Hello HaskellX!

An Introduction to (IO in) Haskell

Andres Löh – Haskell eXchange 2022

Well-Typed
The Haskell Consultants

Hello

```
main = putStrLn "Hello HaskellX!"
```

Well-Typed

Hello

```haskell
main = putStrLn "Hello HaskellX!"
```

```haskell
"Hello HaskellX!" :: String
```

Well-Typed

Hello

```haskell
main = putStrLn "Hello HaskellX!"
```

```haskell
putStrLn :: ...  -> ...
```

```haskell
"Hello HaskellX!" :: String
```

Hello

```
main = putStrLn "Hello HaskellX!"
```

```
putStrLn :: String -> ...
```

```
"Hello HaskellX!" :: String
```

Well-Typed

Hello

```haskell
main = putStrLn "Hello HaskellX!"
```

```haskell
putStrLn :: String -> IO ()
```

```haskell
"Hello HaskellX!" :: String
```

Hello

```haskell
main :: IO ()

main = putStrLn "Hello HaskellX!"

putStrLn :: String -> IO ()

"Hello HaskellX!" :: String
```

A conversation

```haskell
main = do
  putStrLn "Who are you?"
  name <- getLine
  putStrLn ("Nice to meet you, "  <> name)
```

A conversation

```
main = do
  putStrLn "Who are you?"
  name <- getLine
  putStrLn ("Nice to meet you, "  <> name)
```

```
getLine :: IO String
```

A conversation

```haskell
main = do
  putStrLn "Who are you?"
  name <- getLine
  putStrLn ("Nice to meet you, "  <> name)
```

```haskell
name :: String    getLine :: IO String
```

A conversation

```
main = do
  putStrLn "Who are you?"
  name <- getLine
  putStrLn ("Nice to meet you, "  <> name)
```

```
  name :: String    getLine :: IO String

                                (<>) :: String -> String -> String
```

Well-Typed

Wrong

```
main = do
  putStrLn "Who are you?"
  putStrLn ("Nice to meet you, "  <> getLine)
```

Wrong

```haskell
main = do
  putStrLn "Who are you?"
  putStrLn ("Nice to meet you, "  <> getLine)
```

A `String` is expected, but an `IO String` is provided.

Reduction

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

Reduction

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

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

Reduction

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

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

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

Reduction

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

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

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

```
"abcd"
```

Reduction

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

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

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

```
"abcd"
```

Or:

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

Reduction

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

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

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

```
"abcd"
```

Or:

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

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

Reduction

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

Or:

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

Reduction

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

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

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

```
"abcd"
```

Or:

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

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

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

```
"abcd"
```

Reduction order does not matter!

More reduction

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

More reduction

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

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

More reduction

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

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

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

More reduction

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

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

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

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

More reduction

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

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

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

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

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

More reduction

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

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

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

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

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

```
"aFrodobSam"
```

More reduction

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

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

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

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

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

```
"aFrodobSam"
```

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

More reduction

```
("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")
```

Well-Typed

More reduction

```
("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"
```

More reduction

```
("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"
```

Well-Typed

More reduction

```
("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"
```

More reduction

```
("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"
```

More reduction

```
("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"
```

Suddenly reduction order does matter!

Another example

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

reduces to `"a"` .

Another example

```
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?

Well-Typed

Explicit effects

- Decouple effects from the order of evaluation.
- Order and number of effects are always explicit.
- Side-effecting computations are distinguished from their results.

Laws actually hold

```
length (x <> x) = 2 * length x
```

Very sensible.

Laws actually hold

```
length (x <> x) = 2 * length x
```

Very sensible.

But would actually be wrong if we allowed `x` to be `getLine`.

No escape

There is no[*] function of type

```
IO a -> a
```

because we should not lie!

[*](None that we speak of.)

Marking effects is good

```haskell
sum :: [Int] -> Int
```

vs.

```haskell
sumAndSendSpamMails :: [Int] -> IO Int
```

Abstraction

```haskell
main :: IO ()
main = do
  putStrLn "Who are you?"
  name1 <- getLine
  putStrLn "Who are you?"
  name2 <- getLine
  putStrLn
    ("Nice to meet you, " <> name1 <> " and " <> name2)
```

Abstraction

```haskell
whoAreYou :: IO String
whoAreYou = do
  putStrLn "Who are you?"
  getLine
```

```haskell
main :: IO ()
main = do
  name1 <- whoAreYou
  name2 <- whoAreYou
  putStrLn
    ("Nice to meet you, " <> name1 <> " and " <> name2)
```

Well-Typed

Abstraction

```haskell
prompt :: String -> IO String
prompt text = do
  putStrLn text
  getLine
```

```haskell
whoAreYou :: IO String
whoAreYou = prompt "Who are you?"
```

```haskell
main :: IO ()
main = do
  name1 <- whoAreYou
  name2 <- whoAreYou
  putStrLn
    ("Nice to meet you, " <> name1 <> " and " <> name2)
```

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

```haskell
prompts :: [IO String]
prompts =
  map prompt questions
```

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

```haskell
prompts :: [IO String]
prompts =
  map prompt questions
```

```haskell
prompt :: String -> IO String
```

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

```haskell
prompts :: [IO String]
prompts =
  map prompt questions
```

```haskell
prompt :: String -> IO String
```

```haskell
map :: (a -> b) -> [a] -> [b]
```

Well-Typed

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

```haskell
prompts :: [IO String]
prompts =
  map prompt questions
```

```haskell
askQuestions :: IO [String]
askQuestions =
  sequence prompts
```

Asking many questions

```haskell
questions :: [String]
questions =
  ["Who are you?", "Are you a Haskeller yet?"]
```

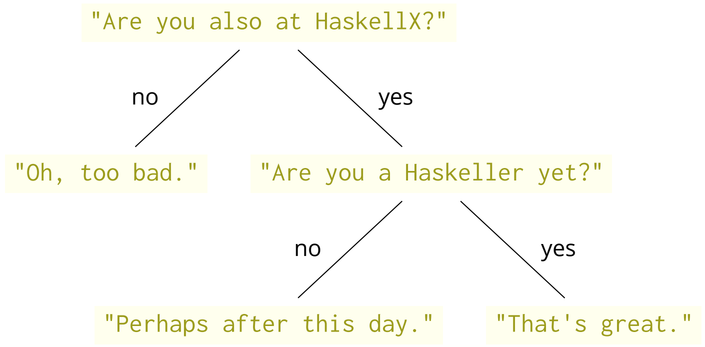
```haskell
prompts :: [IO String]
prompts =
  map prompt questions
```

```haskell
askQuestions :: IO [String]
askQuestions =
  sequence prompts
```

```haskell
sequence :: [IO a] -> IO [a]
```

Well-Typed

Separation of concerns


"Are you also at HaskellX?"
no
yes
"Oh, too bad."
"Are you a Haskeller yet?"
no
yes
"Perhaps after this day."
"That's great."

Well-Typed

A datatype for dialogues

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

A datatype for dialogues

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

```
haskellXConversation :: Dialogue
haskellXConversation =
  Ask "Are you also at HaskellX?"
    (Done "Oh, too bad.")
    (Ask "Are you a Haskeller yet?"
      (Done "Perhaps after this day.")
      (Done "That's great.")
    )
```

Well-Typed

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
```

Well-Typed

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

```
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
```

Well-Typed

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.

Well-Typed

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.
- **Ask many questions.**

`andres@well-typed.com`

Well-Typed