To Infinity and Beyond: coinduction[1]

February 10, 2021

[1] Freely inspired from Andreas Abel's "Copatterns Programming Infinite Objects by Observations"

Representation of Infinite Datastructures

Functional encoding (strict languages: Scheme, ML)

- `thunk` computation with an abstraction
  `fun () -> ...`
- `force` computation with an application of `()`

Usual datatype (lazy language: Haskell)

- Laziness allows any datatype to be infinite
- Program with pattern-matching: "Just Be Careful"

Final semantics

- Reacts to (a finite number of) *observations*
- Infinite structure = coinductive datatype

What not to do[2]

```
CoInductive Stream: Type :=
  cons: nat -> Stream -> Stream

zeros: Stream
zeros = cofix (cons 0)

force: Stream -> Stream
forces = case s of cons x y => cons x y

eq: (s: Stream) -> s = force s
eq s = case s of cons x y => refl

eqzeros: zeros = cons 0 zeros
eqzeros = eq zeros
```

[2] Let's see how things unfold, McBride'09

Motto

"Understand coinduction not through construction, but through observations"

Familiar example: functions

- A function is a black box
- Observation: apply it to an argument (*experiment*) and observe its result (*observation*)
- Application is the defining principle of functions
- λ-abstraction is *derived*, secondary to application
- Typical semantic view of functions

Example: infinite objects

Principles

- A coinductive object is a black box
- Finite set of experiments (*projections*) we can perform
- The object is determined by the observations we make

Example

```
class type ['a] stream =
    object
      method hd: 'a
      method tl: 'a stream
    end
```

- `hd`, `tl`: only 2 experiments we can make on a stream
- a `stream` is defined by the results of these experiments

Defining infinite objects

Defining a cofixpoint

```
class ones : [int] stream =
object
  method hd = 1
  method tl = new ones
end
```

Defining the "constructor"

```
class ['a] cons (x: 'a)(s: 'a stream): ['a] stream =
object
  method hd = x
  method tl = s
end
```

Defining infinite objects

```
class interval (i: int) : [int] stream =
object
  method hd = i
  method tl = new interval (i+1)
end

class nat : [int] stream = interval 0

let rec take (n: int)(s: 'a stream): 'a list =
  if n = 0 then
    []
  else
    s#hd :: take (n-1) s#tl
```

Combinators on infinite objects

```
class ['b] map (f: 'a -> 'b)(s: 'a stream):
                                ['b] stream =
object
  method hd = f s#hd
  method tl = new map f s#tl
end

class ['a, 'b] zip (s1: 'a stream)(s2: 'b stream):
                                  ['a * 'b] stream =
object
  method hd = (s1#hd, s2#hd)
  method tl = new zip s1#tl s2#tl
end

class ['a, 'b, 'c] zipWith (op: 'a -> 'b -> 'c)
     (s1: 'a stream)(s2: 'b stream): ['c] stream =
  [' c] map (fun (x, y) -> op x y) (new zip s1 s2)
```

Categorical perspective

Let $\mathcal{F} \in \mathbb{C} \Rightarrow \mathbb{C}$ be an endofunctor.

An $\mathcal{F}$-algebra consists of

- A carrier $A \in |\mathbb{C}|$
- A morphism $\alpha \in \mathbb{C}(|\mathcal{F}|\ A, A)$

The *initial algebra* is an $\mathcal{F}$-coalgebra $(\mu\mathcal{F}, \mathsf{constr} \in \mathbb{C}(|\mathcal{F}|\ (\mu\mathcal{F}), \mu\mathcal{F})$ such that for any $\mathcal{F}$-algebra (A, α) we have

$$\begin{array}{ccc} |\mathcal{F}|\ (\mu\mathcal{F}) & \xrightarrow{\mathcal{F}^{\rightarrow}\ (\mathsf{fix}\ \alpha)} & |\mathcal{F}|\ A \\ \Big\| \mathsf{constr} & & \Big\downarrow \alpha \\ \mu\mathcal{F} & \xrightarrow{\mathsf{fix}\ \alpha} & A \end{array}$$

Reduction: $\mathsf{fix}\ \alpha\ (\mathsf{constr}\ as) = \alpha\ (\mathcal{F}^{\rightarrow}\ (\mathsf{fix}\ \alpha)\ as)$

Categorical perspective

Let $\mathcal{F} \in \mathbb{C} \Rightarrow \mathbb{C}$ be an endofunctor.

An $\mathcal{F}$-coalgebra consists of

- A carrier $A \in |\mathbb{C}|$
- A morphism $\alpha \in \mathbb{C}(A, |\mathcal{F}|\ A)$

The *terminal coalgebra* is an $\mathcal{F}$-coalgebra $(\nu\mathcal{F}, \mathsf{force} \in \mathbb{C}(\nu\mathcal{F}, |\mathcal{F}|\ (\nu\mathcal{F}))$ such that for any $\mathcal{F}$-coalgebra (A, α) we have

$$\begin{array}{ccc} A & \xrightarrow{\quad\mathsf{cofix}\ \alpha\quad} & \nu\mathcal{F} \\ {\scriptstyle\alpha}\downarrow & & \Vert\,{\scriptstyle\mathsf{force}} \\ |\mathcal{F}|\ A & \xrightarrow{\quad\mathcal{F}^{\rightarrow}\ (\mathsf{cofix}\ \alpha)\quad} & |\mathcal{F}|\ (\nu\mathcal{F}) \end{array}$$

Reduction: $\mathsf{force}\ (\mathsf{cofix}\ \alpha\ a) = \mathcal{F}^{\rightarrow}\ (\mathsf{cofix}\ \alpha)\ (\alpha\ a)$

Example: stream

$$\mathcal{F}\, X = \mathbb{N} \times X$$

$$\mathsf{nstream} = \nu\, \mathcal{F}$$

$$\alpha\, () = (1, ())$$

$$\mathsf{ones} = \mathsf{cofix}\, \alpha\, ()$$

$$\begin{array}{ccc} \mathit{unit} & \xrightarrow{\mathsf{cofix}\,\alpha} & \mathsf{nstream} \\ \Big\downarrow{\scriptstyle \alpha} & & \Big\downarrow{\scriptstyle \mathsf{hd,tl}} \\ \mathbb{N} \times \mathit{unit} & \xrightarrow{\mathsf{id} \times (\mathsf{cofix}\,\alpha)} & \mathbb{N} \times \mathsf{nstream} \end{array}$$

Fibonacci

Recurrence equation:

fib	0	1	1	2	3	5	8	...
fib#tl	1	1	2	3	5	8	13	...
zipWith (+)	1	2	3	5	8	13	21	...

```
class fib: [int] stream =
object
  method hd = 0
  method tl =
    object
      method hd = 1
      method tl = new zipWith (+) (new fib) (new fib#tl)
    end
end
```

Conclusion

Positive	Negative
`'a option`	`'a -> 'b`
`'a list`	`'a stream`
`type`	`object`
constructor	methods / copatterns
pattern-matching	projection
algebra	coalgebra