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
Interactive Programs in Agda

Objects

GUIs

Conclusion

Bibliography
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 implemented in the ooAgda library, ooAgda provides a syntax similar to the one used in

Library: https://github.com/agda/ooAgda
Interactive Programs in Agda

Objects

GUIs

Conclusion

Bibliography
IO-Trees (Non-State Dependent)

\[ p : \text{IO} \quad \overset{r}{\longrightarrow} \quad c : \text{C} \]

\[ p' : \text{IO} \quad \overset{(r : \text{R} \ c)}{\longrightarrow} \quad c' : \text{C} \]

\[ p'' : \text{IO} \quad \overset{(r' : \text{R} \ c')}{\longrightarrow} \quad c'' : \text{C} \]
An IOInterface is a record having fields Command and Response:

```agda
class IOInterface : Set₁ where
  field Command : Set
  Response : Command → Set
```
Interactive Programs in Agda

**IO**

mutual

record \( \text{IO} \omega \) (\( I : \text{IOInterface} \)) (\( A : \text{Set} \)) : \text{Set} where

coinductive

field force : \( \text{IO} \ I \ A \)

data \( \text{IO} \) (\( I : \text{IOInterface} \)) (\( A : \text{Set} \)) : \text{Set} where

\[ \text{do} : (c : \text{Command} \ I) (f : \text{Response} \ I \ c \to \text{IO} \omega \ I \ A) \to \text{IO} \ I \ A \]

\[ \text{return} : A \to \text{IO} \ I \ A \]
Interactive Programs in Agda

Running Interactive Programs

\{-# NON_TERMINATING #-\}

\[\text{translateIO} : \forall \{A\} \]

\[ (tr : (c : C) \rightarrow \text{NativeIO} (R c)) \]

\[ (p : \text{IO}\infty \ I A) \]

\rightarrow \text{NativeIO} A

\text{translateIO} \ tr \ p = \text{case (force } p\text{) of } \lambda

\{ \text{ (do } c \ f) \rightarrow (tr \ c) \text{ native} \gg= \lambda \ r \rightarrow \text{translateIO} \ tr \ (f \ r) \}

\; (\text{return } a) \rightarrow \text{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/

Welcome to 99 Bottles of Beer

This Website holds a collection of the Song 99 Bottles of Beer programmed in different programming languages. Actually the song is represented in 1500 different programming languages and variations. For more detailed information refer to historic information.

All these little programs generate the lyrics to the song 99 Bottles of Beer as an output. In case you do not know the song, you will find the lyrics to the song here.
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 : \mathbb{N} \rightarrow \text{String}
bottles 0 = "no more bottles"
bottles 1 = "1 bottle"
bottles \text{n} = \text{show } \text{n} \llap{+} + " bottles"

verse : \mathbb{N} \rightarrow \text{String}
verse 0 = "No more bottles of beer on the wall,"
   \llap{+} + "no more bottles of beer.\n"
   \llap{+} + "Go to the store and buy some more,"
   \llap{+} + "99 bottles of beer on the wall."
verse (\text{suc } \text{n}) = \text{bottles (suc } \text{n})
   \llap{+} + " of beer on the wall, "
   \llap{+} + \text{bottles } (\text{suc } \text{n})
   \llap{+} + " of beer.\n"
   \llap{+} + "Take one down and pass it around, "
   \llap{+} + \text{bottles } \text{n}
Interactive Programs in Agda

99 Bottles in ooAgda

main : ConsoleProg
main = run (sequenceIO (map (WriteString ◦ verse) (downFrom 100)))
State Dependent IO-Trees

\[
p'' : \text{IO } s'' \quad (s'' = n \ s' \ c' \ r') \quad \quad c'' : \text{C } s''
\]

\[
p' : \text{IO } s' \quad (s' = n \ s \ c \ r) \quad \quad c' : \text{C } s'
\]

\[
\begin{align*}
\bullet & \quad \bullet & \quad \bullet \\
(r' : \text{R } s' \ c')
\end{align*}
\]

\[
p : \text{IO } s \quad \quad c : \text{C } s
\]

\[
\begin{align*}
\bullet & \quad \bullet & \quad \bullet \\
(r : \text{R } s \ c)
\end{align*}
\]
record IOInterface$^s$ : Set$_2$ where

  field

  State$^s$ : Set$_1$

  Command$^s$ : (s : State$^s$s) → Set$_1$

  Response$^s$ : (s : State$^s$s)(c : Command$^s$s s) → Set

  next$^s$ : (s : State$^s$s)(c : Command$^s$s s)
  (r : Response$^s$s s c)
  → State$^s$s
State Dependent IO

record \( \text{IO}^s (A : S \rightarrow \text{Set}) (s : S) : \text{Set}_1 \) where
coinductive
field
  force\(^s\) : \( \text{IO}^{s'} A s \)

data \( \text{IO}^{s'} (A : S \rightarrow \text{Set}) : S \rightarrow \text{Set}_1 \) where
  do\(^{s'}\) : \{s : S\} (c : C s)
    (f : (r : R s c) \rightarrow \text{IO}^s A (\text{next} s c r))
    \rightarrow \text{IO}^{s'} A s
  return\(^{s'}\) : \{s : S\} (a : A s) \rightarrow \text{IO}^{s'} A s
Interactive Programs in Agda

Objects

GUIs

Conclusion

Bibliography
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 : \((m : Method \ I) \rightarrow Result \ I \ m \times Object \ I\)
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 : ∀{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
The cell object is defined as follows:

\[
\text{Cell} : \text{Set} \to \text{Set} \\
\text{Cell } A = \text{Object } (\text{cellI } A)
\]

\[
\text{cell} : \{A : \text{Set}\} \to A \to \text{Cell } A
\]

\[
\text{objectMethod } (\text{cell } a) \text{ get } = (a, \text{cell } a)
\]

\[
\text{objectMethod } (\text{cell } a) \text{ (put } b) = (\text{unit, cell } b)
\]
IO Objects are like Objects, but methods execute an interactive program before returning the result:

```
record IOObject (l_io : IOInterface) (I : Interface) : Set where
  coinductive
  field method : (m : Method I) → IO∞ l_io (Result I m × IOObject l_io I)
```
Interactive Programs in Agda

Objects

GUIs

Conclusion

Bibliography
SpaceShip Example

Start Text

click
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
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.newMVar) #-}
{-# COMPILE GHC nativeTakeVar = (\_ -> Control.Concurrent.takeMVar) #-}
{-# COMPILE GHC nativePutVar = (\_ -> Control.Concurrent.putMVar) #-}
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:

\[
\text{data VarList : Set}_1 \text{ where} \\
[] : \text{VarList} \\
\text{addVar : } (A : \text{Set}) \rightarrow \text{Var } A \rightarrow \text{VarList } \rightarrow \text{VarList}
\]

- We form the product of its elements:

\[
\text{prod : VarList } \rightarrow \text{Set} \\
\text{prod } [] = \text{Unit} \\
\text{prod } (\text{addVar } A \nu []) = A \\
\text{prod } (\text{addVar } A \nu l) = A \times \text{prod } l
\]
Variable Lists

- We lift nativeTakeVar, nativePutVar to VarList:

  \[
  \text{takeVar} : (l : \text{VarList}) \rightarrow \text{NativeIO} \ (\text{prod} \ l) \\
  \text{putVar} : (l : \text{VarList}) \rightarrow \text{prod} \ l \rightarrow \text{NativeIO} \ \text{Unit}
  \]
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:

\[
\text{dispatch} : (l : \text{VarList}) \\
\text{handler} : \text{prod } l \to \text{NativeIO (prod } l)) \\
\to \text{NativeIO Unit}
\]

\[
\text{dispatch } l \text{ handler } = \text{takeVar } l \text{ native} \gg= \lambda a \to \\
\text{handler } a \text{ native} \gg= \lambda a_1 \to \\
\text{putVar } l \ a_1
\]
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.

```agda
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 $\text{Set}_1$.
- It contains as well operations for creating variables.
- It is a state dependent interface, depending on the created variables.
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

\[
\text{GuiLev1Interface : IOInterface} \\
\text{Command GuiLev1Interface} = \text{GuiLev1Command} \\
\text{Response GuiLev1Interface} = \text{GuiLev1Response}
\]
Graphics Level2 Commands

\[
\text{GuiLev2State} : \text{Set}_1 \\
\text{GuiLev2State} = \text{VarList}
\]

data \text{GuiLev2Command} \ (s : \text{GuiLev2State}) : \text{Set}_1 \ 	ext{where} \\
\text{level1C} : \text{GuiLev1Command} \rightarrow \text{GuiLev2Command} \ s \\
\text{createVar} : \{A : \text{Set}\} \rightarrow A \rightarrow \text{GuiLev2Command} \ s \\
\text{setButtonHandler} : \text{Button} \\
\quad \rightarrow \ 	ext{List} \ (\text{prod} \ s \rightarrow \text{IO GuiLev1Interface} \ appet (\text{prod} \ s)) \\
\quad \rightarrow \ 	ext{GuiLev2Command} \ s \\
\text{setOnPaint} : \text{Frame} \\
\quad \rightarrow \ 	ext{List} \ (\text{prod} \ s \rightarrow \text{DC} \rightarrow \text{Rect} \rightarrow \text{IO GuiLev1Interface} \ appet (\text{prod} \ s)) \\
\quad \rightarrow \ 	ext{GuiLev2Command} \ s
Graphics Level2 Response + Next

\[
\text{GuiLev2Response} : (s : \text{GuiLev2State}) \rightarrow \text{GuiLev2Command } s \\
\rightarrow \text{Set}
\]

\[
\text{GuiLev2Response }_\_ (\text{level1C } c) = \text{GuiLev1Response } c \\
\text{GuiLev2Response }_\_ (\text{createVar } \{A\} a) = \text{Var } A \\
\text{GuiLev2Response }_\_ _ = \text{Unit}
\]

\[
\text{GuiLev2Next} : (s : \text{GuiLev2State}) \rightarrow (c : \text{GuiLev2Command } s) \\
\rightarrow \text{GuiLev2Response } s c \\
\rightarrow \text{GuiLev2State}
\]

\[
\text{GuiLev2Next } s (\text{createVar } \{A\} a) \text{ var} = \text{addVar } A \text{ var } s \\
\text{GuiLev2Next } s _ _ = s
\]
GUIs

Graphics Level2 Interface

\[ \text{GuiLev2Interface} : \text{IOInterface}^s \]

\[ \text{State}^s \quad \text{GuiLev2Interface} = \text{GuiLev2State} \]

\[ \text{Command}^s \quad \text{GuiLev2Interface} = \text{GuiLev2Command} \]

\[ \text{Response}^s \quad \text{GuiLev2Interface} = \text{GuiLev2Response} \]

\[ \text{next}^s \quad \text{GuiLev2Interface} = \text{GuiLev2Next} \]
Action Handling Object

data ActionHandlerMethod : Set where
  onPaintM : DC \rightarrow Rect \rightarrow ActionHandlerMethod
  moveSpaceShipM : Frame \rightarrow ActionHandlerMethod
  callRepaintM : Frame \rightarrow ActionHandlerMethod

ActionHandlerResult : ActionHandlerMethod \rightarrow Set
ActionHandlerResult _ = Unit

ActionHandlerInterface : Interface
Method ActionHandlerInterface = ActionHandlerMethod
Result ActionHandlerInterface = ActionHandlerResult

ActionHandler : Set
ActionHandler = IOObject GuiLev1Interface ActionHandlerInterface
**Action Handling Object**

\[
\text{actionHandler} : \mathbb{Z} \rightarrow \text{ActionHandler}
\]

method \((\text{actionHandler} \ z) \ (\text{onPaintM} \ dc \ rect) =
\]
\[
do\infty \ (\text{drawBitmap} \ dc \ \text{ship} \ (z, (+150)) \ \text{true}) \ \lambda_\_ \rightarrow
\]
\[
\text{return}\infty \ (\text{unit}, \text{actionHandler} \ z)
\]

method \((\text{actionHandler} \ z) \ (\text{moveSpaceShipM} \ fra) =
\]
\[
\text{return}\infty \ (\text{unit}, \text{actionHandler} \ (z + (+20)))
\]

method \((\text{actionHandler} \ z) \ (\text{callRepaintM} \ fra) =
\]
\[
do\infty \ (\text{repaint} \ fra) \ \lambda_\_ \rightarrow
\]
\[
\text{return}\infty \ (\text{unit}, \text{actionHandler} \ z)
\]

\[
\text{actionHandlerInit} : \text{ActionHandler}
\]

\[
\text{actionHandlerInit} = \text{actionHandler} (\ +150)
\]
Action Handlers

\[
onPaint : \text{ActionHandler} \rightarrow \text{DC} \rightarrow \text{Rect}
\rightarrow \text{IO GuiLev1Interface ActionHandler}
onPaint \ obj \ dc \ rect = \text{mapI}O \ \text{proj}_2 \ (\text{method} \ obj \ (\text{onPaintM} \ dc \ rect))
\]

\[
\text{moveSpaceShip} : \text{Frame} \rightarrow \text{ActionHandler}
\rightarrow \text{IO GuiLev1Interface ActionHandler}
\text{moveSpaceShip} \ fra \ obj = \text{mapI}O \ \text{proj}_2
(\text{method} \ obj \ (\text{moveSpaceShipM} \ fra))
\]
Action Handlers

callRepaint : Frame → ActionHandler → IO GuiLev1Interface ActionHandler

callRepaint fra obj = mapIO proj₂ (method obj (callRepaintM fra))

buttonHandler : Frame → List (ActionHandler → IO GuiLev1Interface ActionHandler)

buttonHandler fra = moveSpaceShip fra :: [ callRepaint fra ]
Spaceship Program

```agda
program : IO^s GuiLev2Interface (λ _ → Unit) []
program = do^s (level1C makeFrame) λ fra →
          do^s (level1C (makeButton fra)) λ bt →
          do^s (level1C (addButton fra bt)) λ _ →
          do^s (createVar actionHandlerInit) λ _ →
          do^s (setButtonHandler bt (moveSpaceShip fra :
                      :: [ callRepaint fra ])) λ _ →
          do^s (setOnPaint fra [ onPaint ])
          return^s

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.


Anto Setzer.
How to reason coinductively informally.