Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Lecture A3-a: Basic IO

Doaitse Swierstra

Utrecht University

August 18, 2010

Input and Output

Thus far:

- a program contained a collection of function and type definitions
- we compute a value by typing its expression in GHCi

Input and Output

Thus far:

- a program contained a collection of function and type definitions
- we compute a value by typing its expression in GHCi

We want to be able to write real programs, such as

- Quake
- Haskell compilers
- Graphical editors
- Tax programs
- Web servers, Wiki systems
- ...

What is the problem?

Haskell offers *referential transparency*, i.e. we can always replace a variable by the right-hand side of its definition and vice versa, so

What is the problem?

Haskell offers *referential transparency*, i.e. we can always replace a variable by the right-hand side of its definition and vice versa, so

> **let** $x = <expr>$ **in** $\ldots x \ldots x \ldots$

should always have the same meaning as:

> $\ldots <expr> \ldots <expr> \ldots$

What is the problem?

Haskell offers *referential transparency*, i.e. we can always replace a variable by the right-hand side of its definition and vice versa, so

$$\textbf{let } x = <expr> \textbf{ in } \ldots x \ldots x \ldots$$

should always have the same meaning as:

$$\ldots < expr > \ldots < expr > \ldots$$

And also:

$$(\lambda x \rightarrow < expr1 >) < expr2 >$$

should always be equivalent to:

$$\textbf{let } x = <expr2> \textbf{ in } < expr1 >$$

Side-effects ;-{{

Now suppose we have a Haskell function *random* which delivers a different value for each call –so uses the concept of a state to avoid returning the same value over and over–

Side-effects ;-{{

Now suppose we have a Haskell function *random* which delivers a different value for each call –so uses the concept of a state to avoid returning the same value over and over– then we require that:

> $\textbf{let}\ x = random\ \textbf{in}\ x \equiv x$

and

> $random \equiv random$

should mean the same; which is not what we expect.

Side-effects ;-{{

The same holds for reading a character from input:

$$\textbf{let } x = \mathit{getChar} \textbf{ in } x \equiv x$$

en

$$\mathit{getChar} \equiv \mathit{getChar}$$

should mean the same, but it is hard to see how to achieve this.

Commands

This problem is solved by introducing a special type $IO\ _$:

$IO\ _$

Values of this type

$$IO\ ()$$

represent a sequence of I/O commands.

Commands

This problem is solved by introducing a special type $IO\ _$:

$IO\ _$

Values of this type

$$IO\ ()$$

represent a sequence of I/O commands.

A "normal" Haskell program contains a single function $main :: IO\ ()$.

Commands

This problem is solved by introducing a special type $IO\ _$:

$IO\ _$

Values of this type

$$IO\ ()$$

represent a sequence of I/O commands.

A "normal" Haskell program contains a single function $main :: IO\ ()$. This is the function which is called when the program runs; it returns a sequence of IO commands which are executed as soon as they are returned.

putChar

Consider:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO Empty
Prelude> putChar 'a'
a
```

putChar

Consider:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO Empty
Prelude> putChar 'a'
a
```

The differences:

1. the first value `'a'` is not of type $IO\ ()$; GHCi implicitly adds printing of the result

putChar

Consider:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO Empty
Prelude> putChar 'a'
a
```

The differences:

1. the first value `'a'` is not of type $IO\ ()$; GHCi implicitly adds printing of the result

putChar

Consider:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO Empty
Prelude> putChar 'a'
a
```

The differences:

1. the first value `'a'` is not of type $IO\ ()$; GHCi implicitly adds printing of the result
2. the second value is of type $IO\ ()$; the sequence of actions is executed

Composing actions into larger actions

We can compose actions (think of concatenation) with the **do** construct:

```
twoLetters :: IO ()
twoLetters = do putChar 'H'
                putChar 'i'
```

Composing actions into larger actions

We can compose actions (think of concatenation) with the **do** construct:

```
twoLetters :: IO ()
twoLetters = do putChar 'H'
                putChar 'i'
```

Indentation is important! Note that **do** is yet another instance of syntactic sugar.

Recursive actions

Most functions taking a list are recursive; when returning $IO\,..$ we can do the same:

```
putStr :: String→IO ()
putStr (c : cs) = do putChar c
                     putStr   cs
```

Recursive actions

putStr

```
putStr :: String→IO ()
putStr (c : cs) = do putChar c
                     putStr   cs
putStr []       = return ()
```

Here *return* () corresponds to the empty sequence of actions. We will come back to its parameter.

return

```
return :: a→IO a
```

Passing values

Now we know how to compose actions the question arises:

> *How can we use results of earlier actions in the construction of later actions?*

Passing values

Now we know how to compose actions the question arises:

> *How can we use results of earlier actions in the construction of later actions?*

Example:

$$
\begin{aligned}
&echo :: IO\ () \\
&echo = \textbf{do}\ v \leftarrow getChar \\
&\qquad\qquad putChar\ v
\end{aligned}
$$

Passing values

Now we know how to compose actions the question arises:

> *How can we use results of earlier actions in the construction of later actions?*

Example:

$$
\begin{aligned}
&echo :: IO\ () \\
&echo = \textbf{do}\ v \leftarrow getChar \\
&\qquad\qquad putChar\ v
\end{aligned}
$$

```
Test> :t getChar
getChar :: IO Char
Test> echo
a
a
Test>
```

Question ...

What do you think is the result of:

```
twoLetters' :: IO ()
twoLetters' = do putChar 'H'
                 return   "does nothing"
                 putChar 'i'
```

Question ...

What do you think is the result of:

```
twoLetters' :: IO ()
twoLetters' = do putChar 'H'
                 return   "does nothing"
                 putChar 'i'
```

```
Test> twoLetters'
Hi
Test>
```

Question ...

What do you think is the result of:

$$
\begin{aligned}
&twoLetters' :: IO\ () \\
&twoLetters' = \textbf{do}\ putChar\ \texttt{'H'} \\
&\qquad\qquad\quad\ \ return\ \ \texttt{"does nothing"} \\
&\qquad\qquad\quad\ \ putChar\ \texttt{'i'}
\end{aligned}
$$

```
Test> twoLetters'
Hi
Test>
```

So *return* gets an argument, which is available for the rest of the computation, but is not used here.

Compare this with ...

```
twoLetters'' :: IO ()
twoLetters'' = do putChar 'H'
                  text <- return ", FP students"
                  putChar 'i'
                  putStr  text
```

Compare this with ...

$$
\begin{aligned}
&twoLetters'' :: IO\ () \\
&twoLetters'' = \textbf{do}\ putChar\ \texttt{'H'} \\
&\qquad\qquad\qquad\quad text \leftarrow return\ \texttt{", FP students"} \\
&\qquad\qquad\qquad\quad putChar\ \texttt{'i'} \\
&\qquad\qquad\qquad\quad putStr\ \ text
\end{aligned}
$$

```
Test> twoLetters''
Hi, FP students
Test>
```

De underlying notation

$$
\begin{aligned}
&\mathit{twoLetters''} :: \mathit{IO}\ () \\
&\mathit{twoLetters''} = \textbf{do}\ _ \leftarrow \mathit{putChar}\ \texttt{'H'} \\
&\qquad\qquad\quad \mathit{text} \leftarrow \mathit{return}\ \texttt{", FP students"} \\
&\qquad\qquad\quad _ \leftarrow \mathit{putChar}\ \texttt{'i'} \\
&\qquad\qquad\quad _ \leftarrow \mathit{putStr}\ \ \mathit{text}
\end{aligned}
$$

Most actions return $()$), which we do not bind and thus are discarded.

Values of type $IO\ a$ are first class values!

```
printLater =
  do _        ← putChar 'H'
     pr_text  ← return   (putStr ", FP students")
     _        ← putChar 'i'
     _        ← pr_text
```

Values of type $IO\ a$ are first class values!

$$
\begin{array}{l}
printLater = \\
\quad \textbf{do}\ _ \leftarrow putChar\ \texttt{'H'} \\
\qquad pr_text \leftarrow return\ \ (putStr\ \texttt{", FP students"}) \\
\qquad _ \leftarrow putChar\ \texttt{'i'} \\
\qquad _ \leftarrow pr_text
\end{array}
$$

The types are:

- $return :: \forall\ a.a \rightarrow IO\ a$

Values of type $IO\ a$ are first class values!

```
printLater =
  do _        ← putChar 'H'
     pr_text  ← return   (putStr ", FP students")
     _        ← putChar 'i'
     _        ← pr_text
```

The types are:

- $return :: \forall\ a.a \rightarrow IO\ a$
- $putStr$ `", FP students"` $:: IO\ ()$

Values of type $IO\ a$ are first class values!

```
printLater =
  do _        ← putChar 'H'
     pr_text  ← return   (putStr ", FP students")
     _        ← putChar 'i'
     _        ← pr_text
```

The types are:

- $return :: \forall\ a.a \rightarrow IO\ a$
- $putStr$ `", FP students"` $:: IO\ ()$
- $return\ (putStr$ `", FP students"`$) :: IO\ (\cdot)\ (IO\ ())$

Values of type $IO\ a$ are first class values!

$$
\begin{array}{l}
printLater = \\
\quad \textbf{do}\ _ \leftarrow putChar\ \texttt{'H'} \\
\qquad pr_text \leftarrow return\ \ (putStr\ \texttt{", FP students"}) \\
\qquad _ \leftarrow putChar\ \texttt{'i'} \\
\qquad _ \leftarrow pr_text
\end{array}
$$

The types are:

- $return :: \forall\ a.a \rightarrow IO\ a$
- $putStr\ \texttt{", FP students"} :: IO\ ()$
- $return\ (putStr\ \texttt{", FP students"}) :: IO\ (\cdot)\ (IO\ ())$
- $pr_text :: IO\ ()$

getLine

The function $getLine :: IO\ String$ reads a whole line:

```
greeting :: IO ()
greeting = do putStr "What is your name? "
              name ← getLine
              putStrLn ("Hello, " ++ name)
```

getLine

The function $getLine :: IO\ String$ reads a whole line:

$$
\begin{aligned}
&greeting :: IO\ () \\
&greeting = \textbf{do}\ putStr\ \texttt{"What is your name? "} \\
&\qquad\qquad\quad name \leftarrow getLine \\
&\qquad\qquad\quad putStrLn\ (\texttt{"Hello, "} +\!\!+\ name)
\end{aligned}
$$

```
Test> greeting
What is your name? Doaitse
Hello, Doaitse
Test>
```

The function *putStrLn* acts just as *putStr*, but adds an extra `'\n'` at the end.

Simple IO from the Haskell prelude (1)

```
print :: Show a ⇒ a→IO ()
print e = putStrLn (show e)

getLine :: IO String
getLine = do v ← getChar
             if v ≡ '\n' then    return ""
                         else do vs ← getLine
                                 return (v : vs)
```

Simple IO from the Haskell prelude (1)

```
print :: Show a ⇒ a→IO ()
print e = putStrLn (show e)

getLine :: IO String
getLine = do v ← getChar
             if v ≡ '\n' then    return ""
                         else do vs ← getLine
                                 return (v : vs)
```

Attention: both branches of the **if** ... **then** ... **else** ... return a value of type *IO* ....

Simple IO from the prelude (2)

The basic IO actions

$$
\begin{array}{lll}
return & :: & a \to IO\ a \\
putChar & :: & Char \to IO\ () \\
putStr & :: & String \to IO\ () \\
putStrLn & :: & String \to IO\ () \\
print & :: & Show\ a \Rightarrow a \to IO\ () \\
getLine & :: & IO\ String \\
writeFile & :: & String \to String \to IO\ () \\
readFile & :: & String \to IO\ String
\end{array}
$$

We define our "own control structures"

Of course we can now abstract from common control structures

sequence

$$sequence :: [IO\ a] \rightarrow IO\ [a]$$

We define our "own control structures"

Of course we can now abstract from common control structures

sequence

$$
\begin{aligned}
&\mathit{sequence} :: [\mathit{IO}\ a] \to \mathit{IO}\ [a] \\
&\mathit{sequence}\ [\,] \qquad\quad = \mathit{return}\ [\,] \\
&\mathit{sequence}\ (v : vs) = \mathbf{do}\ x \leftarrow v \\
&\qquad\qquad\qquad\qquad\quad\ \ xs \leftarrow \mathit{sequence}\ vs \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (x : xs)
\end{aligned}
$$

We define our "own control structures"

Of course we can now abstract from common control structures

sequence

$$
\begin{aligned}
&\mathit{sequence} :: [\mathit{IO}\ a] \to \mathit{IO}\ [a] \\
&\mathit{sequence}\ [\,] \qquad\quad = \mathit{return}\ [\,] \\
&\mathit{sequence}\ (v : vs) = \mathbf{do}\ x \leftarrow v \\
&\qquad\qquad\qquad\qquad\quad\ \ xs \leftarrow \mathit{sequence}\ vs \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (x : xs)
\end{aligned}
$$

The function *sequence* takes a list of $\mathit{IO}\ a$ actions, each returning a value of type a, and constructs one single action of of these, which returns a value of type $[a]$. The result thus is the list of all the individual results.

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_  []       = return (_)
sequence_  (v : vs) = do v
                         sequence_ vs
```

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_  []       = return (_)
sequence_  (v : vs) = do v
                         sequence_ vs
```

We now can write:

```
putStr str = sequence_ (map putChar str)
```

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_  []       = return (_)
sequence_  (v : vs) = do v
                         sequence_ vs
```

We now can write:

```
putStr str = sequence_ (map putChar str)
```

Or even shorter:

```
putStr = sequence_ ∘ map putChar
```

An example of a "real program"

We write a "number guessing" game::

```
Test> main
Think of a number less than 100.
Is it 50? (l =larger, s = smaller, c = correct)
l
Is it 75? (l =larger, s = smaller, c = correct)
s
Is it 62? (l =larger, s = smaller, c = correct)
l
Is it 68? (l =larger, s = smaller, c = correct)
s
Is it 65? (l =larger, s = smaller, c = correct)
c
Guessed!
Test>
```

main

```
main = do
  putStrLn "Think of a number less than 100."
  guess 1 100
```

The function *guess*

```
guess :: Int→Int→IO ()
guess low high =
  do putStrLn
       ("Is it " ++ showInt middle
       ++ "? (l =larger, s = smaller, c = correct)"
       )
     answer ← getLine
     processAnswer answer
  where ...
```

De function *guess*

```
where ...
  processAnswer :: String→IO ()
  processAnswer "l" = guess (middle + 1) high
  processAnswer "s" = guess low (middle − 1)
  processAnswer "c" = putStrLn "Guessed!"
  processAnswer _   = guess low high

  middle :: Int
  middle = (low + high) `div` 2
```

De function *guess*

```
where ...
  processAnswer :: String→IO ()
  processAnswer "l" = guess (middle + 1) high
  processAnswer "s" = guess low (middle − 1)
  processAnswer "c" = putStrLn "Guessed!"
  processAnswer _   = guess low high

  middle :: Int
  middle = (low + high) `div` 2
```

Note how *main* passes control to *guess*, *guess* to *processAnswer*, en *processAnswer* if needed to a new instance of *guess*.

Typical structure of Haskell programs

- The function *main* is of type $IO\ ()$, en is the starting point of the program

Typical structure of Haskell programs

- The function *main* is of type $IO\ ()$, en is the starting point of the program
- At top level we have sequential execution, in which functions pass control, and together construct a value of type $IO\ ()$.

Typical structure of Haskell programs

- The function *main* is of type $IO\ ()$, en is the starting point of the program
- At top level we have sequential execution, in which functions pass control, and together construct a value of type $IO\ ()$.
- This "passing on makes that the IO actions are handled sequentially.

Typical structure of Haskell programs

- The function *main* is of type $IO\ ()$, en is the starting point of the program
- At top level we have sequential execution, in which functions pass control, and together construct a value of type $IO\ ()$.
- This "passing on makes that the IO actions are handled sequentially.
- Lazy evaluation takes care of the fact that not all IO actions have to be actually computed before the first one can be executed!

Exercise

Try to write function:

$$poseQuestions :: [String] \rightarrow IO\ [Bool]$$

which takes a list of questions and returns a list of answers.

Exercise

Try to write function:

$$poseQuestions :: [String] \rightarrow IO\ [Bool]$$

which takes a list of questions and returns a list of answers.

```
Test> demo ["Is is Sunday?", "Is your name Doaitse?"]
Is is Sunday?
(y) or (n)?
n
Is your name Doaitse?
(y) or (n)?
y
[False, True]
Test>
```

Solution

$poseQuestions :: [String] \rightarrow IO\ [Bool]$

Solution

$$poseQuestions :: [String] \rightarrow IO\ [Bool]$$

$$poseQuestions\ qs = sequence\ (map\ poseQuestion\ qs)$$

$$\begin{aligned} demo\ qs = \textbf{do}\ & answers \leftarrow poseQuestions\ qs \\ & print\ answers \end{aligned}$$

Solution

$$poseQuestions :: [String] \rightarrow IO\ [Bool]$$

$$poseQuestions\ qs = sequence\ (map\ poseQuestion\ qs)$$

$$\begin{aligned} demo\ qs = \textbf{do}\ & answers \leftarrow poseQuestions\ qs \\ & print\ answers \end{aligned}$$

$$\begin{aligned} poseQuestion\ q = \textbf{do}\ & putStrLn\ q \\ & putStrLn\ \texttt{"(y) or (n)?"} \\ & (v\colon _) \leftarrow getLine \\ & \textbf{if}\ v \equiv \texttt{'y'} \\ & \quad \textbf{then}\ return\ True \\ & \quad \textbf{else if}\ v \equiv \texttt{'n'}\ \textbf{then}\ return\ False \\ & \qquad\qquad\qquad \textbf{else}\ \ poseQuestion\ q \end{aligned}$$

Showing part of the machinery ...

The **do**-notation actually is syntactic sugar for a sequence of calls to the function $\gg\!=$:

> **do** $v \leftarrow <expr1>$
> $\quad <expr2>$

actually stands for:

> $<expr1> \gg\!= \lambda v \rightarrow <expr2>$

Showing part of the machinery ...

The **do**-notation actually is syntactic sugar for a sequence of calls to the function $\gg\!=$:

> **do** $v \leftarrow <expr1>$
> $\quad <expr2>$

actually stands for:

> $<expr1> \gg\!= \lambda v \rightarrow <expr2>$

Question: What do you think is the type of $\gg\!=$?

Showing part of the machinery ...

The **do**-notation actually is syntactic sugar for a sequence of calls to the function $\gg\!=$:

$$\begin{array}{l} \textbf{do}\ v \leftarrow <expr1> \\ \quad <expr2> \end{array}$$

actually stands for:

$$<expr1> \gg\!= \lambda v \rightarrow <expr2>$$

Question: What do you think is the type of $\gg\!=$?
Answer:

$$(\gg\!=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$$

Actually it is even more general than that

Asking GHCi:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```