Monadic Input/Output

Luca Abeni

luca.abeni@santannapisa.it

Lazy Evaluation and I/O

- Lazy evaluation: expressions are evaluated "only when needed"...
- Consequence: the evaluation order is often undefined
  - Think about "`g(f_1(x), f_2(x))`"...
  - A lazy language does not specify if "`f_1(x)`" is evaluated before "`f_2(x)`" or after it
- What happens with impure functions that do I/O?
  - Example: "`g(hello(), world())`", where "`hello()`" prints "`hello`" and "`world()`" prints "`world!`"
  - What is printed on the screen???

The Core of the I/O Issues

- Every time we do I/O, we need to impose an ordering between functions...
- ...Otherwise, the program output is not deterministic!

- However, imposing an order in functions execution is against the lazy execution idea
- How to address this issue?
  - I/O cannot be performed in functions!
  - So, who performs the I/O???
- In a lazy language, all functions have to be pure!!!
- But to be useful a program needs to perform some I/O!!!

Functions and Actions

- Impure code (I/O and similar) has to be confined in specific components
  - Remember? We used the `main()` function in functional C/C++ programs...
  - Up to now, we are using the `ghci` REPL in Haskell
- Functions implement the core of the program, and are pure
- Actions (or effects) encode the "dirty work" (impure) and are executed by some "non functional engine"
- There is a *strict* distinction between these two things

Input/Output as a Value

- Algebraic data type “`IO a`”
  - It is a parametric data types
  - Depends on the type variable “`a`”
- Represents an I/O “action” (or “effect”) to be executed by the non-functional runtime
- A value of type “`IO a`” (often called “action”; also known as “computation” or “effect”) has two aspects:
  - Represents an “action” that, when executed, can perform I/O
  - Contains a “regular value” of type “`a`” (the value actually returned by the I/O operation!)

I/O Actions: Example

- “Something” that reads a character from the keyboard and returns it...
- ...Cannot be a function (it has side effects!)
- So, what is it? An I/O action: a value of type “`IO Char`”
  - `IO Char` because it returns a character (type “`Char`”)

```
getChar :: IO Char
```

I/O Actions: Another Example

- How to print a character to the screen? Not with a function (side effects are needed!)
- The character is printed by a specific I/O action
  - The type of the I/O action looks strange, because it is not associated to any returned value...
  - Remember the unit type? This is its purpose! So, the type of the I/O action is "`IO ()`"
- The I/O action is generated based on the character to be printed...
- So, we have a function that given the character produces an I/O action: "`Char -> IO ()`"

```
putChar :: Char -> IO ()
```

Again on I/O Actions

- I/O actions look like a smart trick to hide side effects
  - `getChar` is not a function, but a value encoding a side effect
  - The `putChar` function does not have any side effect (does not perform any I/O), but returns a value that encodes side effects!
- So, all functions are still pure, and the *side effects are all in some kind of runtime that executes the I/O actions*
  - Again, compare with a functional program in C++: side effects can be isolated in the `main()` function, leaving the rest of the program purely functional

Combining I/O Actions

- So, what's special in using an "`IO a`" datatype to encode I/O actions?
  - Let's look at how I/O actions can be combined!
- To have a deterministic output, I/O actions must be executed using an eager evaluation order
  - Or a well-defined order anyway
- This is OK, because actions are not functions...
  - ...So, there is no need to lazily evaluate/execute them!
  - Lazyness is only for functions, not for actions!
- So, we need some kind of operator to combine I/O actions

Combining I/O Actions: the Issue

- Assume we need to read a character and then print it on the screen
  - Something like the imperative

    ```
    char c = getchar();
    putchar(c);
    ```

- How can we do this in a functional way?
  - We need something like "`putChar(getChar())`"...
  - If "`getChar`" has type "`IO Char`" and "`putChar`" has type "`Char -> IO ()`"...
  - ...We end up with "`putChar getChar`", which does not typecheck!!!

The Issue — Again

- "`putChar getChar`" is not possible because "`putChar`" wants a "`Char`", but "`getChar`" is a "`IO Char`"!!!
  - Here, the type system is really saving us...
  - ..."`getChar`" **does not return a character**! Its "execution" actually returns a character...
  - So, passing "`getChar`" as an argument to "`putChar`" **is really wrong**!
- We need a way to *force the execution of the "`getChar`" I/O action* and pass the result to "`putChar`"
- In other words, we need an operator/function that "extracts" the value of type "Char" from "`getChar`"
- Spoiler: this function is named "*bind*

Here Come the Monads

- Instead of inventing random functions/operators, let's look at some theory...
- Monad: very scary name (exercize: just try to search for "monad" on your favourite search engine)
  - We can find monads in philosophy (for example, see Leibniz, ...), mathematics (hyper-real analysis, category theory, ...), computer science, science fiction, ...
- So, what is a monad??? Can be a lot of things
  - Even a burrito...
- Here, let's not look at all the complex theoretical details...
- ...Let's just consider what's important in this context!

Why Monads?

- Why talking about monads, here???
  - Because they can provide what we need for combining I/O actions
  - Actually, they can provide much more (option types, computations with a state, exceptions, ...)
- The "relevant monads" for us are the monads from computer science (related to category theory...)
  - Informally, a monad is a type derived from type α associated to two functions: *bind* and *return*
  - The *bind* and *return* functions must provide some important properties
  - Category theory discusses these properties and their consequences

Practical Monads

- Monad: algebraic data type "`M a`"
  - Parametric type dependent on type variable "`a`" (type α in type theory)
- Two functions "`bind`" and "`return`" must exist.
  - return has domain "`a`" and codomain "`M a`"
  - bind is more complex
    - it is a curryified function: has domain "`M a`" and codomain the set of functions from "`a -> M b`" to "`M b`"
- Using the Haskell syntax:
  - **return** :: a –> M a
  - bind :: M a –> (a –> M b) –> M b

Practical Monads: Informal Interpretation

- "return" transforms a value of type "`a`" into a monadic value of type "`M a`"
- "bind" allows to apply a function "`a -> M b`" to a monadic value "`M a`"
  - It must somehow extract the "`a`" value from the monad, and apply the function to it!
  - "`M a -> (a -> M b) -> M b`" can be seen as a function with two arguments of type "`M a`" and "`a -> M b`" and a result of type "`M b`"
- A type "`M a`" with these 2 functions is a monad if 3 properties hold
  - Basically equivalent to commutative and additive properties

The I/O Monad

- I/O monad: "`IO a`"
  - The "`bind`" function performs the action encoded by "`IO a`, then extracts "a" from this value and passes it to the function received as a second argument
    - It returns a second I/O action!
  - The "`return`" function just encapsulates a value in an I/O action (that does not actually perform any input or output)
- In Haskell, "`bind`" is the "`>>=`" operator

I/O Monad Example

- Let's see the I/O monad in action... In Haskell,

  getChar >>= **putChar**

- Executes the "`getChar`" action (of type "`IO Char`")
- Then, extracts the "`Char`" value from it...
- ...And passes such a value to "`putChar`" (a function "`Char -> IO ()`" that, given the character, returns an "`IO ()`" value)
- When the action returned by "`putChar`" is executed, the character is printed to the screen!!!
- So, this allows to easily combine I/O actions
- The whole complex monads theory from category theory just makes sure that the actions' combination is sound!

Haskell: I/O Serialization

- In Haskell, "`getChar >>= putChar`" evaluates to an I/O action that reads a character and prints is
- Now, let's try to read a character and print it twice

```
getChar >>= (\c -> (putChar c >>= (\x -> putChar c)))
```

- The second bind looks funny
  - The return value of "`putChar c`" is quite useless (it is of type "`IO ()`")...
  - ..."\x −> **putChar** c" discards the "`()`" value!
- The input of the second λ is only needed to serialize the output!!!
  - This is a strong sign that something is impure...
- This trick is often used to serialize I/O... Haskell even has a shortcut for this expression!

Haskell: I/O Serialization

- "a >>= (\x -> b)" can be written "a >> b"
- An action that reads a character, prints a CR, and then prints the character twice is:

```
getChar >>= (\c -> (putChar '\n' >>= (\y -> putChar c >>= (\x -> putChar c))))
```

- Haskell also allows to write it as

```
getChar      >>= \c ->
putChar '\n' >>= \y ->
putChar c    >>= \x ->
putChar c
```

or

```
getChar      >>= \c ->
putChar '\n' >>
putChar c    >>
putChar c
```

- It starts to look like an imperative program???

Haskell: More Complex I/O

- We saw that the "bind" function can be used to sequentially compose I/O actions...
- What is "return" used from?
  - We know it can forge monadic values from non-monadic ones
- Example: read some characters and return a single "IO a" value containing all of them:

```
getChar >>= \a ->
getChar >>= \b ->
...         \x ->
return (a, b, ..., x)
```

- The I/O action encoded by "`return a`" does not perform any I/O...

Even More Complex I/O

- Read a line of characters (until CR is pressed)
- This is a more complex example, using recursion:

```
myget = getChar >>= \c ->
          if c == '\n'
            then
              return []
            else
              myget >>= \rest_of_line ->
              return (c : rest_of_line)
```

- Note: "`getLine`" can be used for this...
- ...We opencoded it only as an example!

Syntactic Sugar for Monads

- We know that Haskell wants to look like an imperative language
  - Remember how currying is hidden behind an imperative-like notation?
- Some syntactic sugar can "hide" monads

```
getChar        >>= \c ->
putChar '\n'   >>= \y ->
putChar c      >>= \x ->
putChar c
```

can be written as

```
do {
  c <- getChar;
  putChar '\n';
  putChar c;
  putChar c
}
```

More about the do Notation

- The "do notation" is just a different syntax for the monads' "`bind`" and "`return`"
  - Again: nothing new... Just syntactic sugar!
- Can be transformed into "regular bind and return" as follows:
  - "**do** x <– e; s" → "e >>= \ x–> **do** s"
  - "**do** e; s" → "e >> **do** s"
  - "**do** e" → "e"
- Notice that "<–" hides a lambda abstraction and a bind
  - This can be seen as similar to the creation of a binding
  - "`x <- e`" binds the name "`x`" to value "`e`"

do Notation and Bindings

- In do notation, "`x <- e`" can be seen as a binding
- But this is not an assignment!!!
  - This "binding"" only modifies the environment; there is no store function!
- That is, this is valid:

```
do {
  s <- putStr "What is your name? ";
  s <- getLine;
  return s
}
```

- It will return a value of type "`IO String`"
  - If "`<-`" was an assignement, this was not valid because the type of "`putStr`" is different from the type of "`getLine`"

Haskell Programs

- We know how to bind names to values, how to define functions, how to do I/O...
  - We generally test things in a REPL (example: `ghci`)
- What are we missing to write a self-contained program?
  - The usual `main()` function!
- In C-like imperative languages, the entry point of a program is a function ("`int main(int argc, char *argv[])`", or similar...)
- What about Haskell? Can "`main`" be a function?
  - Uhm... Functions are pure... They do not perform any I/O...

Haskell and `main`

- The "`main`" entry point in Haskell is actually an action!
  - It cannot be a function, because it needs to do some I/O
- In Haskell, actions are encoded as values of the "`IO a`" data type...
  - ...So, `main` is a value of "`IO ...`"... Which type, exactly?
- Since `main` does not return any value, its type is "`IO ()`" (like "`putChar`" and friends)
  - main is *usually* a function...

Complete Example

```
gcd3 a 0          = a
gcd3 a b          = gcd3 b (a `mod` b)

c2i c = (fromEnum c) - (fromEnum '0')

s2i_1 [] res = res
s2i_1 (c:l) res = s2i_1 l ((c2i c) + res * 10)
s2i s = s2i_1 s 0

main = getLine >>= \s1 ->
       getLine >>= \s2 ->
       print (gcd3 (s2i s1) (s2i s2))
```

Note: implementing “`s2i`” is useless (Haskell provides “`read`”)

Complete Example

```
gcd3 a 0          = a
gcd3 a b          = gcd3 b (a `mod` b)

{- Notice: I implemented "s2i", but we could use
  "read" (which is more generic) instead -}
c2i c = (fromEnum c) - (fromEnum '0')
s2i_1 [] res = res
s2i_1 (c:l) res = s2i_1 l ((c2i c) + res * 10)
s2i s = s2i_1 s 0

main = do {
  s1 <- getLine;
  s2 <- getLine;
  print (gcd3 (s2i s1) (s2i s2))
}
```