Software System Design and Implementation

Controlling Effects

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia

COMP3141 18s1

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Examples of effects

```
printf ("Hello World!");
```

I/O effect

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

I/O effect

```
int *p;

 :
*p = *p + 2;
```

write effect

read effect

```
class MyException
extends Exception {}

 :
throw new MyException();
```

exception effect (non-local control flow)

```
pthread_t mythread;

pthread_create (&mythread, NULL, thread_function,
                NULL);
```

thread-creation effect

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA

Internal versus external effects

- External effects can be observed outside of the function where they occur
    - I/O is an external effect
    - Accessing a global variable may be an external effect
- Internal effects cannot be observed on the outside
    - Allocating, using, and deallocating memory
- We can often treat a function with only internal effects as a pure function
    - Purity is about what is observable!

A definition of pure functions

- A pure function is fully specified by a mapping of argument to result values
- Consequences include the following:
  - Two invocations with the same arguments result in the same result
  - A pure function leaves no observable trace beyond its result
- Caveat:
  - Purity pertains to a particular level of abstraction
  - After all, the assembly instructions of a pure Haskell function are not pure

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

A definition of impure or effectful functions

- An impure or effectful function is one that is not pure:
  - it makes use of information beyond its arguments or
  - produces an observable effect beyond its result (or both)
- They are not functions in the mathematical sense; they are sometimes called procedures

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Why are effects harmful?

- They introduce (often subtle) requirements on the execution order
- They are not readily apparent from a function prototype or signature
- They introduce non-local dependencies
- They interfere badly with strong typing; for example:
    - Subtyping and mutable arrays in Java (even worse with generics)
    - Polymorphism and mutable references in ML

Effects and execution order

- Execution order can be surprising
- Execution order can be indeterminate:
  - Global object initialisation
  - Concurrency

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
    if (getchar () < getchar ()) printf ("yes\n"); else printf ("no\n");
    return 0;
}
```

```c
#include <stdio.h>

void compare_chars (char x, char y)
{
    if (x < y) printf ("yes\n"); else printf ("no\n");
}

int main (int argc, char *argv[])
{
    compare_chars (getchar (), getchar ());
    return 0;
}
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Avoiding effects

- Without effects, all functions are pure
  - Can you program like that?
  - Experience suggest, yes — actually, very well!
  - Need to get used to the programming style, though
- It impacts the program structure (often positively)
- It may require the use of different algorithms
- Leads to purely functional programming

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Sometimes we need effects

- Most notably: I/O is usually effectful
- Interoperating with impure languages requires effects
- Sometimes effectful algorithms are more efficient
    - Internal effects are usually sufficient

Haskell's approach:
Pure by default, effectful when necessary!

Haskell functions are pure by default

```
f :: a -> b
```

- Maps a value of type `a` to a value of type `b` without any effects
- Effectful functions require specialised types

Haskell functions are pure by default

```
f :: a -> b
```

- Maps a value of type **`a`** to a value of type **`b`** without any effects

- Effectful functions require specialised types

perform external effects and then return a value of type b

```
g :: a -> IO b
```

Effects need to be carefully contained

- Effects have to be reflected in the type:

```
g :: a -> World -> (b, World)
```

```
printStr ::  String -> World -> ((), World)
```

```
getChar ::  World -> (Char, World)
```

- Why is this problematic?

Effects need to be carefully contained

- Effects have to be reflected in the type:

```
g :: a -> World -> (b, World)
```

```
newtype IO b = IO (World -> (b, World))
```

conceptually, the `World` is hidden inside the abstract data type `IO`

```
g :: a -> IO b
```

```
main      :: IO ()
```

```
getChar   :: IO Char
```

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

What can we do with `IO` operations?

- Combine them to form more complex IO-operations using the do notation:

```
do {
  putStrLn “Hi, ”;               :: IO()
  putStrLn “how are you?”;       :: IO()
}
```

Braces and semicolon optional

- This is a kind a function composition, as the world is passed from the first to the second operation, and so on

- All operations in the do have to be of type `IO`

The do notation and types

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

Type mismatch:
`IO Char != Char`

```
putChar getChar
```

Type error!

`IO Char`

`Char`

```
do {
  c <- getChar;
  putChar c;
}
```

Strips the `IO`

What can we do with IO operations?

```
do {
    ch1 <- getChar;                    :: IO Char
    ch2 <- getChar;                    :: IO Char
    if (ord ch1 < ord ch2)
      then putStrLn “yes”              :: IO ()
      else putStrLn “no”               :: IO ()
  }
```

What can we do with `IO` operations?

- Call pure functions and bind their return value to a variable:

```
do {
  ch <- getChar;                  :: IO Char
  let chUp = toUpper ch           :: IO Char
  return (ord chUp)               :: IO Int
 }
```

```
return :: a -> IO a
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

What can't we do with `IO` operations?

- We cannot write a function of type:

```
IO a -> a
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Effects need to be carefully contained

```
f x = ...g...

g :: a -> IO b
g y = ...
```

contaminated by effects

- If a pure function `f` calls an impure function `g`, `f` becomes impure
- We have the following rule:
  - Only impure functions can call impure functions
  - Unless the inner function contains only internal effects that we encapsulate

Haskell uses the type system, to enforce this rule

Local state

- Sometimes, local state is sufficient

```
g :: a -> s -> (b, s)
```

state can be different things

```
State s b
```

```
runState :: State s a -> s -> (a, s)
```

- Let's say we want a global counter:

```
type Counter = State Int

getCnt :: Counter Int
incCnt :: Counter ()
setCnt :: Int -> Counter ()
```

```
incCounterMax :: Int -> Counter Bool
incCounterMax max = do
  curr <- getCnt
  if curr < max
    then do {incCnt; return False}
    else do {setCnt 0; return True}
```

Which on type constructors can we use the ‘do’ notation?

- The type constructor `m` has to be a ***monad***

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

called “bind”

- The do-notation is just syntactic sugar to make using bind more convenient
- Following properties have to be met:

```
• return a >>= k                =    k a
• ma >>= return                 =    ma
• ma >>= (\x -> k x >>= h)      =   (ma >>= k) >>= h
```

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA

Monads don't necessarily encapsulate state

- `Maybe` type constructor is another example of a monad

```
data Maybe a
  = Nothing
  | Just a
```

```
return :: a -> Maybe a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _   = Nothing
(>>=) (Just x) f = f x
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Monads don’t necessarily encapsulate state

- `[ ]` type constructor is another example of a monad

```
return :: a -> [a]
return x = [x]

(>>=) :: [a] -> (a -> [b]) -> [b]
(>>=) as f  = concat (map f as)
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Generators in QuickCheck are monads

```
searchTrees :: Gen BinaryIntTree
searchTrees = sized searchTrees'
  where
   searchTrees' 0 = return Leaf
   searchTrees' n = do
      v <- (arbitrary :: Gen Int)
      fmap (insert v) (searchTrees' $ n - 1)
```

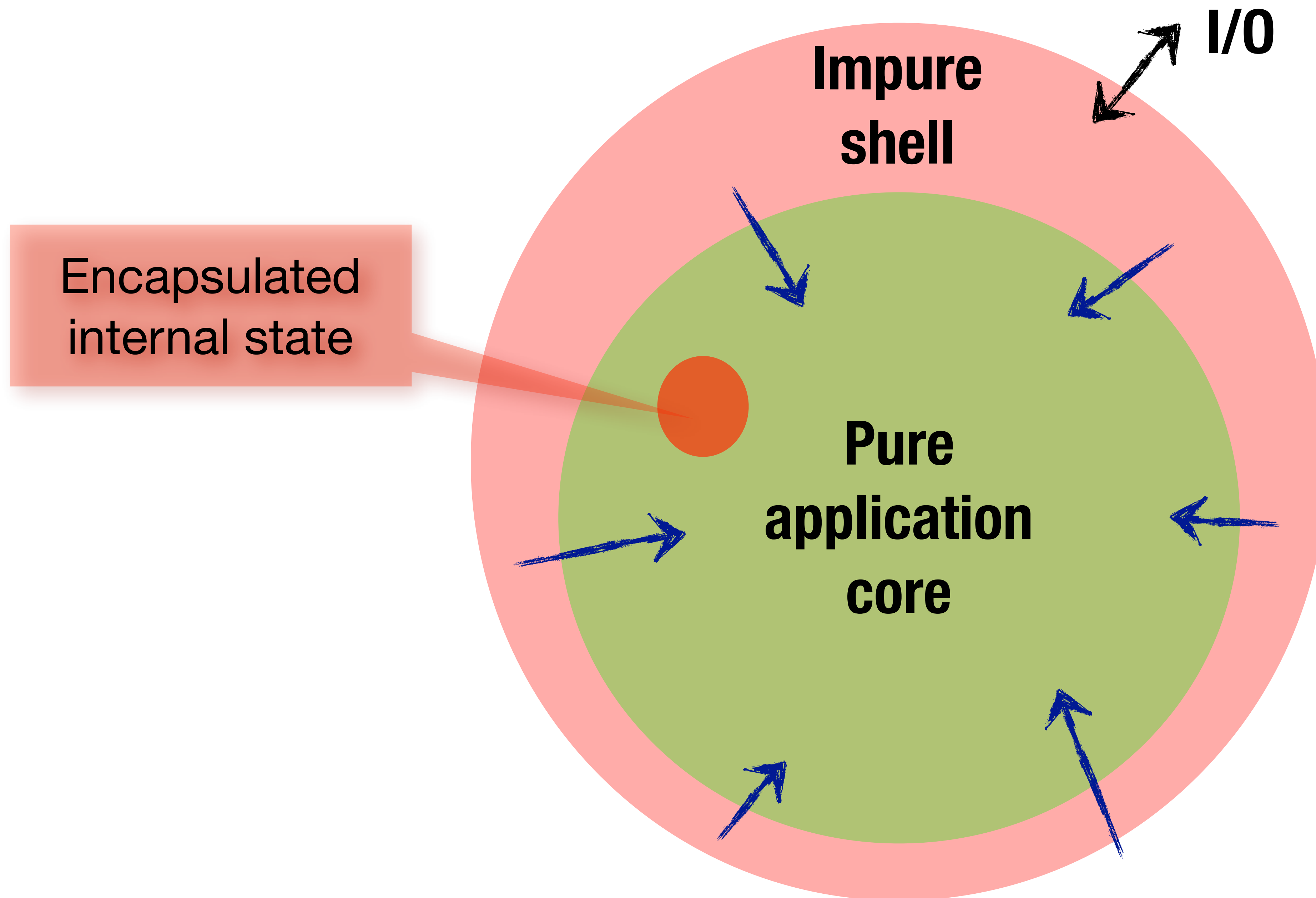
```
fmap :: (a -> b) -> Gen a -> Gen b
```

The benefits of controlling effects

- Absence of effects makes strong typing in pure functions more powerful
  - A type signature captures the entire interface of a function
  - All dependencies are explicit in the form of data dependencies
  - All dependencies are typed
- It is easier to reason about pure code & and it is easier to test pure code
  - Testing and reasoning (formal & informal) is local, independent of context
  - Type checking leads to stronger guarantees

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA

The pure-by-default architecture



Mutable variables in Haskell

Mutable variables in `IO` computations

```
data IORef a
newIORef   :: a               -> IO (IORef a)
readIORef  :: IORef a         -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Let's look at an example

```haskell
import Data.IORef

printIORef :: IORef Int -> IO ()
printIORef ref
  = do
      v <- readIORef ref
      print v

main
  = do
      ref <- newIORef 10
      printIORef ref
      writeIORef ref 42
      printIORef ref
```

Two flavours of mutable variables

```
data IORef a
newIORef   :: a              -> IO (IORef a)
readIORef  :: IORef a        -> IO a
writeIORef :: IORef a -> a -> IO ()
```

can only be run from `main`

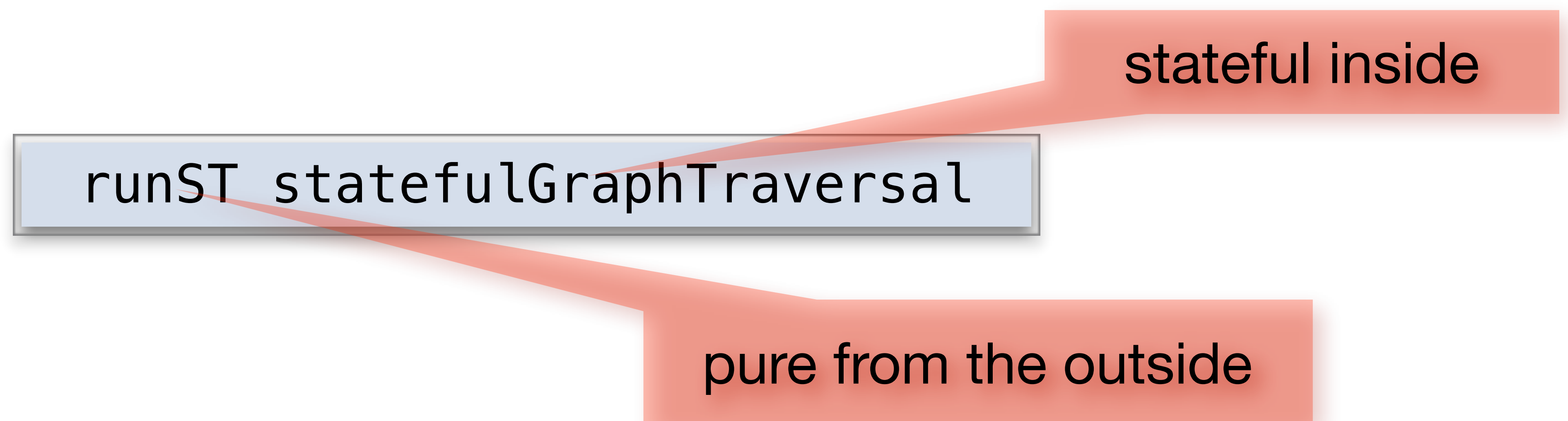
Let's look at an example

```
data STRef s a
newSTRef   :: a                -> ST s (STRef s a)
readSTRef  :: STRef s a        -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

run with `runST :: (forall s. ST s a) -> a`

Encapsulated state

- Some algorithms suggest stateful code
  - For example, a graph traversal marking visited nodes to spot cycles



- The type variable `s` in `STRef s a` represents a state thread
- It ensures that state from one `runST` invocation cannot leak into another

IO versus ST

```
data IORef a
newIORef   :: a                -> IO (IORef a)
readIORef  :: IORef a          -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
data STRef s a
newSTRef   :: a                    -> ST s (STRef s a)
readSTRef  :: STRef s a            -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

- Both `IO` and `ST s` mark the use of state
- Both can be used with the `do` notation

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA