Interactive Programs in Agda

Anton Setzer
(Swansea)

1. Defining IO in Agda.
2. Execution of IO Programs.
3. Dealing with Complex Programs.
4. A Graphics Library for Agda

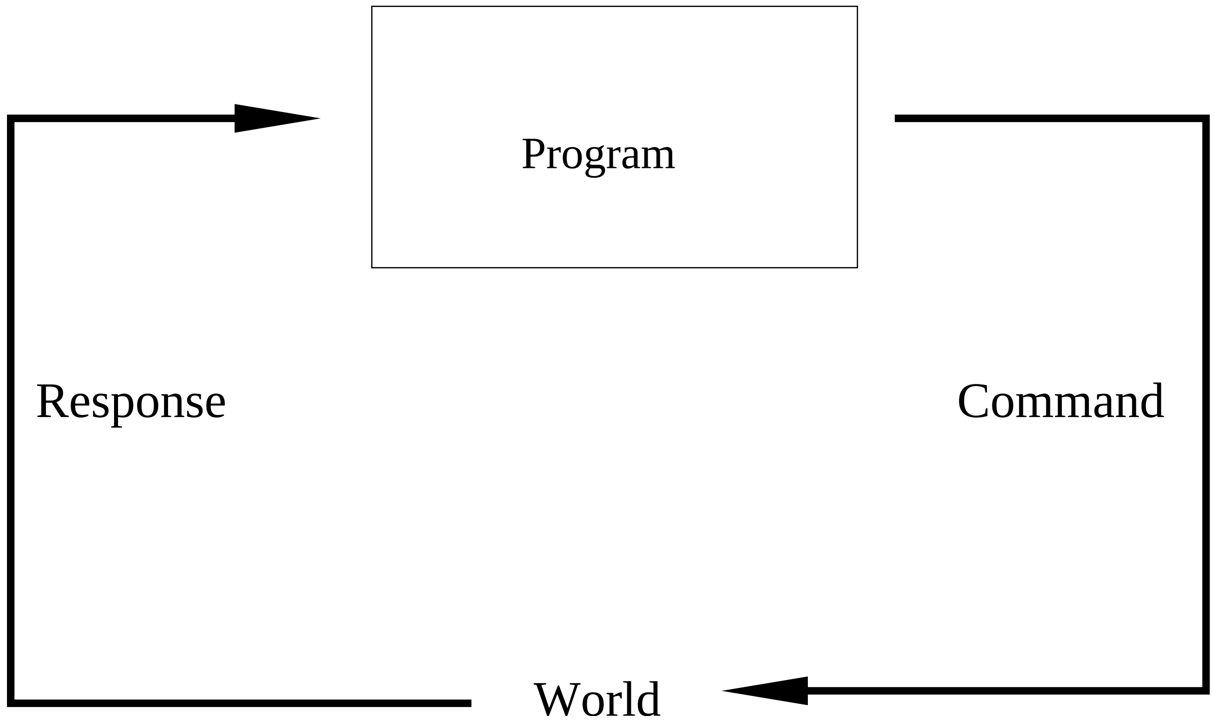
1. Defining IO in Agda

- Critical Systems are interactive. We need to be able to prove the correctness of interactive programs.
- Programming with Dependent Types only convincing, if we can write interactive programs.

1. Interfaces

- We consider programs which interact with the real world:
  - They issue a command . . .
    (e.g.
    (1) get last key pressed;
    (2) write character to terminal;
    (3) set traffic light to red)
  - . . . and obtain a response, depending on the command . . .
    (e.g.
    - in (1) the key pressed
    - in (2), (3) a trivial element indicating that this was done, or a message indicating success or an error element).

Interactive Programs



Interface in Agda

- Interface for interactive program given by
  - A set of commands the program can issue

$$\mathrm{C} : \mathrm{Set}$$

  - A set of responses, depending on commands

$$\mathrm{R} : \mathrm{C} \to \mathrm{Set}$$

Interactive Programs in Agda

- Interactive programs in Agda given by a sequence of commands, and interactive programs depending on the responses.
- Additionally we want programs to terminate giving result $a : A$ for some $A : \mathrm{Set}$.
- We need to allow non-terminating programs. Therefore the type needs to be defined coinductively.

IO Monad in Agda

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{codata}\ \mathrm{IO}\ (C : \mathrm{Set})\ (R : C \longrightarrow \mathrm{Set})\ (A : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}} \\
\quad \mathrm{do} & : & (c : C) \longrightarrow (f : \mathrm{R}\ c \longrightarrow \mathrm{IO}\ C\ R\ A) \longrightarrow \mathrm{IO}\ C\ R\ A \\
\quad \mathrm{return} & : & (a : A) \longrightarrow \mathrm{IO}\ C\ R\ A
\end{array}$$

Monad Operations

- $\eta :=$ return.
- $>>=$ can be defined:

$$
\begin{array}{lcl}
\_ >>= \_ & : & \{C : \mathrm{Set}\} \to \{R : C \to \mathrm{Set}\} \to \{A\ B : \mathrm{Set}\} \\
& & \to \mathrm{IO}\ C\ R\ A \\
& & \to (A \to \mathrm{IO}\ C\ R\ B) \\
& & \to \mathrm{IO}\ C\ R\ B \\
\mathrm{do}\ c\ f >>= q & = & \mathrm{do}\ c\ (\lambda x \to f\ x >>= q) \\
\mathrm{return}\ a >>= q & = & q\ a
\end{array}
$$

IO in Haskell

- There is one uniform IO type in Haskell. We call its translated version

$$\mathrm{nativeIO} : \mathrm{Set} \rightarrow \mathrm{Set}$$

- We can import it together with the monad operations as follows:

Importing nativeIO

```
postulate
  nativeIO   : Set -> Set
  nativeReturn : { A : Set} -> A -> nativeIO A
  _native>>=_   : {A B : Set} -> nativeIO A
                            -> (A -> nativeIO B)
                            -> nativeIO B

{-# COMPILED_TYPE nativeIO IO #-}
{-# COMPILED _native>>=_
 (\_ _ -> (>>=) :: IO a -> (a -> IO b) -> IO b) #-}
{-# COMPILED nativeReturn
 (\_ -> return :: a -> IO a) #-}
```

Simple nativeIO Operations

- Simple nativeIO Operations in Haskell have the form

$$\text{operation} : A_1 \to A_2 \to \cdots \to A_n \to \text{IO}\,B$$

- A collection of such operations can be represented in the true IO type as follows:
  - We form an interface C, R for all operations relevant.
  - C is an inductive data type, with constructors for each ioProg corresponding to the IO type, so we have constructor

$$\text{operationC} : A_1 \to A_2 \to \cdots \to A_n \to \mathrm{C}$$

  - $\mathrm{R} : \mathrm{C} \to \mathrm{Set}$ is defined by case distinction, e.g.

$$\mathrm{R}\ (\text{operationC}\ a_1\ \ldots\ a_n) = B$$

Example

```
postulate
  nativePutStrLn  : String -> nativeIO Unit
  nativeGetLine   : nativeIO String

{-# COMPILED nativePutStrLn putStrLn #-}
{-# COMPILED nativeGetLine getLine #-}
```

Example

```
data ConsoleCommands : Set where
  putStrLn : String -> ConsoleCommands
  getLine  : ConsoleCommands


ConsoleResponses : ConsoleCommands -> Set
ConsoleResponses (putStrLn s) = Unit
ConsoleResponses  getLine     = String

IOConsole : Set -> Set
IOConsole = IO ConsoleCommands ConsoleResponses
```

2. Execution of IO Programs

- In order to define a generic translation Function we assume for our interface C, R a function

$$\mathrm{translateLocal} : (c : \mathrm{C}) \rightarrow \mathrm{nativeIO}\ (\mathrm{R}\ c)$$

Example

```
translateIOConsoleLocal : (c : ConsoleCommands)
                          -> nativeIO (ConsoleResponses c)
translateIOConsoleLocal (putStrLn s) = nativePutStrLn s
translateIOConsoleLocal getLine      =  nativeGetLine
```

Generic Translation

```
translateGeneric :
    forall {A C R}
    ->  (translateLocal : (c : C) -> nativeIO (R c))
    ->  IO C R A
    ->  nativeIO A
translateGeneric translateLocal (do c f) =
                   (translateLocal c) native>>=
                   (\ r
                    -> translateGeneric translateLocal (f r
translateGeneric translateLocal (return a) =
                   nativeReturn a
```

Execution

- An interactive program can now be executed by defining an element $\mathrm{main} : \mathrm{nativeIO}\ A$

Example

```
myProgram : IOConsole Unit
myProgram =  do getLine (\ line -> (
             do (putStrLn line) (\ _ -> (
             do (putStrLn line) (\ _ ->
             myProgram)))))

main : nativeIO Unit
main = translateIOConsole myProgram
```

Termination Checker

- The translation from IO to $\mathrm{nativeIO}$ doesn't termination check.
- The definition of a specific element of $\mathrm{IO}\ \mathrm{C}\ \mathrm{R}$ termination checks, if defined by guarded recursion.
  - IO, $>>=$, $\mathrm{translateGeneric}$ and specific C, R, together with translateLocal can be defined in a library, where termination checker is switched off.
  - User defined code can be termination checked.

3. Dealing with Complex Programs

- When defining recursive programs in $\mathrm{IO}\ C\ R\ A$ we are restricted to a sequence of constructors.
- Especially we are not allowed to use
  - $\mathrm{if\_then\_else\_}$.
  - $>>=$.
- Writing of modular programs difficult.
- One solution: Improve the termination checker, or use something like size types.

Direct Solution

```
data IO+ (C : Set) (R : C -> Set) (A : Set) : Set where
  do  : (c : C) -> (f : R c -> IO C R A) -> IO+ C R A

mutual
  IOrec    : {C : Set} -> {R : C -> Set} -> {A B : Set}
             -> (A -> IO+ C R (A + B))
             ->  A -> IO C R B
  ...
  IOrecaux' : {C : Set} -> {R : C -> Set} -> {A B : Set}
              -> (A -> IO+ C R (A + B))
              ->  IO C R (A + B)  -> IO C R B
   ...

  IOrecaux'' : {C : Set} -> {R : C -> Set} -> {A B : Set}
               -> (A -> IO+ C R (A + B))
               ->  IO+ C R (A + B) -> IO C R B
     ...
```

- Instead of defining

```
mutual
  f : A -> IO C R D
  f a = prog1 a' >>= \ x -> if t then f a'' else g b

  g : B -> IO C R D
  g b = prog2  b' >>= if t' then f a else return d
```

which doesn't termination check

- Define prog1, prog2 as returning elements of IO+ and define

```
rec : A    -> IO C R (A  + D)
rec a = return (inl a)

finish: D -> IO C R (A  + D)
finish d = return (inr d)
```

```
mutual
  f' : A -> IO+ C R (A + D)
  f' a = prog1 a' +>>= \ x -> if t then rec a''
                                    else IO+toIO (g b)

  g : B -> IO+ C R (A + D)
  g b = prog2  b' +>>= if t'  then rec a
                              else finish d

  f : A -> IO C R D
  f a = IORec f' a
```

4. A Graphics Library for Agda

- We use the SOE library from Hudak's book "The Haskell school of expression".
  - Rather limited library.
- We import various native Haskell types, e.g.

```
postulate Window : Set
{-# COMPILED_TYPE Window Window #-}

postulate Size : Set
{-# COMPILED_TYPE Size SOE.Size #-}

postulate size : Int -> Int -> Size
{-# COMPILED size (\ x y -> (x,y) :: SOE.Size) #-}
```

```
data Event : Set where
   Key : Char -> Bool -> Event
   Button : Point -> Bool -> Bool -> Event
   MouseMove : Point -> Event
   Resize : GLSize -> Event
   Refresh : Event
   Closed : Event

{-# COMPILED_DATA Event Event Key Button MouseMove Resize R
```

```
postulate nativeMaybeGetWindowEvent : Window
                                        -> nativeIO (Maybe Eve
{-# COMPILED nativeMaybeGetWindowEvent maybeGetWindowEvent

postulate Graphic : Set
{-# COMPILED_TYPE Graphic SOE.Graphic #-}

postulate nativeDrawInWindow : Window -> Graphic
          -> nativeIO Unit
{-# COMPILED nativeDrawInWindow drawInWindow #-}

postulate text : Point -> String -> Graphic
{-# COMPILED text text #-}

postulate nativeOpenWindow : String -> Size -> nativeIO Win
{-# COMPILED nativeOpenWindow openWindow #-}
```

```
data Color : Set where
  black : Color
  blue : Color
  green : Color
  ...
{-# COMPILED_DATA Color SOE.Color SOE.Black SOE.Blue SOE.Gr

postulate withColor : Color -> Graphic -> Graphic
{-# COMPILED withColor withColor #-}

postulate polygon : List Point -> Graphic
{-# COMPILED polygon polygon #-}

postulate text1 : Point -> String -> Graphic
{-# COMPILED text1 text #-}
```

```
data GraphicsCommands : Set where
  maybeGetWindowEvent : Window               -> GraphicsComman
  drawInWindow        : Window -> Graphic -> GraphicsComman
  openWindow          : String -> Size    -> GraphicsComman
  timeGetTime         :                      GraphicsComman

GraphicsResponses : GraphicsCommands -> Set
GraphicsResponses (maybeGetWindowEvent w) = Maybe Event
GraphicsResponses (drawInWindow w g)      = Unit
GraphicsResponses (openWindow s s')       = Window
GraphicsResponses timeGetTime             = Word32
```

```
IOGraphics : Set -> Set
IOGraphics = IO GraphicsCommands GraphicsResponses


translateIOGraphicsLocal : (c : GraphicsCommands) -> native
translateIOGraphicsLocal (maybeGetWindowEvent w)
         = nativeMaybeGetWindowEvent w
translateIOGraphicsLocal (drawInWindow w g)
         = nativeDrawInWindow w g
...

translateIOGraphics : {A : Set} -> IOGraphics A -> nativeIO
translateIOGraphics = translateGeneric
                      translateIOGraphicsLocal
```

More Code

- Look at IOExperimentRecursion.agda.

Other Agda Work in Swansea

- Combining SAT solver in Agda
  - Implementation of a simple SAT solver in Agda.
  - Proof

$$
\begin{aligned}
&(\varphi : \mathrm{For}) \\
&\to \mathrm{Check}\ \varphi \\
&\to (b : \mathrm{Vec\ Bool}\ (\mathrm{numberVars}\ \varphi)) \\
&\to \mathrm{T}\ (b \models \varphi)
\end{aligned}
$$

  - Allows to proof formulas such as

$$\mathrm{T}((s \wedge_{\mathrm{Bool}} t) \vee_{\mathrm{Bool}} (\neg_{\mathrm{Bool}}\ s) \vee_{\mathrm{Bool}} (\neg_{\mathrm{Bool}}\ t))$$

    for any $s, t : \mathrm{Bool}$.
  - $\mathrm{check} : \mathrm{For} \to \mathrm{Bool}$ replaced by a BUILTIN SAT solver in Agda. (Plugin).

Other Agda Work in Swansea

- Extraction of programs from proofs about real numbers with axioms.
- Experiments with specificying railways in Agda.

Conclusion

- Writing proper interactive programs in Agda is feasible.
- We gain that
  - programs are guaranteed to stay interactive
  - we ihave a flexible IO type which can be adapted to different interactive scenarios
  - IO programs are elements of a proper Agda codata type, which can be transformed and reasoned about.

Future Work

- How to reason about interactive programs.
    - Theoretically clear. How to do it practically?
- With GUIs one would like to associate server side programs. How to do this?
- Dealing with threads, pointers.
- Dealing with Functional Reactive Programming.