Wrapping up `State`: post-order labelling

From a mutable int to a mutable environment

In the first part of the course, we saw an interpreter that used an environment as an explicit parameter:

```
(interpret expr env)
```

We can model the same kind of behaviour using State:

```
interpret :: Expr -> State (Map String a) a
```

Talking to the outside world: `IO`

"If a program runs without any side effects, does it make a sound?"

Key idea: treat the outside world as some big mutable `State`

Standard input, standard output

```
getLine :: World -> (String, World)
print :: String -> World -> ((), World)
```

File I/O

```
open :: Path -> World -> (FILE *, World)
read :: FILE * -> World -> (String, World)
```

Subprocesses

```
run :: [String] -> World -> (PID, World)
kill :: PID -> World -> ((), World)
```

Network requests

```
checkFacebook :: World -> (☺, World)
submitWork :: Assignment -> World -> (★, World)
```

```
State World a = State (World -> (a, World))
```

```
~~State World a = State (World -> (a, World))~~
```

```
IO a
```

A value of type `IO a` represents an I/O action that produces a value of type `a`.

An I/O action is **only** performed when:

1. We evaluate it in the interpreter.
2. We execute it inside `main` (`main :: IO ()`).

Standard input, standard output (for real)

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

File I/O (for real)

```
openFile     :: FilePath -> IOMode -> IO Handle
hGetContents :: Handle -> IO String
hClose       :: Handle -> IO ()
```

Reminder: next week, we have lectures on *Monday and Wednesday* (both in the regular lecture room).

IO is the ultimate *lack of constraints*

`main :: IO ()` --- `main` can do anything!

IO is the ultimate *lack of constraints*

`main :: IO ()` --- `main` can do anything!

Recall this “identity-ish” function…

```
<T> T f(T x) {
  sendDavidSpam();
  return x;
}
```

In Haskell, sending David spam is an IO action…

```haskell
f :: a -> IO a
f x = do
  _ <- sendDavidSpam
  return x
```

A fundamental design principle of a Haskell program is to have an **impure shell** surrounding an **pure core**.

```
analyzeDataFile :: FilePath -> IO ___
analyzeDataFile path = do
  handle <- openFile path ReadMode
  text <- hGetContents handle   -- text is a String
  let result = processData text -- result is a ___
  _ <- hClose handle
  return result

processData :: String -> ___
```

Type checking: where do types come from?

Main sources for **static** type information:

1. literal values, built-ins
2. type annotations (e.g., `int x;`)

type inference

the act of determining the type of expressions in a program, statically and without annotations

Main sources for **static** type information:

1. literal values, built-ins
2. type annotations (e.g., `int x;`)
3. how expressions are used

We’ve already seen how expression types generate constraints on how expressions can be used.

“If x is a `Bool`, then `(x && True)` is valid.”

“If x is not a `Bool`, then `(x && True)` is not valid.”

But how expressions are used also generate constraints on expression types!

“If `(x && True)` is valid, then `x` is a `Bool`.”

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

```
f x y = (words x) !! y
```

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

```
f x y = (words x) !! y
```

$\underbrace{\texttt{words x}}_{\texttt{[String]}}$

x :: String

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

```
f x y = (words x) !! y
```

[String]

x :: String

String

y :: Int

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

```
f x y = (words x) !! y
```

[String]

x :: String

String

y :: Int

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

```
f x y = (words x) !! y
```

[String]

x :: String

String

y :: Int

```
f :: String -> Int -> String
```

Sometimes *no* constraints are generated for an expression.
This leads to **generic polymorphism**.

```haskell
f x y z = (words x) !! y

f :: String -> Int -> a -> String
```

Constraints between types

```
f x y z = if x then y else z
```

branch types must match, but could be any type

Typeclass constraints are generated by using their member functions.

```
f x y z =
  if x == y
  then z
  else z + 1
```

Too specific: `f :: Int -> Int -> Int -> Int`

```
(==) :: Eq a => a -> a -> Bool
(+) :: Num a => a -> a -> a
```

```
f x y z =
  if x == y
  then z
  else z + 1
```

Too general: `f :: a -> a -> b -> b`

```
(==) :: Eq a => a -> a -> Bool
(+) :: Num a => a -> a -> a
```

```
f x y z =
  if x == y
  then z
  else z + 1
```

Just right: `f :: (Eq a, Num b) =>`
`a -> a -> b -> b`