Towards Object Orientation in Agda
Part I: Coalgebras and IO

Stephan Adelsberger
Vienna University of Economics and Business

Anton Setzer
Swansea University, Swansea UK

TCS Seminar, Department of Computer Science, Swansea University

10 April 2017

Coalgebras in Agda

Interactive Programs in Agda

State-Dependent IO

Coalgebras in Agda

Interactive Programs in Agda

State-Dependent IO

Codata Type

- Idea of Codata Types:

$$\begin{array}{l} \textsf{codata Stream} : \textsf{Set where} \\ \quad \textsf{cons} \quad : \quad \mathbb{N} \to \textsf{Stream} \to \textsf{Stream} \end{array}$$

- Same definition as inductive data type but we are allowed to have infinite chains of constructors

$$\textsf{cons}\ n_0\ (\textsf{cons}\ n_1\ (\textsf{cons}\ n_2\ \cdots))$$

- **Problem 1:** Non-normalisation.
- **Problem 2:** Equality between streams is equality between all elements, and therefore undecidable.
- **Problem 3:** Underlying assumption is

$$\forall s : \textsf{Stream}. \exists n, s'. s = \textsf{cons}\ n\ s'$$

which results in undecidable equality.

Solution: Coalgebras Defined by Observations

- We define coalgebras by their observations. Tentative syntax

$$\begin{array}{l} \text{coalg Stream} : \text{Set where} \\ \quad \begin{array}{lcl} \text{head} & : & \text{Stream} \to \mathbb{N} \\ \text{tail} & : & \text{Stream} \to \text{Stream} \end{array} \end{array}$$

- Stream is the largest set of terms which allow arbitrary many applications of tail followed by head to obtain a natural numbers.
- From this one can develop a general model for coalgebras (see our paper [Set16]).
- Therefore no infinite expansion of streams:
  – for each expansion of a stream one needs one application of tail.

Syntax in Agda

- In Agda the record type has been reused for defining coalgebras:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

const and inc can be defined with the syntax as given before

Principle of Guarded Recursion

- Define

$$\begin{array}{llllll} f : A \to \mathrm{Stream} \\ \mathrm{head} & (f\ \ a) & = & \cdots & : & \mathbb{N} \\ \mathrm{tail} & (f\ \ a) & = & \cdots & : & \mathrm{Stream} \end{array}$$

  where

$$\begin{array}{llllll} \mathrm{tail}\,(f\ a) & = & f\ a' & \text{for some} & a' & : & A \\ \text{or} \\ \mathrm{tail}\,(f\ a) & = & s' & \text{for some} & s' & : & \mathrm{Stream}\ \text{given before} \end{array}$$

- **No** function can be applied to the corecursion hypothesis.
- Using sized types one can apply size preserving or size increasing functions to co-IH (Abel).
- Above is example of **copattern matching**.

Example

- Constant stream of $a, a, a, \ldots$

$$\begin{array}{lcl} \text{const} : \{A : \text{Set}\} \to A \to \text{Stream}\ \ A \\ \text{head}\ (\text{const}\ a) & = & a \\ \text{tail}\ \ (\text{const}\ a) & = & \text{const}\ \ a \end{array}$$

- The increasing stream $n, n+1, n+2, \ldots$

$$\begin{array}{lcl} \text{inc} : \mathbb{N} \to \text{Stream}\ \ \mathbb{N} \\ \text{head}\ (\text{inc}\ n) & = & n \\ \text{tail}\ \ (\text{inc}\ n) & = & \text{inc}\ (n+1) \end{array}$$

- Cons is **defined**:

$$\begin{array}{lcl} \text{cons}\ \ : X \to \text{Stream}\ \ X \to \text{Stream}\ \ X \\ \text{head}\ \ (\text{cons}\ \ x\ l) & = & x \\ \text{tail}\ \ (\text{cons}\ \ x\ l) & = & l \end{array}$$

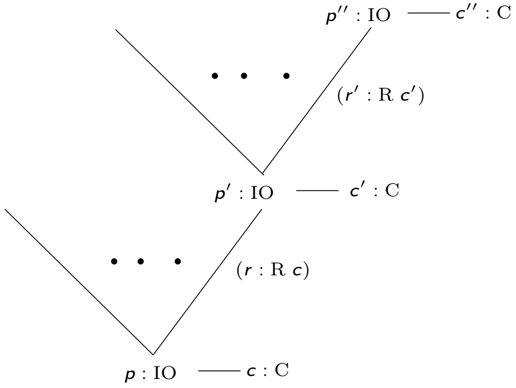
Nested Patter/Copattern Matching

- We can even define functions by a combination of pattern and copattern matching and nest those:
  The following defines the stream

$$\mathsf{stutterDown}\ \ n\ n \ \ = \ \ n, n, n-1, n-1, \ldots 0, 0, n, n, n-1, n-1, \ldots$$

$$
\begin{array}{ll}
\mathsf{stutterDown} : \mathbb{N} \to \mathbb{N} \to \mathsf{Stream}\ \mathbb{N} & \\
\mathsf{head}\ (\mathsf{stutterDown}\ n\ m) & = m \\
\mathsf{head}\ (\mathsf{tail}\ (\mathsf{stutterDown}\ n\ m)) & = m \\
\mathsf{tail}\ (\mathsf{tail}\ (\mathsf{stutterDown}\ n\ (\mathsf{suc}\ m))) & = \mathsf{stutterDown}\ n\ m \\
\mathsf{tail}\ (\mathsf{tail}\ (\mathsf{stutterDown}\ n\ 0)) & = \mathsf{stutterDown}\ n\ n
\end{array}
$$

IO-Trees (Non-State Dependent)



IOInterface

An IOInterface is a record having fields Command and Response:

```
record IOInterface : Set₁ where
  field Command  : Set
        Response : Command → Set
```

Console Interface

```
data ConsoleCommand : Set where
  getLine   : ConsoleCommand
  putStrLn : String → ConsoleCommand

ConsoleResponse : ConsoleCommand → Set
ConsoleResponse getLine        = String
ConsoleResponse (putStrLn s) = Unit

ConsoleInterface : IOInterface
Command ConsoleInterface = ConsoleCommand
Response ConsoleInterface  = ConsoleResponse
```

IO

The set of IO programs IO∞ is the coalgebra having as observation an element of IO.
Elements of IO are IO trees which can have leaves (introduced by return) and nodes (introduced by do):

```
mutual
  record IO∞ (I : IOInterface) (A : Set) : Set where
    coinductive
    field force : IO I A

  data IO (I : IOInterface) (A : Set) : Set where
    do : (c : Command I) (f : Response I c → IO∞ I A)
                           → IO I A
    return : A → IO I A
```

Monadic bind is used to combine programs:

$$
\begin{array}{l}
\mathsf{mutual} \\
\quad \_\gg=\_ : \forall\{A\ B\}\ (m : \mathsf{IO}\ I\ A)\ (k : A \to \mathsf{IO}\infty\ I\ B) \to \mathsf{IO}\ I\ B \\
\quad \mathsf{do}\ c\ f \qquad \gg= k = \mathsf{do}\ c\ \lambda\ x \to f\ x \gg=\infty\ k \\
\quad \mathsf{return}\ a \qquad \gg= k = \mathsf{force}\ (k\ a) \\
\\
\quad \_\gg=\infty\_ : \forall\{A\ B\}\ (m : \mathsf{IO}\infty\ I\ A)\ (k : A \to \mathsf{IO}\infty\ I\ B) \\
\qquad\qquad\qquad \to \mathsf{IO}\infty\ I\ B \\
\quad \mathsf{force}\ (m \gg=\infty\ k) = \mathsf{force}\ m \gg= k
\end{array}
$$

Running Interactive Programs

```
{-# NON_TERMINATING #-}
translateIO : ∀ {A} (tr : (c : C) → NativeIO (R c)) → IO∞ I A
                → NativeIO A
translateIO tr m = case (force m) of λ
  { (do c f)    → (tr c) native>>= λ r → translateIO tr ( f r)
  ; (return a) → nativeReturn a
  }
```

Non termination is unproblematic since this function is only used as part of the compilation process.

Console IO

$$
\begin{array}{l}
\mathrm{IOConsole} : \mathrm{Set} \to \mathrm{Set} \\
\mathrm{IOConsole} = \mathrm{IO}\infty\ \mathrm{ConsoleInterface}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{translateIOConsoleLocal} : & (c : \mathrm{ConsoleCommand}) \\
 & \to \mathrm{NativeIO}\ (\mathrm{ConsoleResponse}\ c) \\
\mathrm{translateIOConsoleLocal}\ (\mathrm{putStrLn}\ s) & = \mathrm{nativePutStrLn}\ s \\
\mathrm{translateIOConsoleLocal}\ \mathrm{getLine} & = \mathrm{nativeGetLine}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{translateIOConsole} : \{A : \mathrm{Set}\} \to \mathrm{IOConsole}\ A \to \mathrm{NativeIO}\ A \\
\mathrm{translateIOConsole} = \mathrm{translateIO}\ \mathrm{translateIOConsoleLocal}
\end{array}
$$

A First Interactive Program

```
cat : IOConsole Unit
force cat = do getLine λ line →
            do∞ (putStrLn line) λ _ →
            cat
```

- This program doesn't termination check because in guarded recursion we are not allowed to apply the defined function do∞o to the corecursive call of cat.
- Can be repaired using sized Types (Abel).
  - Using sized types one can apply size preserving or increasing functions to corecursive calls.
  - The code in the following usually requires decorations by sized types in order to termination check.

Executable Program

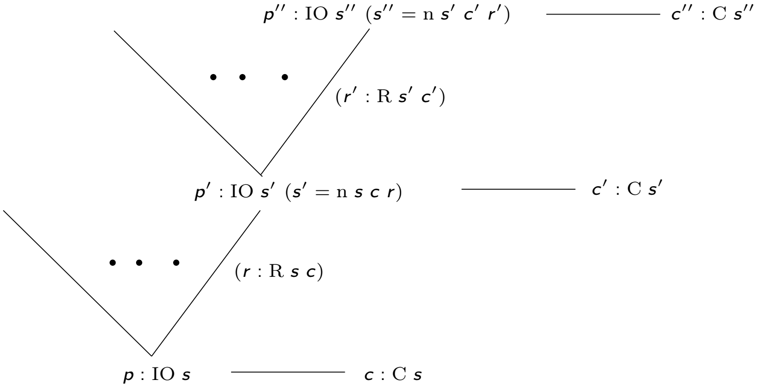
main : NativeIO Unit
main = translateIOConsole cat

Coalgebras in Agda

Interactive Programs in Agda

State-Dependent IO

State Dependent IO-Trees



State Dependent IO – Interface

```
record IOInterfaceˢ : Set₂ where
  field
    Stateˢ    : Set₁
    Commandˢ  : Stateˢ → Set₁
    Responseˢ : (s : Stateˢ) → Commandˢ s → Set
    nextˢ     : (s : Stateˢ) → (c : Commandˢ s)
                → Responseˢ s c
                → Stateˢ
```

State Dependent IO

```
record IOˢ (A : S → Set) (s : S) : Set₁ where
    coinductive
    field
      forceˢ : IOˢ' A s

data IOˢ' (A : S → Set) : S → Set₁ where
   doˢ' :      {s : S} → (c : C s)
           → ( f : (r : R s c) → IOˢ A (next s c r) )
           → IOˢ' A s
   returnˢ' : {s : S} → (a : A s) → IOˢ' A s
```

Bibliography I

Andreas Abel, Stephan Adelsberger, and Anton Setzer.
ooAgda.
Agda Library. Available from `https://github.com/agda/ooAgda`, 2016.

Andreas Abel, Stephan Adelsberger, and Anton Setzer.
Interactive programming in Agda – objects and graphical user interfaces.
Journal of Functional Programming, 27, Jan 2017.

Anton Setzer.
Object-oriented programming in dependent type theory.
In Conference Proceedings of TFP 2006, 2006.
Available from http://www.cs.nott.ac.uk/∼nhn/TFP2006/TFP2006-Programme.html and http://www.cs.swan.ac.uk/∼csetzer/index.html.

Bibliography II

Anto Setzer.
How to reason coinductively informally.
In Reinhard Kahle, Thomas Strahm, and Thomas Studer, editors, Advances in Proof Theory, pages 377–408. Springer, 2016.