# ☺ Effektsysteme ☺

Ingo Blechschmidt
<iblech@speicherleck.de>

5. November 2015
Augsburger Curry-Club

# Gewünschte Instruktionen

```
type St = ...
data StateI :: * -> * where
    Get :: StateI St
    Put :: St -> StateI ()
```

Freie Funktoren und freie Monaden liefern ein allgemeines Konstruktionsrezept für Monaden mit gewünschter operationeller Semantik.

# Gewünschte Instruktionen

```
type St = ...
data StateI :: * -> * where
    Get :: StateI St
    Put :: St -> StateI ()

type Env = ...
data ReaderI :: * -> * where
    Ask :: ReaderI Env

type Log = ...
data WriterI :: * -> * where
    Tell :: Log -> WriterI ()
```

# Gewünschte Instruktionen

```
type Err = ...
data ErrorI :: * -> * where
    Throw :: Err -> ErrorI a

data IOI :: * -> * where
    PutStrLn :: String -> IOI ()
    GetLine  :: IOI String
    Exit     :: IOI a
```

# Freie Funktoren

Wir können aus einem Typkonstruktor `t :: * -> *` auf unspektakulärste Art und Weise einen Funktor machen:

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

data FreeF t a = MkFreeF (exists r. (t r, r -> a))
data FreeF t a = forall r. MkFreeF (t r) (r -> a)
-- MkFreeT :: t r -> (r -> a) -> FreeF t a

liftF :: t a -> FreeF t a
liftF x = MkFreeF x id

instance Functor (FreeF t) where
    fmap phi (MkFreeF x h) = MkFreeF x (phi . h)
```

Wenn `t` ein Funktor *wäre*, so könnte man eine Funktion `r -> a` zu einer Funktion `t r -> t a` liften. Wenn `t` kein Funktor ist, geht das nicht.

Der freie Funktor `FreeF t` über `t` *cheatet*: Ein Wert vom Typ `Free t a` besteht aus einem Wert `x :: t r` zusammen mit einer Funktion `f :: r -> a`. Anschaulich stellen wir uns diese Kombination als das vor, was `fmap f x` ergäbe, wenn `t` ein Funktor wäre.

Die Implementierung von `fmap phi` für `FreeF t` ist dann rein *formal*: Anstatt `phi` wirklich auf einen Wert vom Typ `t r` anzuwenden (was unmöglich ist), notieren wir uns nur die Information, dass `phi` anzuwenden *wäre*.

Das Paar bestehend aus dem *freien Funktor `FreeF t` über `t`* und der Funktion `liftF` erfüllt folgende *universelle Eigenschaft*:

Ist `G` irgendein Funktor und `phi :: t a -> G a` irgendeine Funktion, so existiert genau eine Funktion `phi’ :: FreeF t a -> G a` mit `phi’ . liftF = phi`.

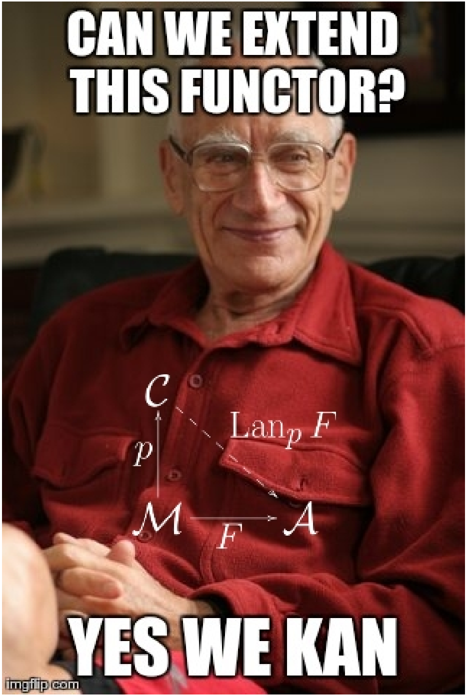
$$
\begin{tikzcd}
\mathtt{t} \arrow[rr, Rightarrow, "\mathtt{phi}"] \arrow[dr, Rightarrow, "\mathtt{liftF}"'] & & G \\
& \mathtt{FreeF\ t} \arrow[ur, Rightarrow, dashed, "\mathtt{phi'}"'] &
\end{tikzcd}
$$

Einen Typkonstruktor `t :: * -> *` kann man auch als einen Funktor $\mathrm{Hask}_0 \to \mathrm{Hask}$ auffassen. Dabei hat die Kategorie $\mathrm{Hask}_0$ dieselben Objekte wie Hask, enthält aber nur Identitätsmorphismen.

Der freie Funktor über `t` ist dann die *Links-Kan-Erweiterung* von `t` längs der Inklusion $\mathrm{Hask}_0 \hookrightarrow \mathrm{Hask}$. Die angegebene Definition von `FreeF t` ist nichts anderes als die *Koendeformel* für Links-Kan-Erweiterungen.

$$\begin{tikzcd}
\mathrm{Hask} \arrow[dr, dashed, "\texttt{FreeF t}"] & \\
\mathrm{Hask}_0 \arrow[u, hook] \arrow[r, "\texttt{t}"] & \mathrm{Hask}
\end{tikzcd}$$

$$\texttt{FreeF t a} = \int^{r \in \mathrm{Hask}_0} (\texttt{t r}, \texttt{r -> a})$$


CAN WE EXTEND THIS FUNCTOR?
$\mathcal{C}$
$\mathrm{Lan}_p\, F$
$p$
$\mathcal{M}$
$F$
$\mathcal{A}$
YES WE KAN
imgflip.com

# Beispiel: Zustand

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

data FreeF t a = forall r. MkFreeF (t r) (r -> a)

data StateI :: * -> * where
    Get :: StateI St
    Put :: St -> StateI ()

-- FreeF StateI ist isomorph zu:
data StateF a = Get (St -> a) | Put St a
```

# Freie Monaden

Wir können aus einem Funktor `f :: * -> *` auf unspektakulärste Art und Weise eine Monade machen:

```
data FreeM f a = Pure a | Roll (f (FreeM f a))

instance (Functor f) => Monad (FreeM f) where
    return x = Pure x

    Pure x >>= k = k x
    Roll u >>= k = Roll (fmap (>>= k) u)
```

Mehr zu freien Monaden in einem separaten Foliensatz:

`http://curry-club-augsburg.de/posts/2015-08-14-ankuendigung-siebtes-treffen.html`

# Zusammengesetzt

```
data FreeF t a = forall r. MkFreeF (t r) (r -> a)
data FreeM f a = Pure a | Roll (f (FreeM f a))

-- Also ist ‘FreeM (FreeF t) a‘ isomorph zu:
data Prog t a =
    Pure a | forall r. Step (t r) (r -> Prog t a)

data StateI :: * -> * where
    Get :: StateI St
    Put :: St -> StateI ()
-- ‘Prog StateI a‘ ist isomorph zu:
data StateProg a
    = Pure a
    | Get    (St -> StateProg a)
    | Put St (StateProg a)
```

Auch ohne die Motivation über freie Funktoren und freie Monaden besitzt `Prog` eine anschauliche Bedeutung. Wir können uns einen Wert vom Typ `Prog t a` als eine Folge von Instruktionen vorstellen, die schlussendlich einen Wert vom Typ `a` produziert. Welche Instruktionen vorkommen können, entscheidet `t`.

Ein Wert vom Typ `Prog t a` ist entweder von der Form `Pure x`, also ein triviale Folge von Instruktionen mit Produktionswert `x`, oder von der Form `Step u k`. Dabei kodiert `u` eine Instruktion, die bei Ausführung einen Wert `x :: r` produziert, und die Continuation `k :: r -> Prog t a` gibt an, wie es danach weitergehen soll.

# Operationelle Semantik

```
data StateProg a
    = Pure a
    | Get    (St -> StateProg a)
    | Put St (StateProg a)

interpret :: StateProg a -> (St -> (a,St))
interpret (Pure x)     st = (x, st)
interpret (Get  k)     st = interpret (k st) st
interpret (Put  st' u) st = interpret u      st'
```

# In Kürze

```haskell
-- Die freie Monade über dem freien Funktor über t:
data Prog t a =
    Pure a | forall r. Step (t r) (r -> Prog t a)
```

- Werte vom Typ `Prog t a` sind rein syntaktische Beschreibungen von Aktionsfolgen. Insbesondere gelten keinerlei besondere Rechenregeln, wie zum Beispiel `Put x (Put x' m) == Put x' m`.
- Erst durch Angabe eines Interpreters wird die Konstruktion zum Leben erweckt.
- Man kann leicht Instruktionsspezifikationen miteinander kombinieren!
- In der naiven Implementierung: Effizienzproblem mit linksassoziativer Verwendung von `(>>=)`

Ein und dieselbe freie Monade kann mehrere verschiedene Interpreter zulassen. Zum Beispiel kann man in Produktion einen Interpreter verwenden, der eine Datenbank anspricht, und zum Testen einen, der fiktive Werte vortäuscht (Mocking).

Siehe unbedingt auch:

- Heinrich Apfelmus. *The Operational Monad Tutorial*. 2010.

  `http://apfelmus.nfshost.com/articles/operational-monad.html`

# Einschub: Koprodukt von Monaden

Sind `m` und `n` Monaden, so kann man eine Monade bauen, die die Fähigkeiten von `m` und `n` vereint. Diese heißt *Koprodukt* von `m` und `n`.

```
data Sum    m n a = Inl (m a) | Inr (n a)
type Coprod m n a = Prog (Sum m n) a

-- Coprod m n vereint m und n:
inl :: m a -> Coprod m n a     inr :: n a -> Coprod m n a
inl x = Step (Inl x) Pure      inr x = Step (Inr x) Pure

-- Ausführung mit (universelle Eigenschaft):
elim :: (Monad m, Monad n, Monad s)
     => (m a -> s a) -> (n a -> s a)
     -> (Coprod m n a -> s a)
elim = ...
```

# Beispiel: State s ∐ Error e

```haskell
type Err e = Either e

ex :: Coprod (State Int) (Err String) Int
ex = do
    st <- inl get
    if st <= 0 then inr (Left "Fehler") else do
    inl $ put (st - 1)
    return $ st^2 + st + 1
```

Ausführung durch Angabe einer Monade, in die man `State Int` und `Err String` einbetten kann – zum Beispiel `IO`:

```
runM :: (Show e) => s -> Coprod (State s) (Err e) a -> IO a
runM st m = do
    ref <- newIORef st
    elim (embedState ref) embedErr m

embedState :: IORef s -> State s a -> IO a
embedState ref m = do
    st <- readIORef ref
    let (x,st') = runState m st
    writeIORef ref st'
    return x

embedErr :: (Show e) => Err e a -> IO a
embedErr (Left  e) = putStrLn ("Fehler: " ++ show e) >> exit
embedErr (Right x) = return x
```

# Effektsysteme

Effektsysteme lösen das Problem der fehlenden Kompositionalität von Monaden.

Die Monade `Eff r` ist wie `Prog t`, nur

- performant bezüglich `(>>=)` und
- mit `r` als *Liste* von möglichen Instruktionen (auf Typebene) anstatt mit `t` als Instruktionen kodierenden *Typkonstruktor*.

```
ask :: (Member (Reader env) r) => Eff r env
get :: (Member (State st)   r) => Eff r st
put :: (Member (State st)   r) => st -> Eff r ()
-- Typen schreiben r nicht eindeutig vor!
```

Wenn man nachträglich seinen Transformerstack ändert, muss man viele Typsignaturen und gelegentlich auch einige Codefragmente anpassen (zum Beispiel `lift` in `lift . lift` ändern). Das ist mühsam.

Bei Verwendung von `Eff` muss man das nicht.

Werte vom Typ `Eff r a` sind wie bei `Prog t a` auch nur *Beschreibungen* von auszuführenden Aktionen; erst durch Angabe von Interpretern können sie ausgeführt werden. Interpreter können leicht miteinander kombiniert werden.

# Interpreter

```
type Env = ...
data Reader :: * -> * where
    Get :: Reader Env

ask :: (Member Reader r) => Eff r Env
ask = Roll (inj Get) (tsingleton Pure)

runReader :: Env -> Eff (Reader ': r) a -> Eff r a
runReader e m = loop m where
    loop (Pure x)   = return x
    loop (Roll u q) = case decomp u of
        Right Get -> loop $ qApp q e
        Left  u   -> Roll u (tsingleton (qComp q loop))
-- kürzer:
runReader e = handleRelay return (\Get k -> k e)
```

Siehe unbedingt:


- Oleg Kiselyov, Hiromi Ishii. *Freer Monads, More Extensible Effects.* 2015.

  `http://okmij.org/ftp/Haskell/extensible/more.pdf`

- Oleg Kiselyov, Amr Sabry, Cameron Swords. *Extensible Effects. An Alternative to Monad Transformers.* 2013.

  `http://okmij.org/ftp/Haskell/extensible/exteff.pdf`

- Andrej Bauer, Matija Pretnar. *Programming with Algebraic Effects and Handlers.* 2012.

  `http://arxiv.org/abs/1203.1539`