

Advanced Programming

More monads

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* $M : \mathcal{C} \rightarrow \mathcal{C}$, 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* $m :: \text{Type} \rightarrow \text{Type}$ with functions

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

that satisfy the equational properties

```
1 join (return x) = x  
2 join (fmap return c) = c  
3 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: it 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.

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.

```
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

```
-- 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*

```

1 fmap    :: (a -> b) -> (M a -> M b)
2 return  :: a -> M a      -- alternative name: pure
3 (>>=)   :: 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* $M = \text{State } s$ has

```

1 type State s a
2 get  :: State s s
3 put  :: s -> State s ()
4 runState :: M s a -> s -> (a, s)

```

satisfying additional equalities, e.g.

```

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

```

Abstracting monads

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

```
1 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 $e :: \text{Type} \rightarrow \text{Type}$:

```
1 data Free e a
2   = Pure a
3   | 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: $\text{interpret} :: e\ a \rightarrow M\ a$.
 - Use monad $T\ a = \text{Free}\ a\ e$ parameterized with e to monadically compose abstracted actions.
 - Evaluate by interpreting the abstracted actions in the desired monad: $\text{eval} :: T\ a \rightarrow M\ a$

State monads, abstractly

- State(-like) monads over states of type `s` as a type class:

```
1 class (Monad m) => StateMonad m s where  
2   put :: s -> m ()  
3   get :: m s
```

- The functional state monad as a specific instance:

```
1 instance StateMonad (FState s) s where  
2   put s = FState (const (()), s)  
3   get = FState dup where dup x = (x, x)
```

- `StateF` (*functional state monad*) is named `State` in the notes.

The free state monad

- The functor for constructing the free state monad:

```
1 data StateF s a
2   = Put s a           -- iso with (s, () -> a)
3   | Get (s -> a)      -- iso with ((), s -> a)
4 instance Functor (StateF s) where
5   fmap f (Put s a) = Put s (f a)
6   fmap f (Get g)   = Get $ f . g
```

- The free state monad (free monad over the state functor):

```
1 type FreeState s a = Free (StateF s) a
2 instance StateMonad (Free (StateF s)) s where
3   put s = Free (Put s (Pure ()))
4   get = Free (Get Pure)
```

- Where does StateF come from?

- Note: Different names used here than in lecture notes.

Constructing the functor: Basic idea

- A monadic computation consists of a *sequence* of actions. E.g.

```
1      get >>=  
2  \x -> (... put e1 >=  
3  \y -> (... get >>=  
4  \z -> (... get >>=  
5  \v -> (... return e2))))
```

- For state monads:

```
1      get >>=  
2  \x -> (... put e1 >>=  
3  \() -> (... get >>=  
4  \z -> (... get >>=  
5  \v -> (... return e2))))
```

since `put :: m ()`.

Constructing the functor: Basic idea

- Replace the concrete actions with constructors; give the second argument of `get` as an argument.

```
1  Get (\x -> (...
2  Put e1 (\() -> (...
3  Get (\z -> (...
4  Get \v -> (...
5  Pure e2))))
```

- Look at the types of the constructors:

```
1  Get: ((), (s -> a)) -- is eventual answer type
2  Put: (s, () -> a))
```

- Isomorphic to

```
1  Get: (s -> a)) -- is eventual answer type
2  Put: (s, a)
```


Constructing the functor: Basic idea

■ Constructors of state functor

```
1 Get: (s -> a)) -- is eventual answer type  
2 Put: (s, a)
```

■ State functor

```
1 data StateF s a  
2   = Put s a           -- iso with (s, () -> a)  
3   | Get (s -> a)      -- iso with ((), s -> a)
```

Imperative state monad

- Construct an *imperative state monad*
- Use an updatable reference to store the current state.
 - We can use `IORef` s, which “live” in the Haskell **IO** monad.
- See source code.

Summary

- Encapsulating side effects by abstracting them as computation monads.
 - Allows “plugging in” different side effects without changing source code.
- Two fundamental methods:
 - Parameterization over monads. Code pattern:
 $g :: (\text{Monad } m) \Rightarrow - \rightarrow T \rightarrow m \ U.$
 - Free monad. Code pattern: Construct state functor
 $F :: \text{Type} \rightarrow \text{Type}$ with constructors for primitive monadic operations. Define free monad `Free F`. Provide
 $\text{interp} :: F \ a \rightarrow M \ a$ for mapping constructors to monadic operations in a specific monad `M`. Optimize (inline code, automatically or by hand).
- Concrete Haskell monads in action: `IO`, `Exception`.
 - See lecture notes.