GUIs, Object Based Programming, and Processes in Agda

Anton Setzer
Swansea University, Swansea UK
(Joint work with Andreas Abel, Stephan Adelsberger, and Bashar Igried)
COST Action EUTYPES WG meeting, Lisbon, Portugal

5 October 2016

Codata Type

- Idea of Codata Types:

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

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

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

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

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

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

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

where

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

- **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 $a, a, a, \ldots$

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

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

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

- Cons is **defined**:

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

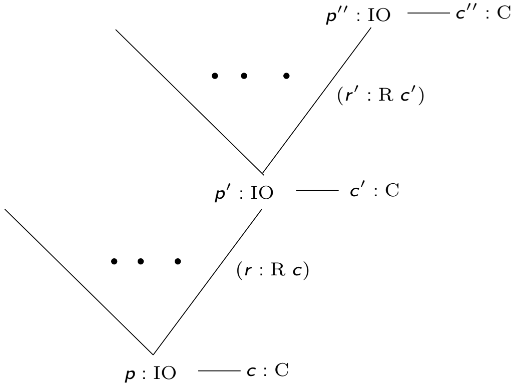
Nested Patter/Copattern Matching

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

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

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

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:

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

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

$$\mathsf{IOConsole} : \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{IOConsole} = \mathsf{IO}\infty\ \mathsf{ConsoleInterface}$$

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

$$\mathsf{translateIOConsole} : \{A : \mathsf{Set}\} \to \mathsf{IOConsole}\ A \to \mathsf{NativeIO}\ A$$
$$\mathsf{translateIOConsole} = \mathsf{translateIO}\ \mathsf{translateIOConsoleLocal}$$

A First Interactive Program

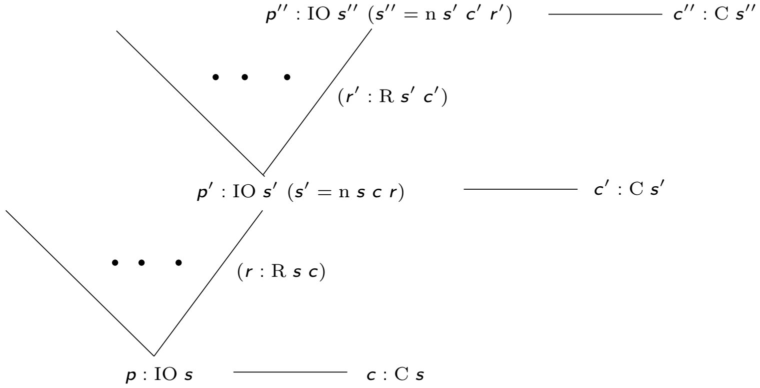
```
cat   :  IOConsole Unit
force cat =  do getLine λ line →
             do∞ (putStrLn line)  λ _ →
             cat
```

- 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

$$
\begin{array}{l}
\mathrm{main} : \mathrm{NativeIO}\ \mathrm{Unit} \\
\mathrm{main} = \mathrm{translateIOConsole}\ \mathrm{cat}
\end{array}
$$

State Dependent IO-Trees



State Dependent IO – Interface

$$
\begin{array}{lll}
\textsf{record}\ \mathsf{IOInterface}^{\mathrm{s}} & : & \mathsf{Set}_2\ \textsf{where} \\
\quad \textsf{field} & & \\
\qquad \mathsf{State}^{\mathrm{s}} & : & \mathsf{Set}_1 \\
\qquad \mathsf{Command}^{\mathrm{s}} & : & \mathsf{State}^{\mathrm{s}} \to \mathsf{Set}_1 \\
\qquad \mathsf{Response}^{\mathrm{s}} & : & (s : \mathsf{State}^{\mathrm{s}}) \to \mathsf{Command}^{\mathrm{s}}\ s \to \mathsf{Set} \\
\qquad \mathsf{next}^{\mathrm{s}} & : & (s : \mathsf{State}^{\mathrm{s}}) \to (c : \mathsf{Command}^{\mathrm{s}}\ s) \\
 & & \to \mathsf{Response}^{\mathrm{s}}\ s\ c \\
 & & \to \mathsf{State}^{\mathrm{s}}
\end{array}
$$

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
```

Object-Oriented/Based Programming

- Object-oriented (OO) programming is currently main programming paradigm.
- Good for bundling operations into one objects, hiding implementations and reuse of code.
- Here restriction to **object-based programming**.
  - Only notion of an object covered.
- Ultimate goal: use objects in order to organise proofs in a better way.

Example: cell in Java

```
class cell <A> {

    /* Instance Variable */
    A content;

    /* Constructor */
    cell (A s) { content = s; }

    /* Method put */
    public void put (A s) { content = s; }

    /* Method get */
    public A get () { return content; }
}
```

Modelling Methods as Objects

- The Type (interface) cell modelled as a coalgebra Cell.
- A method

$$B \ \mathrm{m} \ (A \ x)$$

  is modelled as observation

$$\mathsf{m} : \mathsf{Cell} \to \mathsf{A} \to \mathsf{B} \times \mathsf{Cell}$$

- Return type void is modelled as Unit (one element type).
- A constructor with argument A modelled as a function defined by guarded recursion

$$\mathsf{cell} : \mathsf{A} \to \mathsf{Cell}$$

Cell in Agda

```
record Cell (X : Set) : Set where
  coinductive
  field
    put : X    → Unit × Cell X
    get : Unit → X    × Cell X

cell : {X : Set} → X → Cell X
put  (cell x) y = (unit , cell y)
get  (cell x) _ = (x    , cell x)
```

Generic Version

An interface for an object consist of methods and the result type:

```
record Interface   :  Set₁ where
   field  Method   :  Set
          Result   :  Method → Set
```

An Object of an interface I has a method which for every method returns an element of the result type and the updated object:

```
record Object (I : Interface) : Set where
   coinductive
   field objectMethod : (m : Method I) → Result I m × Object I
```

Example: A Cell

A cell contains one element.
The methods allow to get its content and put a new value into the cell:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{CellMethod}\ A : \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{get} \ :\ \mathsf{CellMethod}\ A \\
\quad \mathsf{put} \ :\ A \to \mathsf{CellMethod}\ A
\end{array}
$$

$$
\begin{array}{llcl}
\mathsf{CellResult} & & : & \forall\{A\} \to \mathsf{CellMethod}\ A \to \mathsf{Set} \\
\mathsf{CellResult} & \{A\}\ \mathsf{get} & = & A \\
\mathsf{CellResult} & (\mathsf{put}\ \_) & = & \mathsf{Unit}
\end{array}
$$

$$
\begin{array}{llcl}
\mathsf{cellI} & & : & (A : \mathsf{Set}) \to \mathsf{Interface} \\
\mathsf{Method} & (\mathsf{cellI}\ A) & = & \mathsf{CellMethod}\ A \\
\mathsf{Result} & (\mathsf{cellI}\ A)\ m & = & \mathsf{CellResult}\ m
\end{array}
$$

Definition of Cell

The cell object is defined as follows:

```
Cell : Set → Set
Cell A = Object (cellI A)

cell : {A : Set} → A → Cell A
objectMethod (cell a) get      = ( a     , cell a )
objectMethod (cell a) (put b)  = ( unit  , cell b )
```

IO objects

IO Objects are like Objects, but methods execute an interactive program before returning the result:

$$
\begin{array}{l}
\mathsf{record}\ \mathsf{IOObject}\ (I_{\text{io}} : \mathsf{IOInterface})\ (I : \mathsf{Interface}) : \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{coinductive} \\
\quad \mathsf{field}\ \mathsf{method} : (m : \mathsf{Method}\ I) \\
\qquad \rightarrow \mathsf{IO}\infty\ I_{\text{io}}\ (\mathsf{Result}\ I\ m \times \mathsf{IOObject}\ I_{\text{io}}\ I)
\end{array}
$$

IOCell

We define an IOcell which writes on console a trace of its method calls:

```
IOCell : Set
IOCell = IOObject ConsoleInterface (cellI String)


ioCell : (s : String) → IOCell
force (method (ioCell s)  get) =
   do (putStrLn ("getting (" ++ s ++ ")")) λ _ →
   return∞ (s , ioCell s)
force (method (ioCell _) (put t)) =
   do (putStrLn ("putting (" ++ t ++ ")")) λ _ →
   return∞ (_ , ioCell t)
```

Example Program using IOCell

```
program : IOCell  → IO∞ ConsoleInterface Unit
force (program c) =
   do getLine           λ s →
   method c (put s)  ≫=∞ λ{ (_ , c) →
   method c get       ≫=∞ λ{ (t , c) →
   do∞ (putStrLn t )  λ _ →
   program c }}


main : NativeIO Unit
main = translateIOConsole (program (ioCell "Start"))
```

State Dependent Interface

```
record Interfaceˢ : Set₁ where
  field
    Stateˢ   : Set
    Methodˢ  : Stateˢ → Set
    Resultˢ  : (s : Stateˢ) → (m : Methodˢ s) → Set
    nextˢ    : (s : Stateˢ) → (m : Methodˢ s) → Resultˢ s m
               → Stateˢ
```

State Dependent Object

Assuming I : Interface$^{\mathrm{s}}$ we define the set of state dependent objects:

$$
\begin{array}{l}
\text{record Object}^{\mathrm{s}}\ (I : \text{Interface}^{\mathrm{s}})\ (s : \text{State}^{\mathrm{s}}\ I) : \text{Set where} \\
\text{coinductive} \\
\text{field} \\
\quad \text{objectMethod} : \ (m : \text{Method}^{\mathrm{s}}\ I\ s) \\
\qquad\qquad\qquad\qquad \to \Sigma[\ r \in \text{Result}^{\mathrm{s}}\ I\ s\ m\ ]\ \text{Object}^{\mathrm{s}}\ I\ (\text{next}^{\mathrm{s}}\ I\ s\ m\ r)
\end{array}
$$

Example Safe Stack

$$\mathrm{StackState}^{\mathrm{s}} = \mathbb{N}$$

$$\begin{array}{l}
\mathrm{data}\ \mathrm{StackMethod}^{\mathrm{s}}\ (A : \mathrm{Set}) : \mathrm{StackState}^{\mathrm{s}} \to \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{push} : \{n : \mathrm{StackState}^{\mathrm{s}}\} \to A \to \mathrm{StackMethod}^{\mathrm{s}}\ A\ n \\
\quad \mathrm{pop}\ \ : \{n : \mathrm{StackState}^{\mathrm{s}}\} \to \mathrm{StackMethod}^{\mathrm{s}}\ A\ (\mathrm{suc}\ n)
\end{array}$$

$$\begin{array}{l}
\mathrm{StackResult}^{\mathrm{s}} : \ (A : \mathrm{Set}) \to (s : \mathrm{StackState}^{\mathrm{s}}) \to \mathrm{StackMethod}^{\mathrm{s}}\ A\ s \\
\qquad\qquad\qquad \to \mathrm{Set} \\
\mathrm{StackResult}^{\mathrm{s}}\ A\ .n\ (\mathrm{push}\ \{\ n\ \}\ x_1)\ = \mathrm{Unit} \\
\mathrm{StackResult}^{\mathrm{s}}\ A\ (\mathrm{suc}\ .n)\ \ (\mathrm{pop}\ \{n\}\ ) = A
\end{array}$$

$$\begin{array}{l}
\mathrm{n}^{\mathrm{s}} : \ (A : \mathrm{Set}) \to (s : \mathrm{StackState}^{\mathrm{s}}) \to (m : \mathrm{StackMethod}^{\mathrm{s}}\ A\ s) \\
\qquad \to (r : \mathrm{StackResult}^{\mathrm{s}}\ A\ s\ m) \to \mathrm{StackState}^{\mathrm{s}} \\
\mathrm{n}^{\mathrm{s}}\ A\ .n\ (\mathrm{push}\ \{\ n\ \}\ x)\ \ r = \mathrm{suc}\ n \\
\mathrm{n}^{\mathrm{s}}\ A\ (\mathrm{suc}\ .n)\ (\mathrm{pop}\ \ \{\ n\ \})\ r = n
\end{array}$$

Safe Stack

$$
\begin{array}{llll}
\mathsf{StackInterface}^{\mathrm{s}} : (A : \mathsf{Set}) \to \mathsf{Interface}^{\mathrm{s}} & & & \\
\mathsf{State}^{\mathrm{s}} & (\mathsf{StackInterface}^{\mathrm{s}}\ \ A) & = & \mathsf{StackState}^{\mathrm{s}} \\
\mathsf{Method}^{\mathrm{s}} & (\mathsf{StackInterface}^{\mathrm{s}}\ \ A) & = & \mathsf{StackMethod}^{\mathrm{s}}\ A \\
\mathsf{Result}^{\mathrm{s}} & (\mathsf{StackInterface}^{\mathrm{s}}\ \ A) & = & \mathsf{StackResult}^{\mathrm{s}}\ A \\
\mathsf{next}^{\mathrm{s}} & (\mathsf{StackInterface}^{\mathrm{s}}\ \ A) & = & \mathsf{n}^{\mathrm{s}}\ A
\end{array}
$$

$$
\begin{array}{l}
\mathsf{stackO} : \ \forall\{E : \mathsf{Set}\}\ \{n : \mathbb{N}\}\ (v : \ \mathsf{Vec}\ E\ n) \\
\qquad\quad \to \mathsf{Object}^{\mathrm{s}}\ (\mathsf{StackInterface}^{\mathrm{s}}\ E)\ n
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{objectMethod} & (\mathsf{stackO}\ es) & (\mathsf{push}\ e) & = (\_ \ , & \mathsf{stackO}\ \ (e :: es)) \\
\mathsf{objectMethod} & (\mathsf{stackO}\ (e :: es)) & \mathsf{pop} & = (e \ , & \mathsf{stackO}\ \ es)
\end{array}
$$

Example Fibonacci Stack

```
data FibState : Set where
  fib : ℕ → FibState
  val : ℕ → FibState


data FibStackEl : Set where
  _+·    : ℕ → FibStackEl
  ·+fib_ : ℕ → FibStackEl

FibStack : ℕ → Set
FibStack = Objectˢ (StackInterfaceˢ FibStackEl)

emptyFibStack : FibStack 0
emptyFibStack = stackO []
```

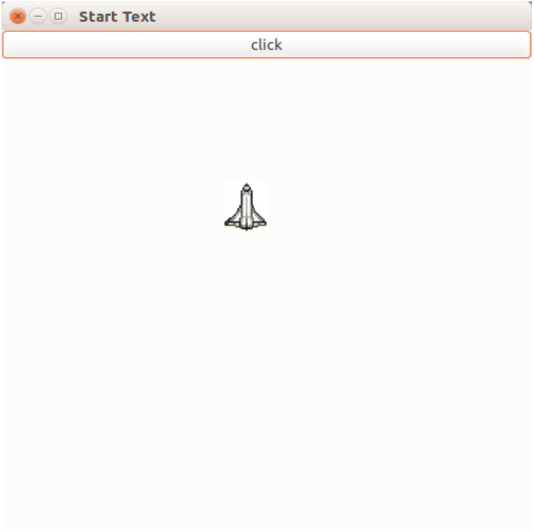
Reduce

```
reduce : Stackmachine → Stackmachine ⊎ ℕ
reduce (n , fib 0 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib 1 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib (suc (suc m)) , stack) =
   objectMethod stack (push (·+fib m)) ▷ λ { (_ , stack₁) →
   inj₁ ( suc n , fib (suc m) , stack₁)      }
reduce (0 , val m , stack) = inj₂ m
reduce (suc n , val m , stack) =
   objectMethod stack pop ▷           λ {  (k +· , stack₁) →
   inj₁ (n , val (k + m) , stack₁) ;
                                          (·+fib k , stack₁) →
   objectMethod stack₁ (push (m +·)) ▷ λ {(_ , stack₂) →
   inj₁ (suc n , fib k , stack₂) }  }
```

Fibonacci Function

```
{-# NON_TERMINATING #-}
iter : Stackmachine → ℕ
iter stack  with reduce stack
... | inj₁ s'  = iter s'
... | inj₂ m  = m

fibUsingStack : ℕ → ℕ
fibUsingStack n = iter (0 , fib n , emptyFibStack)
```

SpaceShip Example



Graphics Interface Level1

```
data GuiLev1Command : Set where
  makeFrame  : GuiLev1Command
  makeButton : Frame → GuiLev1Command
  addButton  : Frame → Button → GuiLev1Command
  drawBitmap : DC    → Bitmap → Point → Bool
               → GuiLev1Command
  repaint    : Frame → GuiLev1Command

GuiLev1Response : GuiLev1Command → Set
GuiLev1Response makeFrame      = Frame
GuiLev1Response (makeButton _) = Button
GuiLev1Response _              = Unit

GuiLev1Interface : IOInterface
Command  GuiLev1Interface = GuiLev1Command
Response GuiLev1Interface = GuiLev1Response
```

Graphics Level2 Commands

$$
\begin{aligned}
&\text{GuiLev2State} : \text{Set}_1\\
&\text{GuiLev2State} = \text{VarList}
\end{aligned}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \text{GuiLev2Command}\ (s : \ \text{GuiLev2State}) : \text{Set}_1\ \textbf{where}}\\
\quad \text{level1C} & : & \text{GuiLev1Command} \rightarrow \text{GuiLev2Command}\ s\\
\quad \text{createVar} & : & \{A : \text{Set}\} \rightarrow A \rightarrow \text{GuiLev2Command}\ s\\
\quad \text{setButtonHandler} & : & \text{Button}\\
 & & \rightarrow \text{List}\ \ (\text{prod}\ s\\
 & & \quad \rightarrow \text{IO GuiLev1Interface}\ \infty\ (\text{prod}\ s))\\
 & & \rightarrow \text{GuiLev2Command}\ s\\
\quad \text{setOnPaint} & : & \text{Frame}\\
 & & \rightarrow \text{List}\ \ (\text{prod}\ s \rightarrow \text{DC} \rightarrow \text{Rect}\\
 & & \quad \rightarrow \text{IO GuiLev1Interface}\ \infty\ (\text{prod}\ s))\\
 & & \rightarrow \text{GuiLev2Command}\ s
\end{array}
$$

Graphics Level2 Response + Next

$$
\begin{array}{lll}
\mathsf{GuiLev2Response} : & (s : \mathsf{GuiLev2State}) \to \mathsf{GuiLev2Command}\ s & \\
 & \to \mathsf{Set} & \\
\mathsf{GuiLev2Response}\ \_\ (\mathsf{level1C}\ c) & = & \mathsf{GuiLev1Response}\ c \\
\mathsf{GuiLev2Response}\ \_\ (\mathsf{createVar}\ \{A\}\ a) & = & \mathsf{Var}\ A \\
\mathsf{GuiLev2Response}\ \_\ \_ & = & \mathsf{Unit}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{GuiLev2Next} : (s : \mathsf{GuiLev2State}) \to (c : \mathsf{GuiLev2Command}\ s)} \\
\multicolumn{3}{l}{\quad \to \mathsf{GuiLev2Response}\ s\ c} \\
\multicolumn{3}{l}{\quad \to \mathsf{GuiLev2State}} \\
\mathsf{GuiLev2Next}\ s\ (\mathsf{createVar}\ \{A\}\ a)\ \ var & = & \mathsf{addVar}\ A\ var\ s \\
\mathsf{GuiLev2Next}\ s\ \_\ \_ & = & s
\end{array}
$$

Graphics Level2 Interface

$$\begin{array}{lrcl}
\multicolumn{4}{l}{\mathsf{GuiLev2Interface} : \mathsf{IOInterface}^{\mathrm{s}}} \\
\mathsf{State}^{\mathrm{s}} & \mathsf{GuiLev2Interface} & = & \mathsf{GuiLev2State} \\
\mathsf{Command}^{\mathrm{s}} & \mathsf{GuiLev2Interface} & = & \mathsf{GuiLev2Command} \\
\mathsf{Response}^{\mathrm{s}} & \mathsf{GuiLev2Interface} & = & \mathsf{GuiLev2Response} \\
\mathsf{next}^{\mathrm{s}} & \mathsf{GuiLev2Interface} & = & \mathsf{GuiLev2Next}
\end{array}$$

Action Handling Object

```
data ActionHandlerMethod : Set where
   onPaintM        : DC    → Rect → ActionHandlerMethod
   moveSpaceShipM  : Frame → ActionHandlerMethod
   callRepaintM    : Frame → ActionHandlerMethod

ActionHandlerResult : ActionHandlerMethod → Set
ActionHandlerResult _ = Unit

ActionHandlerInterface : Interface
Method ActionHandlerInterface = ActionHandlerMethod
Result ActionHandlerInterface = ActionHandlerResult

ActionHandler : Set
ActionHandler = IOObject GuiLev1Interface ActionHandlerInterface
```

Action Handling Object

```
actionHandler : ℤ → ActionHandler
method (actionHandler z) (onPaintM dc rect) =
   do∞ (drawBitmap dc ship (z , (+ 150)) true) λ _ →
   return∞ (unit , actionHandler z)
method (actionHandler z) (moveSpaceShipM fra) =
   return∞ (unit , actionHandler (z + (+ 20)))
method (actionHandler z) (callRepaintM fra) =
   do∞ (repaint fra) λ _ →
   return∞ (unit , actionHandler z)

actionHandlerInit : ActionHandler
actionHandlerInit = actionHandler (+ 150)
```

Action Handlers

```
onPaint : ActionHandler → DC → Rect
            → IO GuiLev1Interface ActionHandler
onPaint obj dc rect = mapIO proj₂ (method obj (onPaintM dc rect))

moveSpaceShip : Frame → ActionHandler
   → IO GuiLev1Interface ActionHandler
moveSpaceShip fra obj = mapIO proj₂
     (method obj (moveSpaceShipM fra))
```

Action Handlers

$$\begin{aligned}\mathrm{callRepaint}\ &:\ \mathrm{Frame} \to \mathrm{ActionHandler}\\ &\quad \to \mathrm{IO}\ \mathrm{GuiLev1Interface}\ \mathrm{ActionHandler}\end{aligned}$$

$$\mathrm{callRepaint}\ \mathit{fra}\ \mathit{obj} = \mathrm{mapIO}\ \ \mathrm{proj}_2\ (\mathrm{method}\ \mathit{obj}\ (\mathrm{callRepaintM}\ \mathit{fra}))$$

$$\begin{aligned}&\mathrm{buttonHandler} : \mathrm{Frame} \to \mathrm{List}\ (\mathrm{ActionHandler}\\ &\quad \to \mathrm{IO}\ \mathrm{GuiLev1Interface}\ \mathrm{ActionHandler})\\ &\mathrm{buttonHandler}\ \mathit{fra} = \mathrm{moveSpaceShip}\ \mathit{fra} :: [\ \mathrm{callRepaint}\ \mathit{fra}\ ]\end{aligned}$$

Spaceship Program

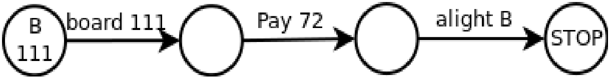
```
program : IOˢ GuiLev2Interface (λ _ → Unit) []
program = doˢ (level1C makeFrame)          λ fra →
          doˢ (level1C (makeButton fra))   λ bt →
          doˢ (level1C (addButton fra bt)) λ _ →
          doˢ (createVar actionHandlerInit) λ _ →
          doˢ (setButtonHandler bt  (moveSpaceShip fra
                                     :: [ callRepaint fra ])) λ _ →
          doˢ (setOnPaint fra [ onPaint  ])
          returnˢ

main : NativeIO Unit
main = start (translateLev2 program)
```

Process Algebras

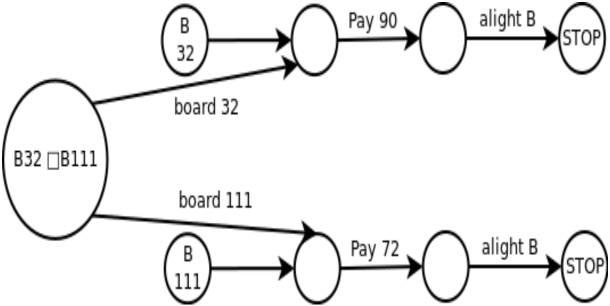
- Goal of process algebras is to model **concurrent systems**.
- They define labelled transition systems with
  - nodes being processes
  - transitions correspond to non-deterministic ways in which a process can proceed,
  - labels control interaction and information flow between different processes.
- Operators have been defined to form new processes from existing ones.
- Various semantics are defined for process algebras.
- Equations are given for relating processes formed from different operators which are semantically equal.

Example (Bus)

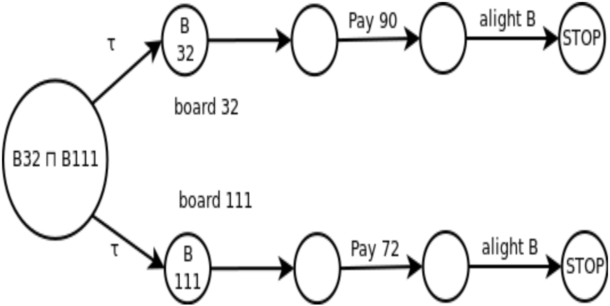
Process B111



External Choice



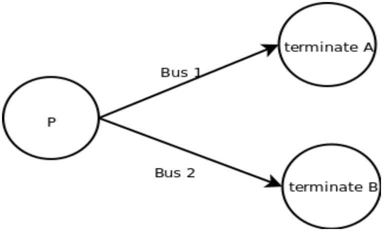
Internal Choice



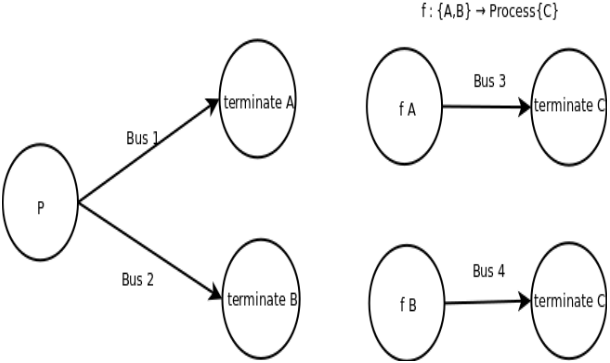
Monadic Composition of Processes

- In order to obtain monadic composition of processes
  - we add a special **terminated process**
  - terminated process have in addition a **return value**,
  - we can combine processes monadically where **next process depends on return value** of terminated process.

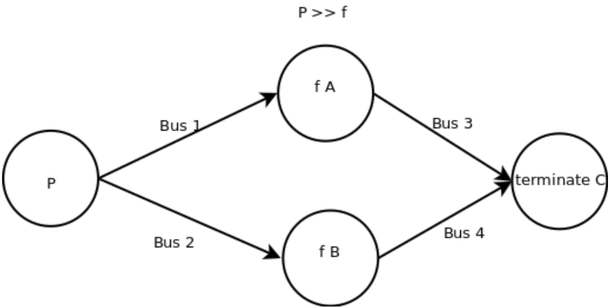
Example Monadic Composition



Example Monadic Composition



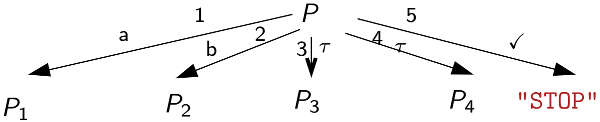
Example Monadic Composition



Definition of Processes Based on Atomic Operation

- In process algebra processes are formed using high level operations (external/internal choice, parallel, composition etc.)
- Instead we from processes from **atomic one step operations**.
- Since processes can loop forever defined **coinductively**.
- In CSP processes can have **3 kinds of transitions**:
  - Labelled **external choice** transitions.
  - **Internal** τ**-**transitions.
  - **Termination events** (✓-transitions).
    - In CSP instead of having terminated processes there are **termination events**.
    - A natural way would be to represent them as τ-transitions to terminated processes
      – however, CSP behave slightly differently.
    - In order to be consistent with CSP we keep termination events.
    - We add return values to termination events in order to allow monadic composition.

Example



3 Levels of Processes

- Process+ will be processes which are not the terminated process.
  - They can have external choice, internal choice and termination events.
  - Defined coinductively as a record.
- Process is the type of processes which can be either the terminated process or continue as an element of Process+
  - Defined as a data.
- Coinductive definitions of elements of Process+ require to define 8 components.
  Sometimes we want to define a process coinductively by using other combinators
  Therefore we have a third kind of process Process∞ (coinductive)
  Essentially bundles the 8 components into one for corecursive definitions.

CSP-Agda

```
record Process+ (i : Size) (c : Choice) : Set where
  coinductive
  field
    E    : Choice
    Lab  : ChoiceSet E → Label
    PE   : ChoiceSet E → Process∞ i c
    I    : Choice
    PI   : ChoiceSet I → Process∞ i c
    T    : Choice
    PT   : ChoiceSet T → ChoiceSet c
    Str+ : String
```

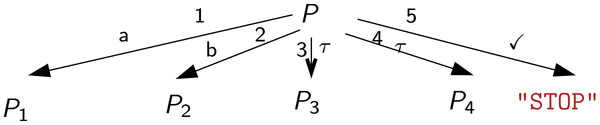
CSP-Agda

```
data Process (i : Size) (c : Choice) : Set where
  terminate : ChoiceSet  c   → Process i c
  node      : Process+   i c → Process i c

record Process∞ (i : Size) (c : Choice) : Set where
  coinductive
  field
    forcep : {j : Size< i} → Process j c
    Str∞   : String
```

Example

$$
\begin{array}{ll}
P = & \text{node}\ (\text{process+}\ E\ Lab\ PE\ I\ PI\ T\ PT\ \texttt{"P"}) \\
 & : \text{Process String} \quad \text{where} \\
E = & \text{code for } \{1, 2\} \qquad\qquad I = \ \text{code for } \{3, 4\} \\
T = & \text{code for } \{5\}
\end{array}
$$

$$
\begin{array}{lllllllll}
Lab\,1 & = & a & Lab\,2 & = & b & PE\,1 & = & P_1 \\
PE\,2 & = & P_2 & PI\,3 & = & P_3 & PI\,4 & = & P_4 \\
PT\,5 & = & \texttt{"STOP"} & & & & & &
\end{array}
$$



Choices Set

- In order to develop a simulator we need
  - to **enumerate all** possible **choices**
    (therefore need to make sure choice sets are finite)
  - **print** them out as a string.
- Elements of the result sets need to be printed out as well.
- Therefore we model both as elements of a **universe** rather than as sets.
- Universes go back to Martin-Löf in order to formulate the notion of a type consisting of types.
- Universes are defined in Agda by an **inductive-recursive** definition.

Choice Sets

We give here the code expressing that Choice is closed under fin, ⊎ and subset'.

```
mutual
data Choice : Set where
   fin     : ℕ → Choice
   _⊎'_    : Choice → Choice → Choice
   subset' : (E : Choice) → (ChoiceSet E → Bool)
      → Choice

ChoiceSet : Choice → Set
ChoiceSet (fin n)     =   Fin n
ChoiceSet (s ⊎' t)    =   ChoiceSet s ⊎ ChoiceSet t
ChoiceSet (subset' E f) = subset (ChoiceSet E) f
```

Interleaving operator

- In this process, the components P and Q execute completely independently of each other.
- Each event is performed by exactly one process.
- The rules in CSP expressing the operational semantics are as follows:

$$\frac{P \xrightarrow{\mu} P'}{P|||Q \xrightarrow{\mu} P'|||Q}\ \mu \neq \checkmark \qquad \frac{Q \xrightarrow{\mu} Q'}{P|||Q \xrightarrow{\mu} P|||Q'}\ \mu \neq \checkmark$$

$$\frac{P \xrightarrow{\checkmark} P' \qquad Q \xrightarrow{\checkmark} Q'}{P|||Q \xrightarrow{\checkmark} P'|||Q'}$$

Interleaving operator

We represent interleaving operator in CSP-Agda as follows:

$$
\begin{array}{l}
\_|||{+}{+}\_ : \{i : \mathsf{Size}\} \to \{c_0\ c_1 : \mathsf{Choice}\} \\
\quad \to \mathsf{Process+}\ i\ c_0 \to \mathsf{Process+}\ i\ c_1 \\
\quad \to \mathsf{Process+}\ i\ (c_0 \times'\ c_1)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E}\quad (P\ |||{+}{+}\ Q) & = & \mathsf{E}\ P \uplus' \mathsf{E}\ Q \\
\mathsf{Lab}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{Lab}\ P\ c \\
\mathsf{Lab}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_2\ c) & = & \mathsf{Lab}\ Q\ c \\
\mathsf{PE}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{PE}\ P\ c\ |||\infty{+}\ \ Q \\
\mathsf{PE}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_2\ c) & = & P\ |||{+}\infty\ \mathsf{PE}\ Q\ c \\
\mathsf{I}\quad (P\ |||{+}{+}\ Q) & = & \mathsf{I}\ P \uplus' \mathsf{I}\ Q \\
\mathsf{PI}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{PI}\ P\ c\ |||\infty{+}\ \ Q \\
\mathsf{PI}\ (P\ |||{+}{+}\ Q)\ (\mathsf{inj}_2\ c) & = & P\ |||{+}\infty\ \mathsf{PI}\ Q\ c \\
\mathsf{T}\quad (P\ |||{+}{+}\ Q) & = & \mathsf{T}\ P \times' \mathsf{T}\ Q \\
\mathsf{PT}\ (P\ |||{+}{+}\ Q)\ (c\ ,,\ c_1) & = & \mathsf{PT}\ P\ c\ ,,\ \mathsf{PT}\ Q\ c_1 \\
\mathsf{Str+}\ (P\ |||{+}{+}\ Q) & = & \mathsf{Str+}\ P\ |||\mathsf{Str}\ \mathsf{Str+}\ Q
\end{array}
$$

Interleaving operator

$$
\begin{array}{l}
\_|||\_ : \{i : \mathrm{Size}\} \to \{c_0\ c_1 : \mathrm{Choice}\} \to \mathrm{Process}\ i\ c_0 \\
\qquad \to \mathrm{Process}\ i\ c_1 \to \mathrm{Process}\ i\ (c_0 \times' c_1) \\
\mathrm{node}\ P\ |||\ \mathrm{node}\ Q \ = \ \mathrm{node}\ \ (P\ |||{+\!\!+}\ Q) \\
\mathrm{terminate}\ a\ |||\ \ Q \quad = \ \mathrm{fmap}\ \ (\lambda\ b \to (a\ ,,\ b))\ Q \\
P\ |||\ \ \mathrm{terminate}\ b \quad = \ \mathrm{fmap}\ \ (\lambda\ a \to (a\ ,,\ b))\ P
\end{array}
$$

Conclusion

- Definition of **coinductive data types** (coalgebras) by their **observations**.
  - Use of **copattern** matching
- **Interactive programs** and **objects** as examples of coalgebras.
- **State dependent objects**.
- **State dependent interactive programs** can be defined similarly.

Conclusion

- Definition of **processes coinductively** based on an **atomic one step operation**.
- Processes defined in a **monadic way**.
- **Copattern matching** allows to define operations for forming new processes **without the need for auxiliary function**.
  - Similarly to what's going on in object oriented programming.
- **Operations from process algebras** are **defined operations**.
  - External choice, internal choice, parallel operations, hiding, renaming, etc. have been defined, see TyDe 2016 paper.

Bibliography I

Andreas Abel, Stephan Adelsberger, and Anton Setzer.
Interactive programming in Agda – objects and graphical user interfaces.
To appear in Journal of Functional Programming. Preprint available at http://www.cs.swan.ac.uk/~csetzer/articles/ooAgda.pdf, 2016.

Bashar Igried and Anton Setzer.
Programming with monadic CSP-style processes in dependent type theory.
To appear in proceedings of TyDe 2016, Type-driven Development, preprint available from
http://www.cs.swan.ac.uk/~csetzer/articles/TyDe2016.pdf, 2016.

Bibliography II

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.

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.