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

Program

Response

World

Command
Interface in Agda

- Interface for interactive program given by
  - A set of commands the program can issue
    \[ C : \text{Set} \]
  - A set of responses, depending on commands
    \[ R : C \rightarrow \text{Set} \]
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 \( A : \text{Set} \).

We need to allow non-terminating programs. Therefore the type needs to be defined coinductively.
IO Monad in Agda

codata IO (C : Set) (R : C → Set) (A : Set) : Set where
  do : (c : C) → (f : R c → IO C R A) → IO C R A
  return : (a : A) → IO C R A
Monad Operations

1. $\eta := \text{return}.$

2. $\gg=\gg$ can be defined:

   
   _ $\gg=\gg$ _ : \{C : \text{Set}\} \rightarrow \{R : C \rightarrow \text{Set}\} \rightarrow \{A B : \text{Set}\} \rightarrow \text{IO} C R A

   \rightarrow (A \rightarrow \text{IO} C R B)

   \rightarrow \text{IO} C R B

   do c f $\gg=\gg q = do c (\lambda x \rightarrow f x \gg=\gg q)$

   return a $\gg=\gg q = q a$
There is one uniform IO type in Haskell. We call its translated version

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

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

\[ \text{operation} : A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow \text{IO}B \]

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

\[ \text{operationC} : A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow C \]

\( R : C \rightarrow \text{Set} \) is defined by case distinction, e.g.

\[ R (\text{operationC} \ a_1 \ \cdots \ a_n) = B \]
Example

\[
\text{postulate}
\]

\[
\begin{align*}
\text{nativePutStrLn} & : \text{String} \to \text{nativeIO Unit} \\
\text{nativeGetLine} & : \text{nativeIO String}
\end{align*}
\]

\{
{-# COMPILED nativePutStrLn putStrLn #-}
\}

\{
{-# COMPILED nativeGetLine getLine #-}
\}
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

\[
\text{translateLocal} : (c : C) \rightarrow \text{nativeIO} \ (R \ c)
\]
Example

\[
\text{translateIOConsoleLocal} : (c : \text{ConsoleCommands}) \\
\quad \to \text{nativeIO} (\text{ConsoleResponses} c) \\
\text{translateIOConsoleLocal} \ (\text{putStrLn} \ s) = \text{nativePutStrLn} \ s \\
\text{translateIOConsoleLocal} \ \text{getLine} \quad = \quad \text{nativeGetLine}
\]
Generic Translation

\[
\text{translateGeneric : } \\
\quad \text{forall } \{A \ C \ R\} \\
\quad \rightarrow (\text{translateLocal} : (c : C) \rightarrow \text{nativeIO} (R \ c)) \\
\quad \rightarrow \text{IO} \ C \ R \ A \\
\quad \rightarrow \text{nativeIO} \ A \\
\text{translateGeneric translateLocal (do c f) = } \\
\quad (\text{translateLocal} c) \ \text{native} >>= \\
\quad (\ \backslash \ r \ \\
\quad \rightarrow \text{translateGeneric translateLocal} (f \ r)) \\
\text{translateGeneric translateLocal (return a) = } \\
\quad \text{nativeReturn} \ a
\]
Execution

An interactive program can now be executed by defining an element \texttt{main : nativeIO A}.
Example

\[
\begin{align*}
\text{myProgram} & : \text{IOConsole Unit} \\
\text{myProgram} & = \text{do getLine } (\lambda \text{ line } \rightarrow ( \\
& \quad \text{do (putStrLn line) } (\lambda \_ \rightarrow ( \\
& \quad \quad \text{do (putStrLn line) } (\lambda \_ \rightarrow \\
& \quad \quad \quad \text{myProgram})))))) \\
\text{main} & : \text{nativeIO Unit} \\
\text{main} & = \text{translateIOConsole myProgram}
\end{align*}
\]
Termination Checker

- The translation from IO to nativeIO doesn’t termination check.

- The definition of a specific element of IO \( C R \) termination checks, if defined by guarded recursion.
  - IO, \( >>= \), 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 $\text{IO CR A}$ we are restricted to a sequence of constructors.
- Especially we are not allowed to use $\text{if\_then\_else\_}$.
- $\ggg$.

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

Anton Setzer: Interactive programs in dependent type theory
Instead of defining

mutual

\[ f : A \rightarrow \text{IO} C R D \]
\[ f \ a = \text{prog1} \ a' \ >>= \ \lambda \ x \rightarrow \text{if} \ t \ \text{then} \ f \ a'' \ \text{else} \ g \ b \]

\[ g : B \rightarrow \text{IO} C R D \]
\[ g \ b = \text{prog2} \ b' \ >>= \ \text{if} \ t' \ \text{then} \ f \ a \ \text{else} \ \text{return} \ d \]

which doesn’t terminate check
Define prog1, prog2 as returning elements of IO+ and define

\[ \text{rec : } A \rightarrow \text{IO C R (A + D)} \]
\[ \text{rec } a = \text{return (inl a)} \]

\[ \text{finish : } D \rightarrow \text{IO C R (A + D)} \]
\[ \text{finish } d = \text{return (inr d)} \]
mutual

\[ f' : A \to IO^+ \ C \ R \ (A + D) \]
\[ f' \ a = \text{prog1} \ a' +>>= \ \lambda \ x \to \begin{cases} 
  \text{if } t \text{ then rec } a'' \\
  \text{else } IO^+\text{toIO} \ (g \ b)
\end{cases} \]

\[ g : B \to IO^+ \ C \ R \ (A + D) \]
\[ g \ b = \text{prog2} \ b' +>>= \begin{cases} 
  \text{if } t' \text{ then rec } a \\
  \text{else } \text{finish } d
\end{cases} \]

\[ f : A \to IO \ C \ R \ D \]
\[ f \ a = \text{IORec} \ f' \ a \]
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.

```haskell
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 Refresh #}
postulate nativeMaybeGetWindowEvent : Window
    -> nativeIO (Maybe Event)
{-# 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 Window
{-# COMPILED nativeOpenWindow openWindow #-}
data Color : Set where
    black : Color
    blue : Color
    green : Color
...
{-# COMPILED_DATA Color SOE.Color SOE.Black SOE.Blue SOE.Green

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

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

postulate text1 : Point -> String -> Graphic
{-# COMPILED text1 text1 text #-}
data GraphicsCommands : Set where
  maybeGetWindowEvent : Window \rightarrow GraphicsCommands
  drawInWindow : Window \rightarrow Graphic \rightarrow GraphicsCommands
  openWindow : String \rightarrow Size \rightarrow GraphicsCommands
  timeGetTime : GraphicsCommands

GraphicsResponses : GraphicsCommands \rightarrow 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) → nativeIO
translateIOGraphicsLocal (maybeGetWindowEvent w)
    = nativeMaybeGetWindowEvent w
translateIOGraphicsLocal (drawInWindow w g)
    = nativeDrawInWindow w g
...

translateIOGraphics : {A : Set} → IOGraphics A → nativeIO A
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

\[(\varphi : \text{For}) \rightarrow \text{Check } \varphi \rightarrow (b : \text{Vec Bool } (\text{numberV ars } \varphi)) \rightarrow T (b \models \varphi)\]

Allows to proof formulas such as

\[T((s \land_{\text{Bool}} t) \lor_{\text{Bool}} (\neg_{\text{Bool}} s) \lor_{\text{Bool}} (\neg_{\text{Bool}} t))\]

for any \(s, t : \text{Bool}\).

check : For \(\rightarrow\) 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 specifying railways in Agda.
Conclusion

Writing proper interactive programs in Agda is feasible.

We gain that

- programs are guaranteed to stay interactive
- we have 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.