Haskell and Explicit Effects

Andres Löh

CNRS / ISCPIF, Paris, 2023-04-04 —

Well-Typed

The Haskell Consultants

About me

- PhD (Utrecht University) 2004
- Lecturer at Utrecht University 2007–2010
- Partner at Well-Typed 2010–

About Well-Typed

- Founded 1998.
- Haskell consulting (development, advice, support, training).
- Currently ∼20 people, distributed over the USA, Europe, South Africa and India.
- Clients mainly in Europe and USA (most work done remotely).

Haskell

Haskell

- Originally an attempt to create a standard **lazy** functional programming language.
- First version 1990.
- Most recent standard version still Haskell2010, but . . .
- Main implementation: GHC (Glasgow Haskell Compiler), developed by Simon Peyton Jones and many contributors.
- GHC / Haskell is in continuous development, many language extensions in active use (GHC2021).

Haskell features

Technical:

- easy to define datatypes
- high abstraction level
- strong type system
- separation of effectful and pure computations
- very versatile

Social:

- large helpful community
- culture of solving problems properly
- open-source (BSD) by default
- vast amount of libraries in central repository (Hackage)

Haskell features

Technical:

- easy to define datatypes
- high abstraction level
- strong type system
- **separation of effectful and pure computations**
- very versatile

Social:

- large helpful community
- culture of solving problems properly
- open-source (BSD) by default
- vast amount of libraries in central repository (Hackage)

Most other languages . . .

```
int dbl(int x) {
  return x + x;
}
```

Most other languages . . .

```
int dbl(int x) {
  return x + x;
}
```

```
int dblSpam(int x) {
  sendSpamMails(x);
  return x + x;
}
```

Both functions have the same type!

Haskell

```haskell
dbl :: Int -> Int
dbl x = x + x
```

Haskell

```
dbl :: Int -> Int
dbl x = x + x
```

```
dblSpam :: Int -> IO Int
dblSpam x = do
  sendSpamMails x
  return (x + x)
```

Haskell

```haskell
dbl :: Int -> Int
dbl x = x + x
```

```haskell
dblSpam :: Int -> IO Int
dblSpam x = do
  sendSpamMails x
  return (x + x)
```

The type of `dblSpam` reflects that it is performs side effects.

Laws?

Do you think that

```
x + x
```

should be the same as

```
2 * x
```

?

Laws!

In Haskell, it is!

Laws!

In Haskell, it is!

But if `dblSpam :: Int -> Int`, then how many spam mails would

```
dblSpam + dblSpam
```

and

```
2 * dblSpam
```

send?

Explicit effects

- Side-effecting computations are marked as such in their types.
- Side-effecting computations are distinguished from their results.
- The **absence** of `IO` gives us peace of mind.

More examples

Consider

```
getLine :: IO String
```

(as it is in Haskell) vs.

```
getLine :: String
```

Reduction order should not matter

```haskell
("a" <> "b") <> ("c" <> "d")
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
```

```
"ab" <> ("c" <> "d")
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
```

```
"ab" <> ("c" <> "d")
```

```
"ab" <> "cd"
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
```

```
"ab" <> ("c" <> "d")
```

```
"ab" <> "cd"
```

```
"abcd"
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
```

```
"ab" <> ("c" <> "d")
```

```
"ab" <> "cd"
```

```
"abcd"
```

Or:

```
("a" <> "b") <> ("c" <> "d")
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
"ab" <> ("c" <> "d")
"ab" <> "cd"
"abcd"
```

Or:

```
("a" <> "b") <> ("c" <> "d")
("a" <> "b") <> "cd"
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
"ab" <> ("c" <> "d")
"ab" <> "cd"
"abcd"
```

Or:

```
("a" <> "b") <> ("c" <> "d")
("a" <> "b") <> "cd"
"ab" <> "cd"
```

Reduction order should not matter

```
("a" <> "b") <> ("c" <> "d")
"ab" <> ("c" <> "d")
"ab" <> "cd"
"abcd"
```

Or:

```
("a" <> "b") <> ("c" <> "d")
("a" <> "b") <> "cd"
"ab" <> "cd"
"abcd"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

```
("a" <> "Frodo") <> ("b" <> getLine)
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

```
("a" <> "Frodo") <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> getLine)
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

```
("a" <> "Frodo") <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> "Sam")
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

```
("a" <> "Frodo") <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> "Sam")
```

```
"aFrodo" <> "bSam"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
```

```
("a" <> "Frodo") <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> getLine)
```

```
"aFrodo" <> ("b" <> "Sam")
```

```
"aFrodo" <> "bSam"
```

```
"aFrodobSam"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> getLine) <> ("b" <> "Frodo")
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> getLine) <> ("b" <> "Frodo")
("a" <> getLine) <> "bFrodo"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> getLine) <> ("b" <> "Frodo")
("a" <> getLine) <> "bFrodo"
("a" <> "Sam") <> "bFrodo"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> getLine) <> ("b" <> "Frodo")
("a" <> getLine) <> "bFrodo"
("a" <> "Sam") <> "bFrodo"
"aSam" <> "bFrodo"
```

Reduction order with uncontrolled effects matters

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> "Frodo") <> ("b" <> getLine)
"aFrodo" <> ("b" <> getLine)
"aFrodo" <> ("b" <> "Sam")
"aFrodo" <> "bSam"
"aFrodobSam"
```

```
("a" <> getLine) <> ("b" <> getLine)
("a" <> getLine) <> ("b" <> "Frodo")
("a" <> getLine) <> "bFrodo"
("a" <> "Sam") <> "bFrodo"
"aSam" <> "bFrodo"
"aSambFrodo"
```

Lazy evaluation

```
take 1 (("a" <> "b") <> ("c" <> "d"))
```

reduces to `"a"` .

Lazy evaluation

```
take 1 (("a" <> "b") <> ("c" <> "d"))
```

reduces to `"a"` .

```
take 1 (("a" <> getLine) <> ("b" <> getLine))
```

reduces to `"a"` , but how many lines of input should it read?

Explicit effects

- Side-effecting computations are marked as such in their types.
- Side-effecting computations are distinguished from their results.
- The **absence** of `IO` gives us peace of mind.

Explicit effects

- Side-effecting computations are marked as such in their types.
- Side-effecting computations are distinguished from their results.
- The **absence** of `IO` gives us peace of mind.
- Decouple effects from the order of evaluation.
- Order and number of effects are always explicit.

No escape

There is no[*] function of type

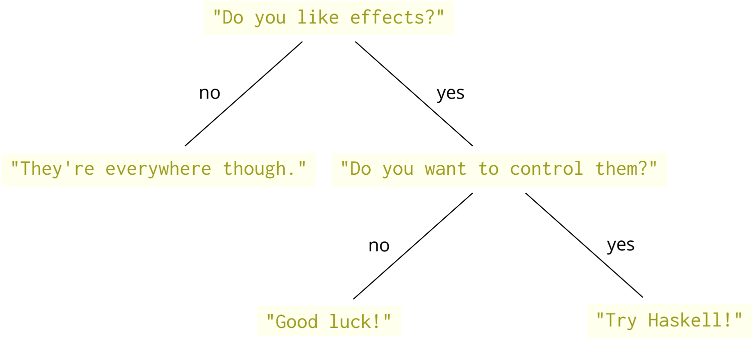
```
IO a -> a
```

because we should not lie!

[*] (None that we speak of.)

Effects everywhere?

Separation of concerns



"Do you like effects?"

no

"They're everywhere though."

yes

"Do you want to control them?"

no

"Good luck!"

yes

"Try Haskell!"

A datatype for dialogues

```haskell
data Dialogue =
     Ask String Dialogue Dialogue
   | Done String
```

A datatype for dialogues

```haskell
data Dialogue =
    Ask String Dialogue Dialogue
  | Done String
```

```haskell
effectsConversation :: Dialogue
effectsConversation =
  Ask "Do you like effects?"
    (Done "They're everywhere though.")
    (Ask "Do you want to control them?"
      (Done "Good luck!")
      (Done "Try Haskell!")
    )
```

Running a dialogue

```haskell
interactiveDialogue :: Dialogue -> IO ()
interactiveDialogue (Ask question no yes) = do
  response <- askBooleanQuestion question
  if response
    then interactiveDialogue yes
    else interactiveDialogue no
interactiveDialogue (Done response) =
  putStrLn response
```

Running a dialogue

```haskell
interactiveDialogue :: Dialogue -> IO ()
interactiveDialogue (Ask question no yes) = do
  response <- askBooleanQuestion question
  if response
    then interactiveDialogue yes
    else interactiveDialogue no
interactiveDialogue (Done response) =
  putStrLn response
```

```haskell
askBooleanQuestion :: String -> IO Bool
askBooleanQuestion question = do
  putStrLn question
  getBool

getBool :: IO Bool
getBool = do
  c <- getChar
  putStrLn ""
  if c == 'y'
    then pure True
    else if c == 'n'
      then pure False
      else do
        putStrLn "Please type 'y' or 'n'"
        getBool
```

Running a dialogue in the browser

```haskell
webDialogue :: Dialogue -> IO ()
webDialogue d =
  scotty 8000 $ do
    get "/" $ from ""
    get "/:responses" $ do
      responseString <- param "responses"
      from responseString
  where
    from responseString = do
      let responses = mapMaybe parseResponse responseString
      case replay d responses of
        Just (Ask question _ _) ->
          htmlPage $ do
            p (string question)
            ul $ do
              li (a ! href (stringValue (responseString <> "y")) $ "yes")
              li (a ! href (stringValue (responseString <> "n")) $ "no")
        Just (Done response) ->
          htmlPage $
            p (string response)
        Nothing -> status status404

htmlPage :: Html -> ActionM ()
htmlPage =
  html . renderHtml . H.html . H.body

parseResponse :: Char -> Maybe Bool
parseResponse 'y' = Just True
parseResponse 'n' = Just False
parseResponse _   = Nothing

replay :: Dialogue -> [Bool] -> Maybe Dialogue
replay (Ask _ _  yes) (True  : responses) = replay yes responses
replay (Ask _ no _  ) (False : responses) = replay no  responses
replay d              []                  = Just d
replay _              _                   = Nothing
```

IO or nothing?

```
IO      a -- IO, exceptions, random numbers, concurrency, ...
Gen     a -- random numbers only
ST s    a -- mutable variables only
STM     a -- software transactional memory log variables only
State s a -- (persistent) state only
Error   a -- exceptions only
Signal  a -- time-changing value
...
```

New effect types can be defined. Effects can be combined.

Conclusions

- Precise types marking the presence of side effects.
- Require us to be explicit about order when effects are present.
- Peace of mind if `IO` is absent.
- Not a high price to pay.
- `IO` actions are first class.
- Encourages coding style that limits side effects.
- More options for testing.
- More precise effect types possible.

`andres@well-typed.com`