mathrm{cons} : (a : A) \to (\ell : \mathrm{List}_A) \to \mathrm{Fresh}\ \ell\ a \to \mathrm{List}_A$

  $\mathrm{Fresh} : \mathrm{List}_A \to A \to \mathrm{Prop}$

  $\quad \mathrm{Fresh}\ \mathrm{nil}\ a = \mathrm{True}$

  $\quad \mathrm{Fresh}\ (\mathrm{cons}\ a_0\ \ell)\ a = a_0 \neq a \wedge \mathrm{Fresh}\ \ell\ a$

- Nested General Recursion

- Leftist Heaps (Priority Queues)

Wander Types are a coinductive version of induction/recursion.

We simultaneously define:

- A coinductive type
- A recursive function on it.

The same syntactic restrictions as for IR types guarantee soundness.

Dybjer/Setzer codes can be used.

Some usual type constructions can be realized by ~~w~~ wander types.

Mixed Induction - Coinduction:

We can use the function component to impose ~~~~ wellfoundedness of certain constructors.

Example:

Type of streams of zeros and ones with no infinite sequence of ones.

Wander Type ZeroOne : Set

$$\mathrm{Zero} : \mathrm{ZeroOne} \longrightarrow \mathrm{ZeroOne}$$
$$\mathrm{One} : \mathrm{ZeroOne} \longrightarrow \mathrm{ZeroOne}$$

$$\mathrm{count1} : \mathrm{ZeroOne} \longrightarrow \mathbb{N}$$
$$\mathrm{count1}\ (\mathrm{Zero}\ s) = 0$$
$$\mathrm{count1}\ (\mathrm{One}\ s) = \mathrm{count1}\ s + 1$$

count1 counts the number of Ones before the next Zero. They must be finite.

This is equivalent to requiring that

Zero is a coinductive constructor

One is an inductive constructor

Exercise:

We previously defined a mixed inductive/coinductive type

$\mathrm{SProc}_{A,B}$ of stream processors.

Can you use the previous ideas to formalize it as a wander type?

IR types and Wander types can't be defined in Coq (OK in Agda)

But Conor McBride invented a way to encode ~~the~~ IR definitions which works also for Wander types.

Idea: Instead of a single type define a family indexed on the result of the functional component.

Example:

The ZeroOne type can be encoded as a family FamZO

Idea: an element s : ZeroOne with (count1 s) = n is encoded as an element of (FamZO n)

Coinductive FamZO : $\mathbb{N} \longrightarrow$ Set

fZero : $(n : \mathbb{N})\,(\text{FamZO } n) \to (\text{FamZO } 0)$

fOne : $(n : \mathbb{N})\,(\text{FamZO } n) \to (\text{FamZO } (n+1))$

Then we can define a single type by dependent sum

$\text{ZeroOne} = \Sigma n : \mathbb{N}.\ \text{FamZO } n$

$\text{Zero} : \text{ZeroOne} \longrightarrow \text{ZeroOne}$

$\text{Zero } \langle n, x \rangle = \langle 0, \text{fZero } x \rangle$

$\text{One} : \text{ZeroOne} \longrightarrow \text{ZeroOne}$

$\text{One } \langle n, x \rangle = \langle n+1, \text{fOne } x \rangle$

$\text{count1} : \text{ZeroOne} \longrightarrow \mathbb{N}$

$\text{count1 } \langle n, x \rangle = n$

Coq Formalization