Programming with GUIs in Agda

Anton Setzer
Swansea University, Swansea UK
(Joint work with Andreas Abel and Stephan Adelsberger)
Agda Implementors' Meeting XXV, Gothenburg, Sweden

12 May 2017

ooAgda JFP-Paper [AAS17]

JFP 27, e8, 54 pages, 2017. © Cambridge University Press 2017 1
doi:10.1017/S0956796816000319

Interactive programming in Agda – Objects and graphical user interfaces

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Sweden
(*e-mail:* andreas.abel@gu.se)

STEPHAN ADELSBERGER
Department of Information Systems and Operations, Vienna University of Economics, Austria
(*e-mail:* sadelsbe@wu.ac.at)

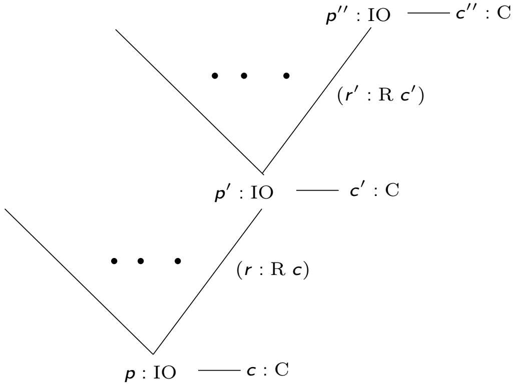
ANTON SETZER
Department of Computer Science, Swansea University, Swansea SA2 8PP, UK
(*e-mail:* a.g.setzer@swan.ac.uk)

Abstract

We develop a methodology for writing interactive and object-based programs (in the sense of Wegner) in dependently typed functional programming languages. The methodology is

Library: https://github.com/agda/ooAgda

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
```

IO

```
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
```

Running Interactive Programs

```
{-# NON_TERMINATING #-}
translateIO : ∀ {A}
               (tr : (c : C) → NativeIO (R c))
               (p : IO∞ I A)
               → NativeIO A
translateIO tr p = case (force p) 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.

A First Interactive Program

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

main : NativeIO Unit
main = translateIOConsole cat
```

99 Bottles of Beer

- Andreas Abel
  - wrote a version of 99 Bottles of Beer program
  - based on the Haskell program,
  - submitted it to http://www.99-bottles-of-beer.net/

Output of 99 Bottles of Beer Program

99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

. . .

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

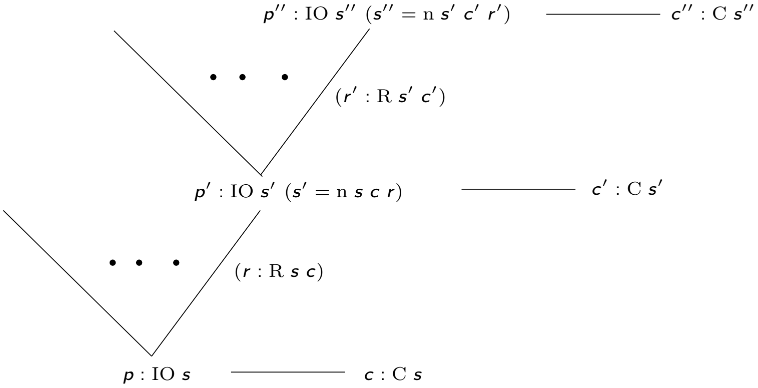
99 Bottles in ooAgda

```
bottles : ℕ → String
bottles 0 = "no more bottles"
bottles 1 = "1 bottle"
bottles n = show n ++ " bottles"

verse : ℕ → String
verse 0 = "No more bottles of beer on the wall,"
        ++ "no more bottles of beer.\n"
        ++ "Go to the store and buy some more,"
        ++ "99 bottles of beer on the wall."
verse (suc n) = bottles (suc n)
              ++ " of beer on the wall, "
              ++ bottles (suc n)
              ++ " of beer.\n"
              ++ "Take one down and pass it around, "
              ++ bottles n
```

99 Bottles in ooAgda

```
main : ConsoleProg
main = run (sequenceIO (map (WriteString ∘ verse) (downFrom 100)))
```

State Dependent IO-Trees



State Dependent IO – Interface

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

State Dependent IO

$$
\begin{array}{l}
\text{record } \mathsf{IO}^{\mathrm{s}}\ (A : S \to \mathsf{Set})\ (s : S) : \mathsf{Set}_1 \text{ where} \\
\qquad \text{coinductive} \\
\qquad \text{field} \\
\qquad\quad \mathsf{force}^{\mathrm{s}} : \mathsf{IO}^{\mathrm{s}}{}'\ A\ s
\end{array}
$$

$$
\begin{array}{l}
\text{data } \mathsf{IO}^{\mathrm{s}}{}'\ (A : S \to \mathsf{Set}) : S \to \mathsf{Set}_1 \text{ where} \\
\quad \mathsf{do}^{\mathrm{s}}{}' : \quad \{s : S\}\ (c : C\ s) \\
\qquad\qquad\quad (f : (r : R\ s\ c) \to \mathsf{IO}^{\mathrm{s}}\ A\ (next\ s\ c\ r)) \\
\qquad\qquad\quad \to \mathsf{IO}^{\mathrm{s}}{}'\ A\ s \\
\quad \mathsf{return}^{\mathrm{s}}{}' : \{s : S\}\ (a : A\ s) \to \mathsf{IO}^{\mathrm{s}}{}'\ A\ s
\end{array}
$$

Objects

- An object is a server-side interactive program
- It receives method calls, and depending on the method returns an element of the return type.
- 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: Cell Interface

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:

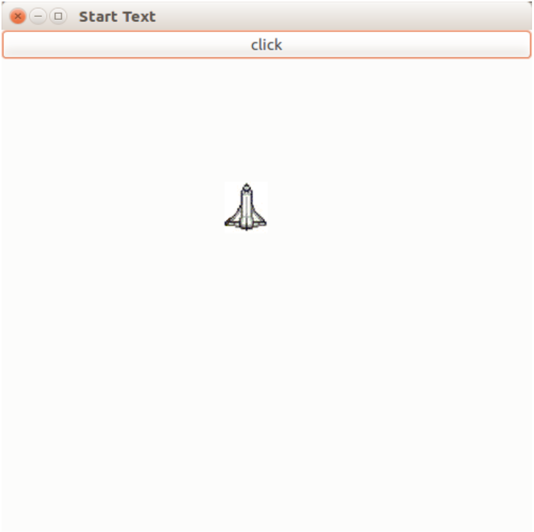
```
Cell : Set → Set
Cell A = Object (cellI A)

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:

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

SpaceShip Example



WxHaskell

- The use of WxHaskell and MVar in Agda is work by Stephan Adelsberger.
- Haskell library for writing GUIs which supports server side programs
- Examples:
  - frame [text := "Frame Title"]

    Will create a frame with title Frame Title.
  - set myframe [on paint := prog]

    sets for myframe the on paint method to execute prog, where

    prog :: IO ()
  - Similar code allows to set action listeners to buttons.

MVar

- We need to share values between the different action handlers.
- Action listeners can be executed in parallel.
- Use of MVar to communicate values between action handlers.
- MVar are a mutual location which can be empty or contain a value of a given type.
- There are commands for
  - creating a new MVar
  - putting a value into an MVar
  - taking a value out of an MVar.

Defining MVar

```
postulate MVar : Set → Set
{-# COMPILE GHC MVar = type Control.Concurrent.MVar #-}

Var : Set → Set
Var = MVar
```

IO programs for handling MVar

```
postulate
  nativeNewVar  : {A : Set} → A     → NativeIO (Var A)
  nativeTakeVar : {A : Set} → Var A → NativeIO A
  nativePutVar  : {A : Set} → Var A → A → NativeIO Unit

{-# COMPILE GHC nativeNewVar = (\ _ -> Control.Concurrent.newM
{-# COMPILE GHC nativeTakeVar = (\ _ -> Control.Concurrent.takeM
{-# COMPILE GHC nativePutVar = (\ _ -> Control.Concurrent.putMV
```

Thread Safety of MVar

- A thread running nativePutVar
  - blocks until the MVar is empty,
  - then puts a value into that location.
- A thread running nativeTakeVar
  - blocks until the variable is non-empty,
  - then reads the value,
  - leaving the location empty.

Variable Lists

- We want to deal with multiple Variable Lists:

```
data VarList : Set₁ where
  []     : VarList
  addVar : (A : Set) → Var A → VarList → VarList
```

- We form the product of its elements:

```
prod : VarList → Set
prod []               = Unit
prod (addVar A v []) = A
prod (addVar A v l)  = A × prod l
```

Variable Lists

- We lift nativeTakeVar, nativePutVar to VarList:

$$\text{takeVar} : (l : \text{VarList}) \to \text{NativeIO}\ (\text{prod}\ l)$$

$$\text{putVar} : (l : \text{VarList}) \to \text{prod}\ l \to \text{NativeIO Unit}$$

Dispatch

- An action handler will now
  - take the variables from our current varlist
  - execute some IO commands
  - modify those values
  - and put them back into the current varlist:

$$
\begin{aligned}
\mathsf{dispatch} :\ & (l : \mathsf{VarList}) \\
& (handler : \mathsf{prod}\ l \rightarrow \mathsf{NativeIO}\ (\mathsf{prod}\ l)) \\
& \rightarrow \mathsf{NativeIO\ Unit} \\
\mathsf{dispatch}\ l\ handler = \ & \mathsf{takeVar}\ l\ \mathsf{native}{\gg=}\ \lambda\ a \ \rightarrow \\
& handler\ a\ \mathsf{native}{\gg=}\ \lambda\ a_1 \rightarrow \\
& \mathsf{putVar}\ l\ a_1
\end{aligned}
$$

Running Multiple Handlers in Sequence

- While an action handler is running, it is blocking the VarList and therefore other action handlers.
- We want to trigger other action handlers from one action handlers, and want to allow them to execute in between an action handler.
- Therefore we replace action handlers by a list of action handlers, which are run in sequence.

```
dispatchList : (l : VarList)
               (handler : List (prod l → NativeIO (prod l)))
               → NativeIO Unit
dispatchList l []          = nativeReturn _
dispatchList l (p :: rest) = dispatch l p native>>= λ _ →
                             dispatchList l rest
```

Two Levels of IO programs

- We obtain two IO interfaces.
  - Level 1 is the IO interface for writing action handlers.
    We add to it all commands which don't make use of action handlers.
  - Level 2 is in which the program is written which
    - creates the GUI
    - adds level 1 action handlers to events.
  - It contains all Level 1 commands.
  - For size reasons Level 2 will be in Set_1.
  - It contains as well operations for creating variables.
  - It is a **state dependent** interface, depending on the created variables.

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
```

Graphics Interface Level1

```
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

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

$$
\begin{aligned}
&\text{GuiLev2Next} : (s : \text{GuiLev2State}) \to (c : \text{GuiLev2Command } s) \\
&\qquad\qquad\quad \to \text{GuiLev2Response } s\ c \\
&\qquad\qquad\quad \to \text{GuiLev2State} \\
&\text{GuiLev2Next } s\ (\text{createVar } \{A\}\ a)\ \mathit{var} = \text{addVar } A\ \mathit{var}\ s \\
&\text{GuiLev2Next } s\ \_\ \_ \qquad\qquad\qquad\quad = s
\end{aligned}
$$

Graphics Level2 Interface

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

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

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

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

Action Handlers

$$\text{callRepaint} : \text{Frame} \rightarrow \text{ActionHandler} \\ \rightarrow \text{IO GuiLev1Interface ActionHandler}$$

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

$$\text{buttonHandler} : \text{Frame} \rightarrow \text{List (ActionHandler} \\ \rightarrow \text{IO GuiLev1Interface ActionHandler)}$$

$$\text{buttonHandler } \mathit{fra} = \text{moveSpaceShip } \mathit{fra} :: [\ \text{callRepaint } \mathit{fra}\ ]$$

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)
```

Conclusion

- **Objects** are essentially **interactive programs**.
- Writing simple interactive programs is relatively easy.
  - Challenge: write your little program in Agda instead of awk, sed, perl, python, ...
- **State dependent interactive programs**.
- **State dependent objects** can be defined similarly.

Conclusion

- **WxHaskell** as a suitable library for server side programs.
- Use of MVar to communicate between threads.
- **2 levels** of IO interfaces needed for dealing with action handlers.
- Handling of Graphical User Interfaces using **action listeners** similar to what is done in Java.
- Bundling of action listeners into one object.
- Writing GUIs in Agda seems feasible.

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.