Object-oriented Programming in Dependent Type Theory

Anton Setzer
Swansea University, Swansea UK
Joint work with Andreas Abel and Stephan Adelsberger
COST Action EUTYPES WG meeting, Ljubljana, Slovenia

31 January 2017

Old Version of Coalgebras: Codata Types

- 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

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

Objects as Elements of Coalgebras

- Coalgebras are used for modelling various phenomena related **infinite sequences of computations**.
  - Correspond to **non-well-founded trees**.
  - Arise when dealing with **interactive programs**.
    - Interactive programs often don't terminate unless terminated by the user.
- Coalgebras arise as representations of **real numbers**.
  - Examples: **streams of digits**, **Cauchy sequences**.
  - In general approximations by finite values
- Coalgebraic programming is heavily used in **object-oriented Programming**.
  - See section on objects below.

Solution: Coalgebras Defined by Observations

- **Problem of codata types:** Non-normalisation and undecidability of equality.
- Instead we define 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
```

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 $a, a, a, \ldots$

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

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

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

- Cons is **defined**:

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

Nested Pattern/Copattern Matching

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

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

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

Hello World in Agda

We can develop IO programs based on coalgebras and get the following hello world program:

```
module helloWorld where

open import ConsoleLib

main : ConsoleProg
main = run (WriteString "Hello World")
```

Object-Oriented/Based Programming

- Object-oriented (OO) programming is currently main programming paradigm.
  - Means that the main programming paradigm is essentially **coalgebraic programming**.
- 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.
  - Steps towards full OO programming work in progress.
- 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:

```
data CellMethod A : Set where
  get  :  CellMethod A
  put  :  A → CellMethod A

CellResult              :  ∀{A} → CellMethod A → Set
CellResult  {A} get     =  A
CellResult  (put _)     =  Unit

cellI                   :  (A : Set) → Interface
Method  (cellI A)       =  CellMethod A
Result  (cellI A) m     =  CellResult m
```

Definition of Cell

The cell object is defined as follows:

$$\begin{array}{l} \mathsf{Cell} : \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Cell}\ A = \mathsf{Object}\ (\mathsf{cellI}\ A) \end{array}$$

$$\begin{array}{llll} \mathsf{cell} : \{A : \mathsf{Set}\} \to A \to \mathsf{Cell}\ A & & & \\ \mathsf{objectMethod}\ (\mathsf{cell}\ a)\ \mathsf{get} & = & (\ a & ,\ \mathsf{cell}\ a\ ) \\ \mathsf{objectMethod}\ (\mathsf{cell}\ a)\ (\mathsf{put}\ b) & = & (\ \mathsf{unit} & ,\ \mathsf{cell}\ b\ ) \end{array}$$

State Dependent Interface

$$
\begin{array}{l}
\text{record Interface}^{\mathrm{s}} : \text{Set}_1 \text{ where} \\
\quad \text{field} \\
\qquad \begin{array}{lcl}
\text{State}^{\mathrm{s}} & : & \text{Set} \\
\text{Method}^{\mathrm{s}} & : & \text{State}^{\mathrm{s}} \to \text{Set} \\
\text{Result}^{\mathrm{s}} & : & (s : \text{State}^{\mathrm{s}}) \to (m : \text{Method}^{\mathrm{s}}\ s) \to \text{Set} \\
\text{next}^{\mathrm{s}} & : & (s : \text{State}^{\mathrm{s}}) \to (m : \text{Method}^{\mathrm{s}}\ s) \to \text{Result}^{\mathrm{s}}\ s\ m \\
 & & \to \text{State}^{\mathrm{s}}
\end{array}
\end{array}
$$

State Dependent Object

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

record Object$^{\mathrm{s}}$ (I : Interface$^{\mathrm{s}}$) (s : State$^{\mathrm{s}}$ I) : Set where
coinductive
field
objectMethod : (m : Method$^{\mathrm{s}}$ I s)
$\to$ Σ[$r \in$ Result$^{\mathrm{s}}$ I s m] Object$^{\mathrm{s}}$ I (next$^{\mathrm{s}}$ I s m r)

Example Safe Stack

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

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

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

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

Safe Stack

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

$$
\begin{array}{l}
\mathsf{stackO} : \forall\{E : \mathsf{Set}\}\ \{n : \mathbb{N}\}\ (v : \mathsf{Vec}\ E\ n) \\
\qquad \to \mathsf{Object}^{\mathrm{s}}\ (\mathsf{StackInterface}^{\mathrm{s}}\ E)\ n \\
\mathsf{objectMethod}\ (\mathsf{stackO}\ es)\ (\mathsf{push}\ e) = (\_\ ,\ \mathsf{stackO}\ (e :: es)) \\
\mathsf{objectMethod}\ (\mathsf{stackO}\ (e :: es))\ \mathsf{pop} = (e\ ,\ \mathsf{stackO}\ es)
\end{array}
$$

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

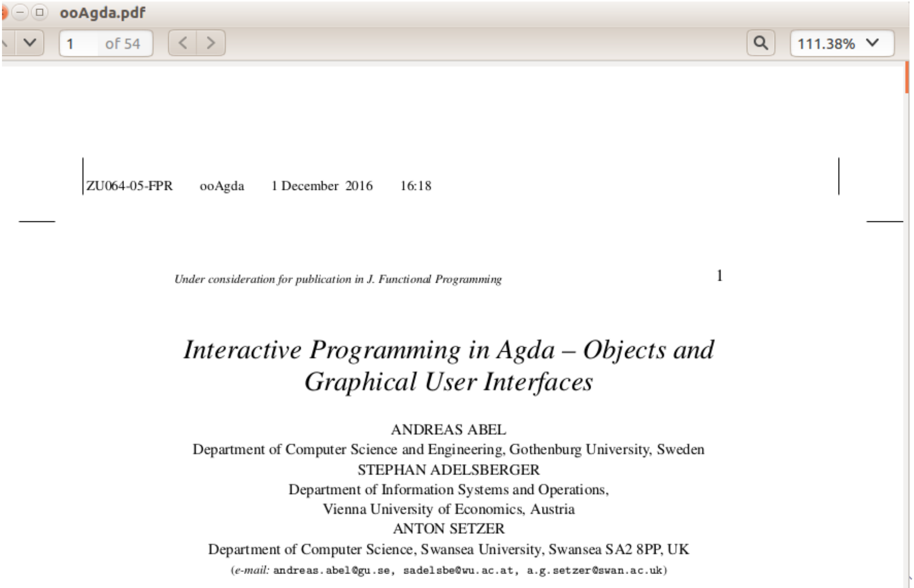
```
reduce : Stackmachine →  Stackmachine ⊎ ℕ
reduce (n , fib 0 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib 1 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib (suc (suc m)) , stack) =
   objectMethod stack (push (·+fib m))  ▷  λ { (_ , stack₁) →
   inj₁ ( suc n , fib (suc m) , stack₁)    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

Paper to appear in JFP [AAS16a]



ZU064-05-FPR ooAgda 1 December 2016 16:18

Under consideration for publication in J. Functional Programming 1

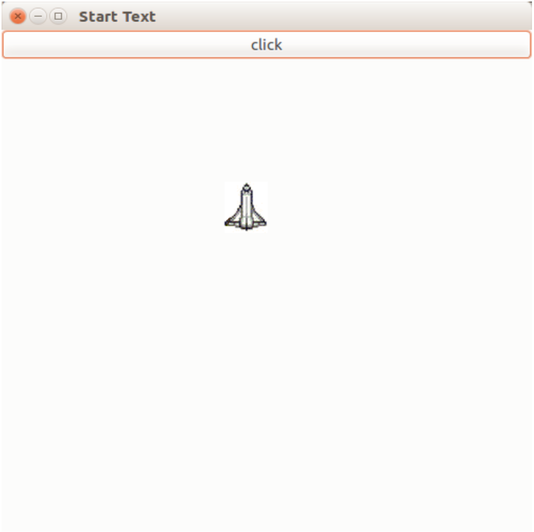
Interactive Programming in Agda – Objects and Graphical User Interfaces

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Sweden
STEPHAN ADELSBERGER
Department of Information Systems and Operations,
Vienna University of Economics, Austria
ANTON SETZER
Department of Computer Science, Swansea University, Swansea SA2 8PP, UK
(*e-mail:* andreas.abel@gu.se, sadelsbe@wu.ac.at, a.g.setzer@swan.ac.uk)

Results in [AAS16a]

- Development of GUIs in Agda.
  - Based on **server-side** programmings.
  - Use of **action listeners** which are part of an **object**
- Verification of laws of a safe and equivalences of implementations of stacks using bisimilarity.
- Library ooAgda on github [AAS16b].
- **Remark:** Library CSP-Agda for the process algebra CSP in Agda is now on github, see [IS17], article: [IS16].

SpaceShip Example



Dynamic Creation of Objects

- Idea is to create a set **Heap**, and pointers on the heap which dereference as objects.
- We have a state dependent object heap
  - depending on the size of the heap,
  - and methods for
    - dereferencing pointers,
    - updating pointers,
    - creating new pointers (which increases the size
- Currently working on linked and double linked lists built on the heap.
  - Goal is to create a proper queue.
  - Idea: Develop and verify the networking protocols such as the Chord protocol.

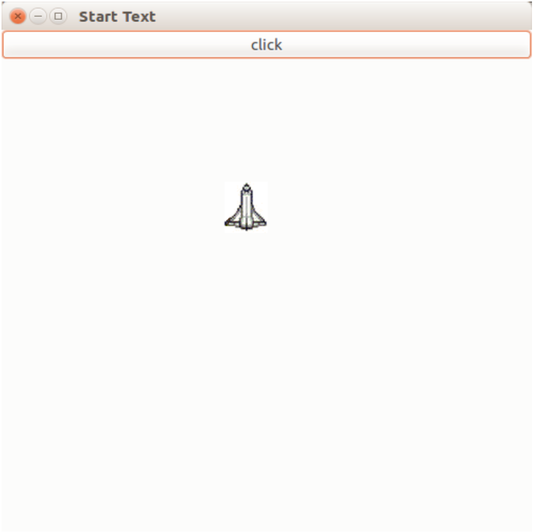
Current Challenges

- What is the right language?
  - At the moment in OO programs in Agda we need to introduce for every new instance of an object a new variable.
    - Can we write a library which hides these new variables?
    - Do we need new language constructs in Agda or (preferred) can we achieve this using the library?
- How to execute programs involving the heap efficiently.
  - At the moment heap is implemented as a list of heap elements.
  - Can we in a compiled version override this by calls to the "real" heap?
  - Do we obtain good performance of a queue?
  - Is it possible to write a true heap directly in Agda without overriding?

Conclusion

- Definition of **coinductive data types** (coalgebras) by their **observations**.
  - Use of **copattern** matching
- **Objects** as examples of coalgebras.
- **State dependent objects**.
- **Current work:** Developing of heap and dynamic creation of objects on the heap.

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

```
GuiLev2State : Set₁
GuiLev2State = VarList


data GuiLev2Command (s : GuiLev2State) : Set₁ where
  level1C           : GuiLev1Command → GuiLev2Command s
  createVar         : {A : Set} → A → GuiLev2Command s
  setButtonHandler  : Button
                      → List (prod s
                         → IO GuiLev1Interface ∞ (prod s))
                      → GuiLev2Command s
  setOnPaint        : Frame
                      → List (prod s → DC → Rect
                         → IO GuiLev1Interface ∞ (prod s))
                      → GuiLev2Command s
```

Graphics Level2 Response + Next

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

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

Graphics Level2 Interface

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

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

$$\text{onPaint} \;:\; \text{ActionHandler} \to \text{DC} \to \text{Rect} \to \text{IO GuiLev1Interface ActionHandler}$$
$$\text{onPaint}\; \mathit{obj}\; \mathit{dc}\; \mathit{rect} = \text{mapIO}\;\; \text{proj}_2\;\; (\text{method}\; \mathit{obj}\; (\text{onPaintM}\; \mathit{dc}\; \mathit{rect}))$$

$$\text{moveSpaceShip} : \text{Frame} \to \text{ActionHandler} \to \text{IO}\;\; \text{GuiLev1Interface ActionHandler}$$
$$\text{moveSpaceShip}\; \mathit{fra}\; \mathit{obj} \;=\; \text{mapIO}\;\; \text{proj}_2\;\; (\text{method}\; \mathit{obj}\; (\text{moveSpaceShipM}\; \mathit{fra}))$$

Action Handlers

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

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

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

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)
```

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.

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

Bibliography II

Bashar Igried and Anton Setzer.
Programming with monadic CSP-style processes in dependent type theory.
In Proceedings of the 1st International Workshop on Type-Driven Development, TyDe 2016, pages 28–38, New York, NY, USA, 2016. ACM.
http://doi.acm.org/10.1145/2976022.2976032.

Bashar Igried and Anton Setzer.
CSP-Agda.
Agda library. Available at https://github.com/csetzer/cspagdaPublic, 2017.

Bibliography III

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/$\sim$csetzer/index.html.

Anton Setzer.
How to reason coinductively informally.
In Reinhard Kahle, Thomas Strahm, and Thomas Studer, editors, Advances in Proof Theory, pages 377–408. Springer, 2016.
Doi page http://link.springer.com/chapter/10.1007%2F978-3-319-29198-7_12, authors copy at http://www.cs.swan.ac.uk/~csetzer/articles/jaeger60Birthdaymain.pdf.