COMP3141

Software System Design and Implementation

Lecture 5: Effects, State Management

Zoltan A. Kocsis
University of New South Wales
Term 2 2022

Announcements

Assignment 1: due July 3.

Warning

That is a Sunday. But support from James and I may not be available (or be sparse) over the weekend. Plan accordingly!

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.
- Dealt with: data transformations, algorithm implementation in mathematically structured programs.

Midway

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

Where are we headed?

- Toward larger-scale mathematically structured system design.
- Previous focus: **data** structures.
- New focus: **control** structures.
- To reach the level of an adept Haskell programmer, you have to 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.

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.

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 mapping from all elements of the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.
- No observable trace is left beyond the result of the function.
- No implicit notion of time or order of execution.

The Danger of Side Effects

- They introduce (often 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, for example mutable arrays in Java, or reference types in ML.

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

Problem: Equational Reasoning

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

- Imagine we allowed functions with side effects in Haskell, for example, `getInt :: Int`, which prompts the user for an integer input, then returns whatever the user keyed in.
- Since $x - x = 0$ is true for all integers, equational reasoning says that if `x :: Int`, then we can replace `x - x` with `0` *without changing the meaning of our program*.
- But `getInt :: Int`, so we get `getInt - getInt == 0`, which is nonsense (imagine what happens if I input two different integers).

Monads as the Solution

Haskell faced a 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*, a fairly abstract field of mathematics.
In Oct 1992, **Simon Peyton Jones** and **Philip Wadler** presented a new model, based on monads, for performing input and output in pure functional languages such as Haskell. The Haskell community went on to apply monads to many system design problems in functional programming.

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).

`Output: The supervisor of employee #23 is ...` **Demo: live coding Maybe**

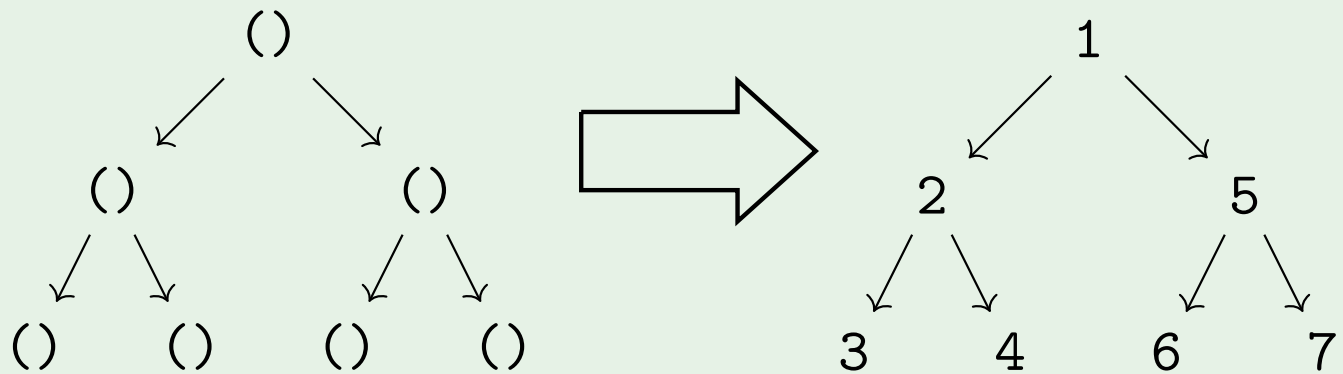
State Passing

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
```



Demo: Labeling Nodes

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
```

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.

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

This type level language itself has a type system!

Kinds

Just as terms in the value level language are given types, terms in the type level language are given *kinds*.
The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`. These are called *nullary types*.
- Seeing as `Maybe` is parameterised by one argument, `Maybe` 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 these for 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]`?

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`?

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

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

Demo: maybeMap implementation

Functor

All of these functions are in the interface of a single type class, called `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`
- Functions (how?)

Demo: fmap for Functions

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's type system it's impossible to make a total `fmap` function that satisfies the first law but violates the second. But this is a surprisingly deep theorem, proved using something called *parametricity*.

FIN

1. **Assignment 1**: due on Sunday, 03 July 2022.
2. Last week's quiz is due 23:59 Thursday, 30 June 2022.
3. Last week's exercise is due 09:10 Thursday, 30 June 2022.
4. This week's stuff is due **after** flex week.