COMP3141

Software System Design and Implementation

Lecture 5: Effects, State Management

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

Announcements

Assignment 1: due July 2.

Warning

That is a Sunday. But support may be sparse over the weekend. Plan accordingly!

Help sessions

Use the extra help sessions this week:

- Wednesday 10AM-12PM (online)
- Thursday 1-3PM (Quadrangle G045)
- Friday 1-3PM (online)

Midway

It's Week 5. We're halfway through!

What have we accomplished?

- Mastered the rudiments of Haskell programming.
- Learned basic reasoning methods.
- Encountered useful algebraic structures.
- Data transformations and algorithm implementation in mathematically structured programs.

Midway

It's Week 5. We're halfway through!

Next: larger-scale mathematically structured system design.

Midway

It's Week 5. We're halfway through!

Next: larger-scale mathematically structured system design.

- Previous focus: **data** structures.
- New focus: **control** structures.

Midway

It's Week 5. We're halfway through!

Next: larger-scale mathematically structured system design.

- Previous focus: **data** structures.
- New focus: **control** structures.
- An adept Haskell programmer needs to also master:
- **Control.Monad** (monads)
- **Control.Lens** (lenses, folds, traversals)

Midway

It's Week 5. We're halfway through!

Next: larger-scale mathematically structured system design.

- Previous focus: **data** structures.
- New focus: **control** structures.
- An adept Haskell programmer needs to also master:
  - **Control.Monad** (monads)
  - **Control.Lens** (lenses, folds, traversals)
- The remainder of this course will mostly be about **Control.Monad**.

Effects

Effects

Effects are observable phenomena from the execution of a program.

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

```
int *p = ...
... // read and write
*p = *p + 1;
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

```
int *p = ...
... // read and write
*p = *p + 1;
```

Example (IO)

```
// console IO
c = getchar();
printf("%d",32);
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

```
int *p = ...
... // read and write
*p = *p + 1;
```

Example (IO)

```
// console IO
c = getchar();
printf("%d",32);
```

Example (Control flow)

```
// exception effect
throw new Exception();
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

```
int *p = ...
... // read and write
*p = *p + 1;
```

Example (IO)

```
// console IO
c = getchar();
printf("%d",32);
```

Example (Non-termination)

```
// infinite loop
while (1) {};
```

Example (Control flow)

```
// exception effect
throw new Exception();
```

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function. *Internal* effects are not observable from outside.

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function. *Internal* effects are not observable from outside.

Example (External effects)

Console, file and network I/O; termination and non-termination; non-local control flow (exceptions); etc.

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function. *Internal* effects are not observable from outside.

Example (External effects)

Console, file and network I/O; termination and non-termination; non-local control flow (exceptions); etc.

Are memory effects *external* or *internal*?

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function. *Internal* effects are not observable from outside.

Example (External effects)

Console, file and network I/O; termination and non-termination; non-local control flow (exceptions); etc.

Are memory effects *external* or *internal*?

Answer: Depends on the scope of the memory being accessed. Global variable accesses are *external*.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a complete mapping from the domain type `a` to the codomain type `b`.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a complete mapping from the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a complete mapping from the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.
- *Only* the function's return value is observable.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a complete mapping from the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.
- *Only* the function's return value is observable.
- Evaluation order becomes irrelevant.

The Danger of Side Effects

- Introduces (subtle) requirements on the evaluation order.

The Danger of Side Effects

- Introduces (subtle) requirements on the evaluation order.
- They are not visible from the type signature of the function.

The Danger of Side Effects

- Introduces (subtle) requirements on the evaluation order.
- They are not visible from the type signature of the function.
- They introduce non-local dependencies which is bad for software design, increasing *coupling*.

The Danger of Side Effects

- Introduces (subtle) requirements on the evaluation order.
- They are not visible from the type signature of the function.
- They introduce non-local dependencies which is bad for software design, increasing *coupling*.
- They interfere badly with strong typing (cf. arrays in Java)

The Danger of Side Effects

- Introduces (subtle) requirements on the evaluation order.
- They are not visible from the type signature of the function.
- They introduce non-local dependencies which is bad for software design, increasing *coupling*.
- They interfere badly with strong typing (cf. arrays in Java)

We can't, in general, reason equationally about effectful programs!

Problem: Equational Reasoning

Equational reasoning *fails* in the presence of impure functions.

- $x - x = 0$ is true for all integer expressions.

Problem: Equational Reasoning

Equational reasoning *fails* in the presence of impure functions.

- $x - x = 0$ is true for all integer expressions.
- ...but `getInt() - getInt() == 0` is nonsense. What if I input two different integers?

Monads as the Solution

Haskell faced *the I/O problem*. You can't have both of these:

1. Equational reasoning.

Monads as the Solution

Haskell faced *the I/O problem*. You can't have both of these:

1. Equational reasoning.
2. Functions with side effects.

Monads as the Solution

Haskell faced *the I/O problem*. You can't have both of these:

1. Equational reasoning.
2. Functions with side effects.

Monads

Monads are mathematical structures that were introduced by French mathematician **Roger Godement** in 1950. They come from *category theory*, which we're not learning here.

In Oct 1992, **Simon Peyton Jones** and **Philip Wadler** showed how to use monads to do I/O *without* sacrificing purity in Haskell-like languages. The Haskell community went on to apply monads to many other system design problems.

Monads as the Solution

Haskell faced *the I/O problem*. You can't have both of these:

1. Equational reasoning.
2. Functions with side effects.

Monads

Monads are mathematical structures that were introduced by French mathematician **Roger Godement** in 1950. They come from *category theory*, which we're not learning here.

In Oct 1992, **Simon Peyton Jones** and **Philip Wadler** showed how to use monads to do I/O *without* sacrificing purity in Haskell-like languages. The Haskell community went on to apply monads to many other system design problems.

The next 3 lectures: building up to understand SPJ and PW's solution to the I/O problem.

Scenario I

We will **not** introduce monads in this lecture. However, we will perform some system design tasks that hint at their existence.

Getting stuff from a DB

Imagine we have a database full of employee records:

```
data Employee = Employee
  { idNumber :: ID
  , name :: String
  , supervisor :: Maybe ID
  } deriving (Show, Eq)
```

Each employee has a unique id number, a name, and possibly a supervisor.

Scenario I

We have a search field, where the user can type an ID. When the user presses the `Search` button, the system should output the record of the **supervisor** of the employee with the given ID (if any).


ID: 23

```
Output: The supervisor of employee #23 is ...
```

State Passing

Example (Labeling Nodes)

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf
```

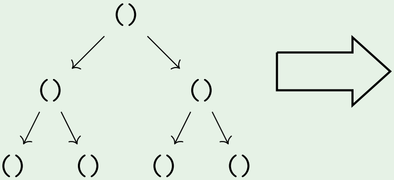
State Passing

Example (Labeling Nodes)

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf
```

Given a tree, label each node with an ascending number, by labeling the left subtree first.

```haskell
label :: Tree a -> Tree Integer
```



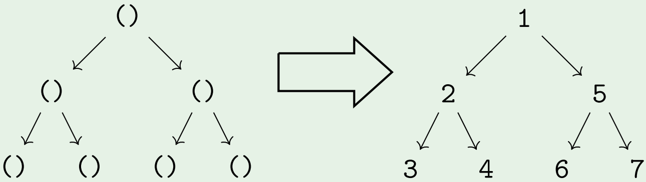
State Passing

Example (Labeling Nodes)

```
data Tree a = Node a (Tree a) (Tree a) | Leaf
```

Given a tree, label each node with an ascending number, by labeling the left subtree first.

```
label :: Tree a -> Tree Integer
```



Bind for State

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expressed as a function:

```
s -> (s, a)
```

Bind for State

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expressed as a function:

```
s -> (s, a)
```

Rather than change the state, we return a new copy of the state.

State Implementation

The Haskell standard library has a `State` type that is essentially implemented as the same state-passing we did before! But note that we had a `type` synonym, whereas they have a bona fide `data` type.

```
data State s a = State (s -> (s,a))
```

Caution

In the Haskell standard library `mtl`, the `State` type is actually implemented slightly differently, but the implementation essentially works the same way.

State

State Operations

```
get :: State s s
put :: s -> State s ()
return :: a -> State s a  -- our yield
evalState :: State s a -> s -> a
```

State

State Operations

```
get :: State s s
put :: s -> State s ()
return :: a -> State s a  -- our yield
evalState :: State s a -> s -> a
```

Bind

```
-- our bindS is declared infix
(>>=) :: State s a -> (a -> State s b) -> State s b
-- usage (implements the `use` fn):
get         >>= \x ->
put (x+1) >>= \_ ->
return x
```

Higher Kinds

Types and Values

Haskell is actually comprised of two languages in a layered cake:

- The *value level*, with `if`, `let`, `3` etc.

Types and Values

Haskell is actually comprised of two languages in a layered cake:

- The *value level*, with `if`, `let`, `3` etc.
- The *type level*, consisting of types `Int`, `Bool`, synonyms like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]` etc.

Types and Values

Haskell is actually comprised of two languages in a layered cake:

- The *value level*, with `if`, `let`, `3` etc.
- The *type level*, consisting of types `Int`, `Bool`, synonyms like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]` etc.

The type level also has a type system!

Kinds

Just as value-level terms are assigned types, terms on the type level are assigned *kinds*.

The most basic kind is written as $*$.

- Types such as `Int` and `Bool` have kind $*$. These are called *nullary types* (because they take no type parameters).

Kinds

Just as value-level terms are assigned types, terms on the type level are assigned *kinds*.

The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`. These are called *nullary types* (because they take no type parameters).
- `Maybe` take one parameter, so it has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`). This makes `Maybe` a *unary type*.

Kinds

Just as value-level terms are assigned types, terms on the type level are assigned *kinds*.

The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`. These are called *nullary types* (because they take no type parameters).
- `Maybe` take one parameter, so it has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`). This makes `Maybe` a *unary type*.
- There are binary types etc. But there are also *higher-kinded types* such as `(* -> *) -> *`. We won't deal with them now.

Lists

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function to give us some numbers:

```
getNumbers :: Seed -> [Int]
```

How can I compose `toString` with `getNumbers` to get a function `f` of type `Seed -> [String]`?

Lists

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function to give us some numbers:

```
getNumbers :: Seed -> [Int]
```

How can I compose `toString` with `getNumbers` to get a function `f` of type `Seed -> [String]`?

Answer: we use `map`:

```
f = map toString . getNumbers
```

Maybe

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function that may give us a number:

```
tryNumber :: Seed -> Maybe Int
```

How can I compose `toString` with `tryNumber` to get a function `f` of type `Seed -> Maybe String`?

Maybe

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function that may give us a number:

```
tryNumber :: Seed -> Maybe Int
```

How can I compose `toString` with `tryNumber` to get a function `f` of type `Seed -> Maybe String`?

We want something like a `map` function but for the Maybe type:

```
f = maybeMap toString . tryNumber
```

Functor

All of these functions are captured by the type class `Functor`.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor

All of these functions are captured by the type class `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous type classes we've seen like `Ord` and `Semigroup`, `Functor` is over types of kind `* -> *`.

Functor

All of these functions are captured by the type class `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous type classes we've seen like `Ord` and `Semigroup`, `Functor` is over types of kind `* -> *`.

Instances for:

- Lists
- `Maybe`
- `Gen` (QuickCheck generators)
- Functions (how?)

...and many more

Functor Laws

The functor type class must obey two laws:

Functor Laws

1. `fmap id x == x`
2. `fmap f (fmap g x) == fmap (f . g) x`

Functor Laws

The functor type class must obey two laws:

Functor Laws

1. `fmap id x == x`
2. `fmap f (fmap g x) == fmap (f . g) x`

(In Haskell, it's impossible to write a total `fmap` function that satisfies the first law but not the second. This follows from something called *parametricity*, which is beyond the scope of the course.)

FIN

1. **Assignment 1**: due on Sunday, 02 July 2023.
2. Last week's quizzes and exercises are due one Thursday.
3. This week's quizzes and exercises are due **after** flex week.