Advanced Programming

More monads
(Preliminary version)

Fritz Henglein

Datalogisk Institut, Københavns Universitet (DIKU)

About me

Fritz Henglein

Professor of Programming Languages and Systems
University of Copenhagen

Head of Research
Deon Digital AG

Areas of interest
- Programming language technology
- Theoretical computer science (algorithms, semantics, logic)
- Blockchain technology
- Contract management
- Financial technology
- Enterprise systems

Related background

- European Blockchain Consortium (ebcc.eu, CPH), European Blockchain Institute (NRW)
- Steering committee chair, Innovation network for Finance IT (CFIR.dk, 2014-2018)
- Founder of research groups: Decentralized Systems, Functional High-Performance Computing

Academic background, affiliations, guest positions

Goals

Week 3 (last week):

- monadic parsing;
- parsing combinator;
- applicative parsing;
- parsing of Arithmetic Programming Language (APL).

Week 4:

- monads revisited;
- free monads;
- monads for programming with side effects;
- monadic input/output;
- monadic exceptions;
- updatable references;
- free monads for factoring out side effects.

Monad, categorically

- Let $\mathcal{C}$ be category, e.g. Hask consisting of types and functions between them.
- A *monad* on $\mathcal{C}$ is a triple consisting of a *functor* $M : \mathcal{C} \to \mathcal{C}$, a *unit* (or *return*) natural transformation and a *join* (or *bind*) natural transformation that satisfy certain equalities.

Monads in programming (here: Haskell)

- Monad: A *functor* `m :: Type -> Type` with functions

```
class Functor m => Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

that satisfy the equational properties

```
join (return x) = x
join (fmap return c) = c
join (join c) = join (fmap join c)
```

Monads for compositional side-effecting programming

- Executing code has generally two effects:
  - Returning a *value*: The value is used in subsequently executed code (main effect);
  - *Side effects*: Other effects that may also impact the outcome of subsequently executed code such as inputting or outputting data, updating memory, sending/receiving messages from/to other processes, etc.
- Conceptually, executing code consists of two phases:
  1. perform all side effects;
  2. return the computed value.
- Code may be
  - purely functional: it has no side effects (its total effect is described by the return value);
  - purely imperative: is returns no value (its total effect is performed by side effects);
  - a mixture of both.
- Command/query separation: Separate basic interface to stateful object into purely functional and purely imperative code.

Computations

- A *computation* is a *value* that wraps code that, *if and only if* executed, performs all its side effects and finally returns a value.
  - Also called *thunk* or *parameterless function* in other contexts.
- Computations can be constructed *monadically*:
  - `type M(a)`: computations with side effects described by `M` and return values of type `a`.
  - `return v`: Perform no side effects. Return value `v`.
  - `c >>= f`: Perform the side effects of `c`, then the side effects of `f v` where `v` is the value returned by `c`. Return the value returned by `f v`.
- Key points:
  - Computations themselves are *values* that can be stored in lists, passed to functions, etc.
  - `>>=` (or, equivalently, `join`) is the *only way of composing computations* in monadically constructed computations.
  - No side effects are executed without another function (usually `run...`) that kicks off execution.
  - Computations constructed monadically satisfy the monadic equalities.

 Monads revisited

Example: State monad

- Monad `State s`: Update a state of type s (side effects) and return values of any type.
- Prototypical type of computation: Update a state. Return a value.

```haskell
newtype State s a = State (s -> (a, s))

instance Functor (State s) where
  fmap f (State g) = State $ \s ->
    let (a, s') = g s
    in  (f a, s')
instance Applicative (State s) where
  pure a = State $ \s -> (a, s)
  (State sf) <*> (State sa) = State $ \s ->
    let (f, s')  = sf s
        (a, s'') = sa s'
    in  (f a, s'')
```

Example: State monad

```haskell
-- newtype State s a = State (s -> (a, s))

instance Monad (State s) where
  return = pure
  (State sa) >>= f = State $ \s ->
    let (a, s')       = sa s
        (State sb)    = f a
    in  sb s'

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> ((), s)
```

Monads as abstract data types

- A *concrete monad* is a parameterized *abstract data type* `M a` with the *monad interface*

```
fmap   :: (a -> b) -> (M a -> M b)
return :: a -> M a    -- alternative name: pure
(>>=)  :: M a -> (a -> M b) -> M b  -- or join
```

  satisfying the monad equalities (on a previous slide).
- It has typically additional operations, e.g. the parameterized *state monad* `M = State s` has

```
type State s a
get :: State s s
put :: s -> State s ()
runState :: M s a -> s -> (a, s)
```

  satisfying additional equalities, e.g.

```
put x >>= \() -> get   =   return x
```

Abstracting monads

- We have seen a purely functional implementation of the state monad where

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

- What if we want to change the implementation the state monad?
  - to eventually make it more efficient, but also
  - keep purely functional implementation for rapid development, stubbing, reference implementation and regression testing?

Free monads

- Free monad over functor `e :: Type -> Type`:

```
data Free e a
  = Pure a
  | Free (e (Free e a))
```

- Basic idea: Defer implementation of primitive monad operations.
  - Replace concrete monad actions (e.g. get, put in the state monad) by value *constructors* (Get, Put).
  - Interpret the constructors in a concrete monad by a separate interpretation function: `interpret :: e a -> M a`.
  - Use monad `T a = Free a e` parameterized with e to monadically composes abstracted actions.
  - Evaluate by interpreting the abstracted actions in the desired monad: `eval :: T a -> M a`

(To be continued.)