Slide 1

Imperative Programming in a Functional Language

Advanced Functional Programming (04s1)

Manuel M. T. Chakravarty
University of New South Wales

These two lectures are based on *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, Simon L. Peyton Jones, 2000 Marktoberdorf Summer School. See `~cs4132/papers/`.

Slide 2

Imperative Versus Functional Programming

What is Imperative Programming?

- Commands that alter the state of the machine
- The most essential command: assignment

What is Functional Programming?

- Composition of mappings from inputs to outputs
- No notion of an implicit state

The Hardcore FP Programmer's View?

- Imperative programming is merely a special case of functional programming
- Imperative program: FP program with an implicit state parameter threaded through all calls

Slide 3

How does that work?

Consider this function in C:

```
int foo (float x)
{
  <plenty of assignments>
  return <expr>;
}
```

It is roughly equivalent to

```
foo                 :: State -> Float -> (Int, State)
foo initialState x  =
  let
    finalState = <assignment-like state manipulation>
  in
  (<expr>, finalState)
```

Slide 4

Now consider the following use of the C routine:

```
y = foo (1.0) + foo (2.0);
```

What is the result?

In a FP, we have to be explicit about the state manipulation:

```
let
  (state'  , res1) = foo state  1.0
  (resState, res2) = foo state' 2.0
in
res1 + res2
```

In other words, the implicit state becomes explicit!

Slide 5

TRADEOFFS

Pro Explicit State:

- ➜ Semantics is clear
- ➜ Code is compositional
- ➜ Code is reentrant (and race conditions are easier to spot)
- ➜ Persistence is easier to achieve

Contra Explicit State:

- ➜ Carrying the state around is tedious
- ➜ Mistakes in plumbing the state can be fatal (linearity!)
- ➜ Some state we can't really get hold off (disks, screen, the World!)

> *In particular, the world is never loaded into a register, stored in a data structure, or passed to C procedure calls.*
> *– Peyton Jones & Wadler in "Imperative functional programming"*

Can we have our cake and eat it, too?
Yes—welcome to the wonderful world of monads!

Slide 6

THE AWKWARD SQUAD LINEUP

- ➜ Input & Output: terminal I/O, file I/O, graphics, networking, etc.
- ➜ Concurrency
- ➜ Exceptions
- ➜ Interfacing code written in other languages

Observation: This is largely about operational, rather than denotational program elements.
(In other words: behaviour, not values.)

Slide 7

THE BASELINE: INPUT & OUTPUT

The key idea of monadic I/O:

> A value of type `IO a` is an "action" that, when performed, may do some input/output, before delivering a value of type `a`.
> — remember COMP1011?

Given the story about side effects and explicit state,

```
type IO a = World -> (a, World)
```

Moreover, we have operations, such as

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

✧

Slide 8

The "`do`" Notation:

- ➜ The explicit use of >>=, >>, and lambda abstractions gets very tedious
- ➜ The "`do`" notation let's us write:

```
do
  <statement 1>
  ...
  <statement n>
```

where each statement is either

① an I/O action without a result,

② of the form

```
var <- <IO action>
```

or

③ of the form `return <expr>`.

Comparing it now to the notation used in C:

```
int foo (float x)
{
  <plenty of assignments>
  return <expr>;
}
```

Slide 9

becomes

```
foo   :: Float -> IO Int
foo x  = do
           <assignment-like state manipulation>
           return <expr>
```

The natural question: What about pointers?

Let's take a step back first!

- ➜ What's going on? Where is the explicit state gone?
- ➜ It is being encapsulated in the `IO` monad:

```
type IO a = World -> (a, World)
```

- ➜ The functions `>>=`, `>>`, and `return` make sure that is plumbed through `IO` actions.
- ➜ In other words, the `IO` monad serves as an abstraction facility:

  It removes the state from our sight (makes it implicit like in C) and makes sure it is being handled properly.

Slide 10

Let's review the arguments against explicit state:

- ✔ Carrying the state around is tedious
- ✔ Mistakes in plumbing the state can be fatal (linearity!)
- ✔ Some state we can't really get hold off (disks, screen, the World!)

We get all the benefits of implicit state, but can distinguish pure from impure functions (by their type) and have a handle on the semantics.

`IORef`: Pointers in Haskell

Given an implicit state, just a matter of the API:

```
data IORef a          -- a pointer to a value 'a'

newIORef   :: a -> IO (IORef a)      -- creation
readIORef  :: IORef a -> IO a        -- de-referencing
writeIORef :: IORef a -> a -> IO ()  -- assignment
```

Slide 11

- ➜ These pointers can only point to Haskell objects
- ➜ Latter on, we will meet C-style pointers that are just numbers interpreted as addresses in a process' address space (including pointer arithmetic)

Side Remarks

Unsafe I/O:

- ➜ Sometimes we don't care about the order in which side effects are performed:

```
unsafePerformIO :: IO a -> a    -- DANGEROUS OPERATION!!!
```

- ➜ For example, we may want to print debugging messages in purely functional code (see also, `Debug.Trace.trace`)

Slide 12

Implementation considerations:

- ➜ How do we represent the `World`?

```
type IO a = World -> (a, World)
```

- ➜ Do we make this representation explicit in the compiler?
- ➜ There is some discussion in SPJ's paper.

Semantics: The Meaning of Implicit State

Slide 13

Can we use a denotational semantics?

- What is a denotational semantics?
- `www.foldoc.org` says

  A technique for describing the meaning of programs in terms of mathematical functions on programs and program components. Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory.
- Here a domain is a mathematical structure consisting of a set of values (or "points") and an ordering relation, $\subseteq$ on those values.
- Objects of different type map to different domains.
- Eg, a domain $X \to Y$ may model functions from domain X to domain Y, where $f \subseteq g$ iff for all x in X, $f x \subseteq g x$
- The meaning of programs is, then, expressed in the form of fixed points over the corresponding domains.

Slide 14

Using denotational semantics for defining the meaning of `IO` implies to find a proper domain for

```
type IO a = World -> (a, World)
```

Problems with this approach:

- The programs

  ```
  loop :: IO ()
  loop  = loop
  ```

  and

  ```
  loopX :: IO ()
  loopX  = putChar 'x' >> loopX
  ```

  would be assigned the same denotation.
- It is hard to add concurrency.

All this may be overcome in a trace-based approach (in the spirit of trace-based CSP semantics), but it seems easier to deal with an operational semantics.

An Operational Semantics for `IO`

Slide 15

- Semantics stratified into two levels:
  1. inner denotational semantics (for the meaning of pure terms) and
  2. outer monadic transition semantics (to describe the behaviour of `IO` computations).
- So, only the awkward squad gets a transition semantics; for the pure part of the language, we can still use the usual denotational approach.
- In the semantics, we don't consider all of Haskell, but only it's *core* (see language definition)
- We add a set of new constants (over a purely functional language): these constants represent monadic operations

Slide 16

$$
\begin{array}{ll}
x,\ y & \ni \mathit{Variable} \\
k & \ni \mathit{Constant} \\
con & \ni \mathit{Constructor} \\
c & \ni \mathit{Char}
\end{array}
$$

Terms:

$$M, N, H \quad ::= \quad x \mid V \mid M\ N \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid \cdots$$

Values:

$$
\begin{array}{lll}
V & ::= & \backslash x\ \texttt{->}\ M \mid k \mid con\ M_1 \cdots M_n \mid c \\
 & \mid & \texttt{return}\ M \mid M\ \texttt{>>=}\ N \mid \texttt{putChar}\ c \mid \texttt{getChar}
\end{array}
$$

Terms versus values:

- Terms are the input to our semantics; they represent programs
- Values are the output of the purely-functional part of the semantics

Slide 17

Points to note:

- Primitive monadic `IO` operations are treated as values: they are not interpreted in the purely functional part of the semantics
- As values, some monadic operations have values as arguments: this ensures that their arguments are always evaluated (strictness)

Slide 18

A Labelled Transition System

- Program states evolve into new program states by making a transition
- We represent a program state as a term in curly braces: $\{M\}$
- A labelled transition

$$P \rightarrow^{\alpha} Q$$

rewrites P into Q when the event α occurs

- Events represent communication with the external environment

In particular, we have

- $P \rightarrow^{!c} Q$ means "P can move to Q by writing character c to stdout"
- $P \rightarrow^{?c} Q$ means "P can move to Q by reading character c from stdin"

Slide 19

Two transition rules of our semantics:

$$\{\texttt{putChar}\ c\} \rightarrow^{!c} \{\texttt{return ()}\}$$
$$\{\texttt{getChar}\} \rightarrow^{?c} \{\texttt{return}\ c\}$$

But how do we compose these to cover more complicated programs?

Slide 20

Evaluation Contexts

Consider

```
main = getChar >>= \c -> putChar c
```

- How can we formalise the sequentiality?
- Consider that it can get more difficult:

```
main = (getChar >>= \c -> getChar) >>= \d -> return ()
```

How about the sequencing here?

- We use evaluation contexts: holes that determine which operation comes next:

$$\mathbb{E} ::= [\cdot] \quad | \quad \mathbb{E} \texttt{ >>= } M$$

- We write $\mathbb{E}[M]$ to denote filling the hole of an evaluation context.

Slide 21

Transition Rules:

$$\{\mathbb{E}[\texttt{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\texttt{return ()}]\} \quad \text{(PUTC)}$$

$$\{\mathbb{E}[\texttt{getChar}]\} \xrightarrow{?c} \{\mathbb{E}[\texttt{return}\ c]\} \quad \text{(GETC)}$$

$$\{\mathbb{E}[\texttt{return}\ N \texttt{ >>= } M]\} \longrightarrow \{\mathbb{E}[M N]\} \quad \text{(LUNIT)}$$

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{\mathbb{E}[M]\} \longrightarrow \{\mathbb{E}[V]\}} \text{(FUN)}$$

Slide 22

Observations:

- Using this operational semantics, we can distinguish between `loop` and `loopX`:

```
loop  = loop
loopX = putChar 'x' >> loopX
```

- How does this semantics help us? We can reason about programs and we can reason about the semantics:
  - We can tell when two programs are the same.
  - We can tell whether a program meets its specification.
  - We can establish properties that hold for all possible programs.
- We can quite nicely extend this to cover `IORef`s, too.

— End of Part I —

Slide 23

The Meaning of IORefs

Remember...

- We are dealing with a two-level semantics
- Purely functional computations (i.e., those with a type other than `IO` a) are given a meaning by an inner denotational semantics
- This corresponds to an evaluation function

$$\mathcal{E}[\![\cdot]\!] :: M \to V$$

taking terms to values.

- The behaviour of `IO` computations is defined by an outer monadic transition semantics
- This is defined by way of labelled transitions

$$P \xrightarrow{\alpha} Q$$

where α, so far, may be $!c$ (output) or $?c$ (input)

Slide 24

Terms:

$$M, N, H \quad ::= \quad x \mid V \mid M\ N \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid \cdots$$

Values (including support for `IORef`):

$$\begin{aligned} V \quad ::= \quad & \backslash x \texttt{ -> } M \mid k \mid con\ M_1 \cdots M_n \mid c \\ \mid \quad & \texttt{return}\ M \mid M \texttt{ >>= } N \mid \texttt{putChar}\ c \mid \texttt{getChar} \\ \mid \quad & \texttt{writeIORef}\ r\ N \mid \texttt{readIORef}\ r \mid \texttt{newIORef}\ M \mid r \end{aligned}$$

where $r \in$ `IORef`

Evaluation Contexts:

$$\mathbb{E} \quad ::= \quad [\cdot] \mid \mathbb{E} \texttt{ >>= } M$$

So, we have

$$\texttt{(getChar >>=}\backslash c \texttt{ -> getChar) >>=}\backslash d \texttt{ -> return ()} = \mathbb{E}[\texttt{getChar}]$$

where

$$\mathbb{E} = \texttt{(}[\cdot] \texttt{ >>=}\backslash c \texttt{ -> getChar) >>=}\backslash d \texttt{ -> return ()}$$

Slide 25

Transition Rules:

$$\{\mathbb{E}[\texttt{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\texttt{return ()}]\} \quad \text{(PUTC)}$$

$$\{\mathbb{E}[\texttt{getChar}]\} \xrightarrow{?c} \{\mathbb{E}[\texttt{return}\ c]\} \quad \text{(GETC)}$$

$$\{\mathbb{E}[\texttt{return}\ N \texttt{ >>= } M]\} \longrightarrow \{\mathbb{E}[M N]\} \quad \text{(LUNIT)}$$

$$\{\mathbb{E}[\texttt{readIORef}\ r]\} \mid \langle M\rangle_r \longrightarrow \{\mathbb{E}[\texttt{return}\ M]\} \mid \langle M\rangle_r \quad \text{(READIO)}$$

$$\{\mathbb{E}[\texttt{writeIORef}\ r\ N]\} \mid \langle M\rangle_r \longrightarrow \{\mathbb{E}[\texttt{return ()}]\} \mid \langle N\rangle_r \quad \text{(WRITEIO)}$$

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{\mathbb{E}[M]\} \longrightarrow \{\mathbb{E}[V]\}} \text{(FUN)}$$

$$\frac{r \notin fn(\mathbb{E}, M)}{\{\mathbb{E}[\texttt{newIORef}\ M]\} \longrightarrow \nu r.\{\mathbb{E}[\texttt{return}\ r]\} \mid \langle M\rangle_r} \text{(NEWIO)}$$

Enrich Program States for `IORefs`:

$P, Q, R ::= \{M\} \mid \langle M\rangle_r \mid (P \mid Q) \mid \nu x.P$

Slide 26

There is a problem with this...

Consider

$$\langle M\rangle_r \mid \langle M\rangle_s \mid \mathbb{E}[\texttt{readIORef}\ r]$$

- ➜ If you define a formal system, you have to be really precise
- ➜ We can't just swap parts of a program state around without formalising what is allowed and what not
- ➜ The way out:
  - structural congruence relation (e.g., "|" is commutative and associative)
  - structural transitions (e.g., transition to the left of "|")

Slide 27

$$P \mid Q \equiv Q \mid P$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$\nu x.\nu y.P \equiv \nu y.\nu x.P$$

$$(\nu x.P) \mid Q \equiv \nu x.(P \mid Q) \quad , x \notin fn(Q)$$

$$\nu x.P \equiv \nu y.P[y/x] \quad , y \notin fn(P)$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \text{(PAR)} \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \text{(NU)}$$

$$\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \text{(EQUIV)}$$

Slide 28

CONCURRENCY

Example Applications:

- ➜ Web server: servicing multiple clients simultaneously
- ➜ GUI programs: background work, non-modal behaviour
- ➜ We are not concerned about parallelism here:
  - Parallelism uses concurrency for the sake of performance (and nothing else)
  - Concurrent programs have concurrency as part of their specification (not just as a matter of a faster implementation)

Slide 29

Concrete Example: From a Web Server

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket =
  forever $ do
              conn <- accept socket
              forkIO (serviceConn config conn)
  where
    forever a = a >> forever a  -- new control structure!
```

with

```
type Connection = (Handle,      -- stream handle
                   SockAddr)    -- peer details
accept :: Socket -> IO Connection
```

and

```
forkIO :: IO a -> IO ThreadId
```

- ➜ `forkIO` gets an `IO` computation as an argument
- ➜ The forked thread doesn't get any extra arguments (compare this to `pthread_create()`)

Slide 30

Communication:

- ➜ How can multiple threads in one program communicate?
- ➜ How can threads communicate with the rest of the world?
- ➜ What sort of interference can there be between the I/O operations of different threads?

Communication by `IORefs`:

- ➜ Not a good idea!
- ➜ Without extra synchronisation, we would have lots of races

Alternatives? `IORefs` With Builtin Synchronisation:

```
data MVar a              -- abstract
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO a
```

- ➜ An `MVar` can be empty or full
- ➜ `putMVar` puts a value into an empty `MVar`
- ➜ `takeMVar` takes a value from a full `MVar` and leaves it empty
- ➜ `putMVar` blocks on full and `takeMVar` on empty `MVars`

Slide 31

Example Usage:

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket = do
  count <- newEmptyMVar
  putMVar count 0
  forever $ do
              conn <- accept socket
              forkIO $ do
                         inc count
                         -- do something if ‘count’ to high
                         serviceConn config conn
                         dec count

inc, dec :: MVar Int -> IO ()
inc count = do
              v <- takeMVar count
              putMVar count (v + 1)
```

Slide 32

SEMANTICS OF CONCURRENCY

Values:

$$V \;::=\; \backslash x \texttt{ -> } M \mid k \mid con\ M_1 \cdots M_n \mid c$$
$$\mid\ \texttt{return } M \mid M \texttt{ >>= } N \mid \texttt{putChar } c \mid \texttt{getChar}$$
$$\mid\ \texttt{forkIO } M \mid t \mid d$$
$$\mid\ \texttt{putMVar } m\ N \mid \texttt{takeMVar } m \mid \texttt{newEmptyMVar} \mid m$$

Program States:

$$P, Q, R ::= \{M\}_t \mid \langle M \rangle_m \mid \langle\rangle_m \mid (P \mid Q) \mid \nu x.P$$

$$\frac{y \notin fn(M, \mathbb{E})}{\{\mathbb{E}[\texttt{forkIO } M]\}_t \longrightarrow \nu u.\mathbb{E}[\texttt{return } u] \mid \{M\}_u} \quad \text{(FORK)}$$

$$\frac{y \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newEmptyMVar}]\}_t \longrightarrow \nu m.\mathbb{E}[\texttt{return } m] \mid \langle\rangle_m} \quad \text{(NEWM)}$$

$$\{\mathbb{E}[\texttt{takeMVar } m]\}_t \mid \langle M \rangle_m \longrightarrow \{\mathbb{E}[\texttt{return } M]\}_t \mid \langle\rangle_m \qquad \text{(TAKEM)}$$

$$\{\mathbb{E}[\texttt{putMVar } m\ M]\}_t \mid \langle\rangle_m \longrightarrow \{\mathbb{E}[\texttt{return ()}]\}_t \mid \langle M \rangle_m \qquad \text{(PUTM)}$$

Slide 33

Higher Level Concurrency Primitives

- Are `MVars` sufficient?
- Concurrent programs often need abstractions such as semaphores, channels, etc

`MVars` can be used to define all these efficiently:

- `MVars` are a kind of minimal communication primitive
- Haskell's Hierachical Libraries define a whole family of further concurrency abstractions in terms of `MVars`: channels, semaphores, quantity semaphores, sample variables, and streams that are merged nondeterministically

Slide 34

Exceptions and Timeouts

- Robust programs have to allow for I/O errors, timeouts, and possibly even programmer errors.

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket =
  forever $ do
              conn <- accept socket
              forkIO (service conn)
  where
    service conn   = serviceConn config conn
                     `catch`
                     handler conn
    handler :: Connection -> Exception -> IO ()
    handler conn e = do
                       logError config e
                       hClose (fst conn)
```


Slide 35

Haskell 98 Exceptions:

- Exceptions can be raised and caught in `IO` computations
- E.g., good for handling I/O errors
- But can't catch exceptions, such as pattern matching errors, and can't handle asynchronous exceptions

Exceptions in Pure Code:

- Introduce

  ```
  throw :: Exception -> a
  ```
- What do

  ```
  length [throw ex]
  ```

  and

  ```
  throw ex1 + throw ex2
  ```

  behave?
- Semantics uses exceptional values and imprecise exceptions
- Exceptions can only be caught in `IO` computations

Slide 36

Asynchronous Exceptions:

- Like signals between PThreads, one thread may want to raise an exception in another thread
- New primitive:

  ```
  throwTo :: ThreadId -> Exception -> IO ()
  ```
- This also allows us to implement timeouts
- The transition semantics that we developed stepwise can also be extended to cover exceptions (see the paper)

Interfacing to Code Written in Other Languages

➜ There already exist heaps of libraries in other languages
➜ In particular, we need to be able to access the system libraries (e.g., socket API)
➜ We also want to make use of the windowing system, GUI toolkits, etc.

What Functionality Must a Foreign Function Interface (FFI) Have?

Slide 37

➜ Call routines written in other languages from Haskell
➜ Call from other languages routines written in Haskell
➜ Convert data from Haskell into the formats expected by other languages and vice versa
➜ Marshall data structures from the representation of one language in that of another
➜ Startup and shutdown the Haskell runtime from within a program written in another language

Haskell 98 FFI Addendum:

➜ Extension of the H98 standard proposed for standardisation
➜ Currently, supports interaction between Haskell and C
➜ Support for Java and .NET is being worked on

Properties:

Slide 38

➜ Minimal impact in existing standard (adds only one new keyword, namely `foreign`)
➜ General framework that can be instantiated to a range of foreign languages (or rather calling conventions)
➜ Marshalling is heavily library based:
  ➜ The language extension only handles primitive types (`Int`, `Float`, etc)
  ➜ Generic marshalling support
  ➜ For each foreign language,
    - representation of primitives of that language and
    - language-specific marshalling (e.g., `CString`)

Calling C From Haskell:

Slide 39

➜ What is needed?
➜ A `foreign import` declaration associates a Haskell type signature with a C function
➜ For example, given

```
void putChar (char c);
```

we could use

```
foreign import ccall putChar :: Char -> IO ()
```

Calling Haskell From C:

➜ Conversely, `foreign export` makes a Haskell routine

```
foreign export ccall foo :: Float -> Int -> IO Float
```

accessible in C as

```
float foo (float, int);
```

C Pointers:

Slide 40

➜ Finally, we have to face the question: How do we handle C-style pointers in Haskell?
➜ Pointers are just addresses, i.e., numbers—what's the deal?
➜ We could define

```
type Addr = Int    -- but watch the bitsize!
```

➜ What's the disadvantage here?

Parametrised Pointers:

➜ We want to preserve the type information about what a pointer refers to:

```
newtype Ptr a = Ptr Int
```

✧(http://haskell.org/ghc/docs/latest/html/libraries/base/Foreign.Ptr.html)

Slide 41

Help With the FFI:

- Writing a Haskell binding for a large C library can become rather tedious and time consuming
- If the C library changes, we can end up with difficult to find bugs
- There are tools to make life easier:
- GreenCard
- H/Direct
- C→Haskell
- `hsc2hs`

◇

Slide 42

Summary

- Purely functional programming is nice, but
- real programs need to affect the environment and sometimes require control over operational aspects such as concurrency

The Monadic Approach:

- Separate purely functional from side-effecting/operational computations (akin to two-level languages)
- Expressions of type `IO` a form an imperative sub-language
- Semantics uses the standard denotational definition for the purely functional part and a labelled transition system for the `IO` sub-language

Related Material:

- The Haskell web server (see paper of Simon Marlow)
- Textbooks on semantics
- Functional reactive programming