Lecture 4: Wander Types

We want to define a type of binary trees called Zig Zag.

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 \to\text{ZigZag} \to\text{ZigZag}

zigs : ZigZag \to \mathbb{N}
zags : ZigZag \to \mathbb{N}

\begin{align*}
\text{zigs leaf} &= 0 \\
\text{zags leaf} &= 0 \\
\text{zigs (node } t_1, t_2) &= \text{zags } t_1 + 1 \\
\text{zags (node } t_1, t_2) &= \text{zigs } t_2 + 1
\end{align*}

The type ZigZag must be interpreted coinductively: the tree may not be well-founded.

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:

- $t_1 =$ node leaf $t_1$
  - is a well-defined zigzag tree
    - zigs $t_1 = \text{zags leaf} + 1 = 1$
    - zags $t_1 = \text{zigs} t_1 + 1 = 2$

- $t_2 =$ node $t_2$ $t_2$
  - is not well-defined:
    - zigs $t_2 = \text{zags} t_2 + 1 = \text{zigs} t_2 + 2$
    - zags $t_2 = \text{zigs} t_2 + 1$

  This has no solution in the natural numbers.
  (zigs $t_2$) and (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
  - $t_3 =$ node $t_3$ $t_4$
  - $t_4 =$ node leaf $t_3$

  - zigs $t_3 = \text{zags} t_3 + 1 = 3$
  - zags $t_3 = \text{zigs} t_4 + 1 = 2$
  - zigs $t_4 = \text{zags leaf} + 1 = 1$
  - 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 always 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
- $zr : U$
- $un : U$
- $nt : U$
- $pr : U \rightarrow U \rightarrow U$
- $sm : U \rightarrow U \rightarrow U$
- $fn : U \rightarrow U \rightarrow U$

$U$ is a set of codes.

Decoding function by large elimination:

$El : U \rightarrow \text{Set}$

$El \ zr = \emptyset$
$El \ un = \{\ast\}$
$El \ (pr \ t_1 \ t_2) = (El \ t_1) \times (El \ t_2)$
$El \ (sm \ t_1 \ t_2) = (El \ t_1) + (El \ t_2)$
$El \ (fn \ t_1 \ t_2) = (El \ t_1) \rightarrow (El \ t_2)$
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:
    \[ nt = sm \, vn \, nt \]
    We could also define, for \( a : U \)
    the code of the type of streams:
    \[ st_a = \text{pr} \, a \, \text{sta} \]

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 : Set
B : A \to Set
    dependent type:
    \( Ba \) is a set for every \( a : A \)

Dependent Product:
\[ \Pi x : A . B x \]
its elements are functions that map
    each \( a : A \) to an element of \( Ba \)

Dependent Sum:
\[ \Sigma x : A . B x \]
its elements are pairs \( \langle a, b \rangle \)
    with \( a : A \), \( b : Ba \)
We now try to extend $U$ with these two type constructors:

\[
\text{Inductive } U : \text{Set} \\
\pi : (a : U) \rightarrow (\text{El} a \rightarrow U) \rightarrow U \\
\text{sg} : (a : U) \rightarrow (\text{El} a \rightarrow U) \rightarrow U
\]

\[
\text{El} : U \rightarrow \text{Set} \\
\text{El} (\pi a b) = \Pi x : \text{El} a. \text{El} (b x) \\
\text{El} (\text{sg} a b) = \Sigma x : \text{El} a. \text{El} (b x)
\]

We used $E$ in the type of the constructors $\pi$ and $\text{sg}$ of $U$ before it is defined.

$U$ and $\text{El}$ 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)

\[
\begin{align*}
\text{List}_A & : \text{Set} \\
\text{nil} & : \text{List}_A \\
\text{cons}: (a:A) \rightarrow (e: \text{List}_A) & \rightarrow \text{Fresh e a} \rightarrow \text{List}_A
\end{align*}
\]

\[\text{Fresh}: \text{List}_A \rightarrow A \rightarrow \text{Prop}\]
\[
\begin{align*}
\text{Fresh nil a} & = \text{True} \\
\text{Fresh (cons a_0 e) a} & = a_0 \neq a \land \text{Fresh e a}
\end{align*}
\]

- 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/Seitzer codes can be used.
Some usual type constructions can be realized by 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 \( \text{ZeroOne} : \text{Set} \)

\[
\begin{align*}
\text{Zero} &: \text{ZeroOne} \rightarrow \text{ZeroOne} \\
\text{One} &: \text{ZeroOne} \rightarrow \text{ZeroOne} \\
\text{count1} &: \text{ZeroOne} \rightarrow \mathbb{N} \\
\text{count1} \ (\text{Zero} \ s) &= 0 \\
\text{count1} \ (\text{One} \ s) &= \text{count1} \ s + 1
\end{align*}
\]

\(\text{count1}\) 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

\(S\text{Proc}_{A,B}\) 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 : IN → Set

fZero : (n:IN)(FamZO n) → (FamZO 0)
fOne : (n:IN)(FamZO n) → (FamZO (n+1))

Then we can define a single type by dependent sum

ZeroOne = Σ n:IN. FamZO n

Zero : ZeroOne → ZeroOne

Zero <n,x> = <0, fZero x>

One : Zero One → ZeroOne

One <n,x> = <n+1, fOne x>

count1 : ZeroOne → IN

count1 <n,x> = n

Coq Formalization