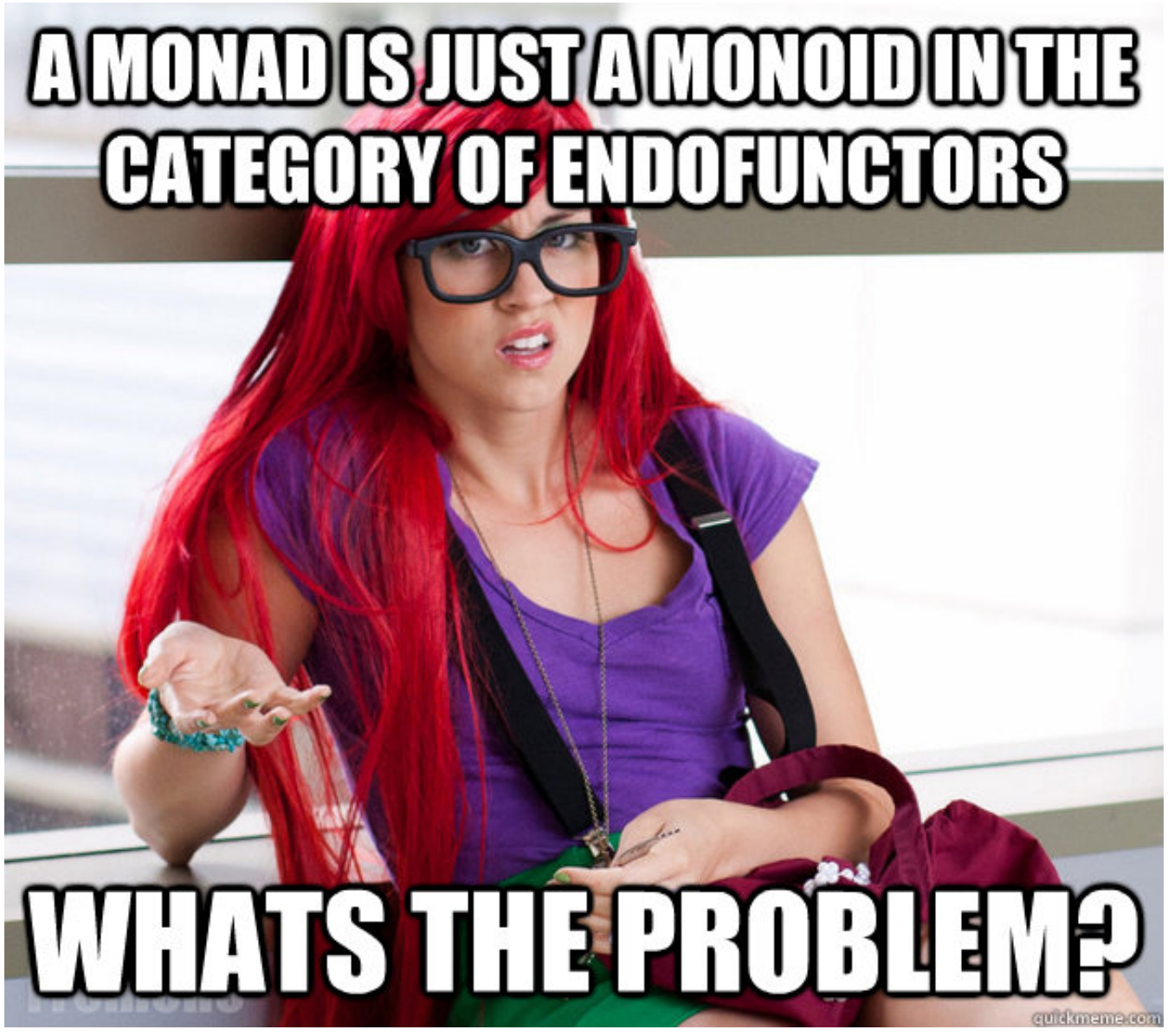
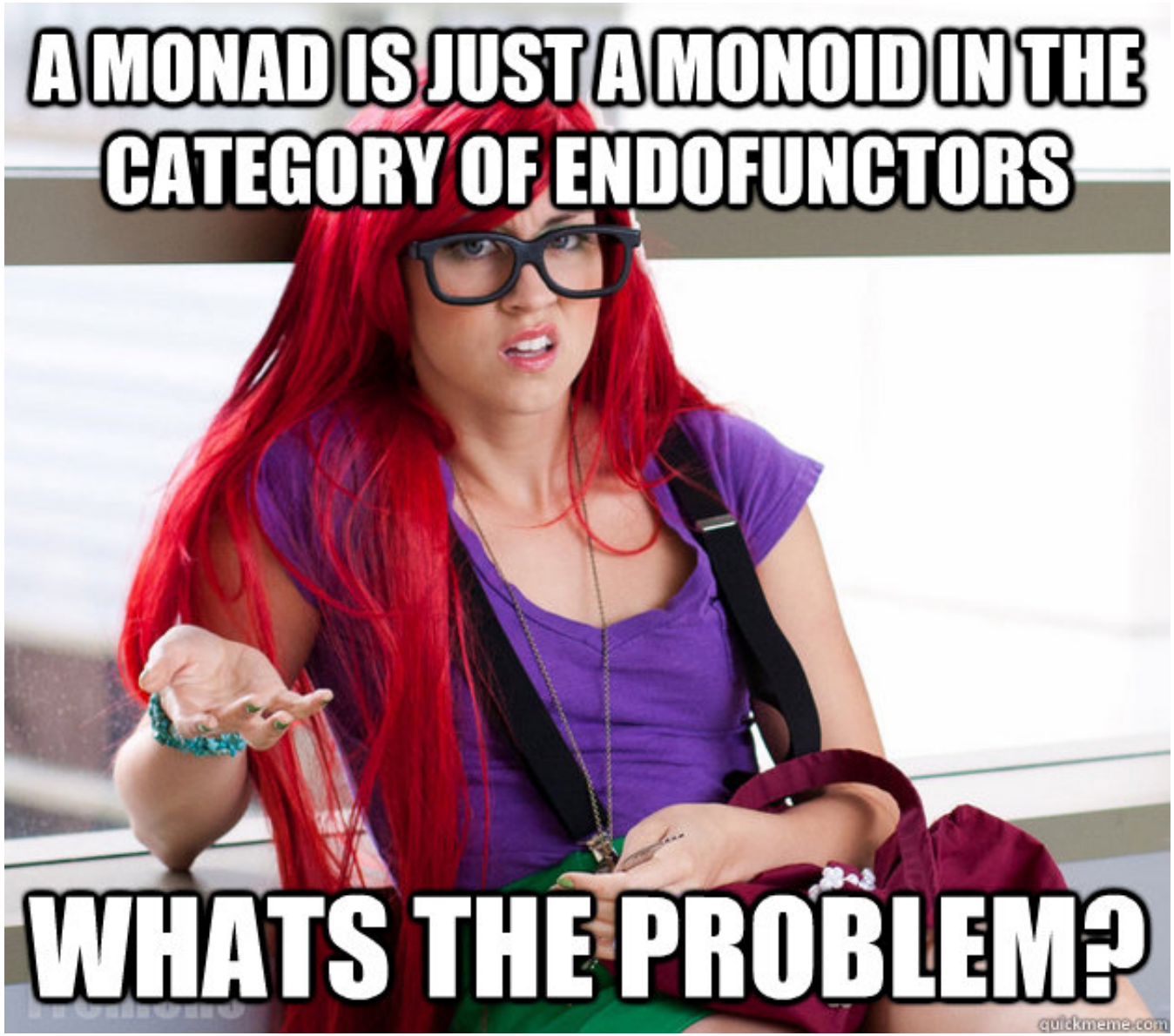
Monaden

Carsten König

DWX 2014

Monade . . . was ist das?


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHATS THE PROBLEM?
quickmeme.com



(jein) **. . . aber wen interessiert das?**

Wir interessieren uns für

C#

```
IEnumerable<Outfit> Outfits()
{
    return
        from schuhe in Schuhe
        from hose in Hosen
        from hemd in Hemden
        select new Outfit {Schuhe = schuhe, Hose = hose, He
}
```

Wir interessieren uns für

C#

```
IEnumerable<Outfit> Outfits()
{
    return
        from schuhe in Schuhe
        from hose in Hosen
        from hemd in Hemden
        select new Outfit {Schuhe = schuhe, Hose = hose, He
}
```

Monade: IEnumerable

oder

F#

```
let fetchAsync(url:string) : Async<string> =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            return! webClient.AsyncDownloadString(uri)
        with
            | ex -> printfn "Fehler: %s" ex.Message
    }
```

oder

F#

```
let fetchAsync(url:string) : Async<string> =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            return! webClient.AsyncDownloadString(uri)
        with
            | ex -> printfn "Fehler: %s" ex.Message
    }
```

Monade: `Async`

oder

Haskell

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

oder

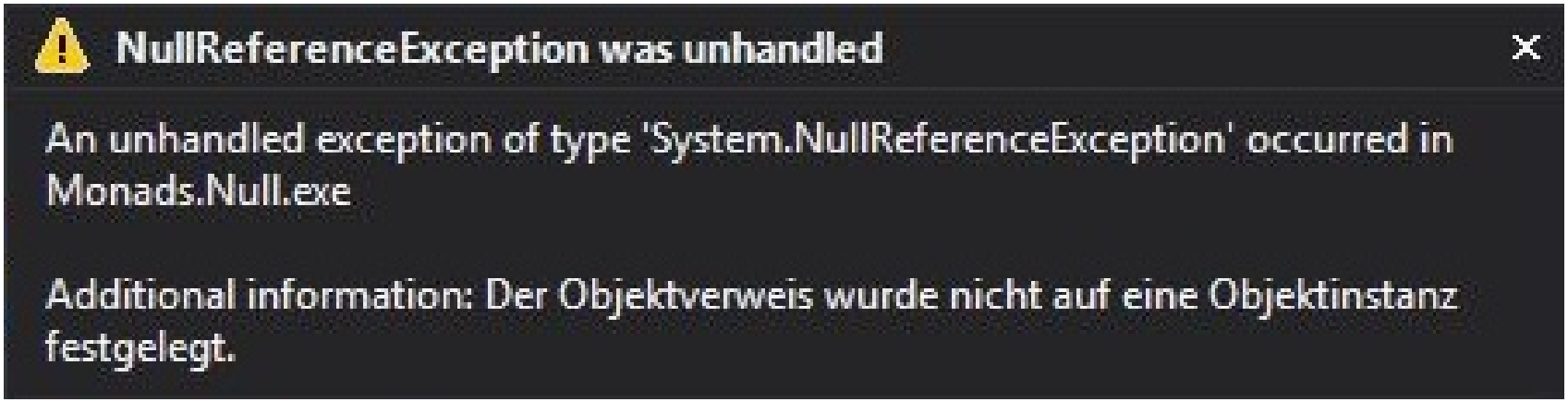
Haskell

```
main :: IO ()
main = putStrLn "Hello, World!"
```

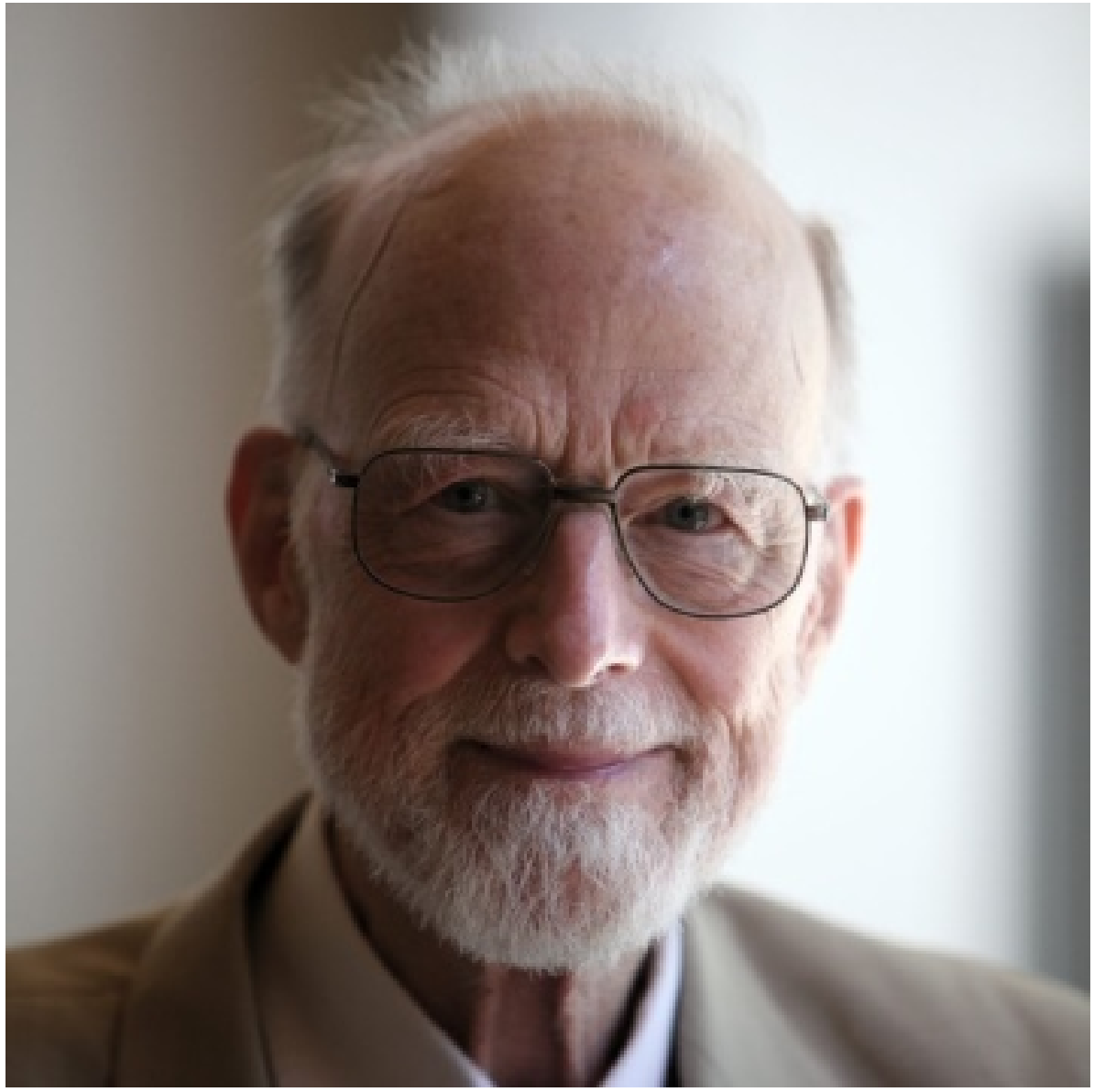
Monade: `IO`

Motivation

schon einmal gesehen?



I call it my billion-dollar mistake.
– Tony Hoare



übliche Lösung

guards

```
var customer = GetCustomer(5);
if (customer != null)
{
  var orders = GetOrdersFor(customer);
  if (orders != null)
  {
    // ...
  }
}
return null; // good idea isn't it?
```

ein Muster...

```
var customer = GetCustomer(5);
if (customer != null)
{
  var orders = GetOrdersFor(customer);
  if (orders != null)
  {
    // ...
  }
  else
  {
    return null;
  }
}
else
{
  return null;
}
```

refactor

```
public tR Guard<tS, tR>(tS source, Func<tS, tR> f)
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}
```

refactor

```
public tR Guard<tS, tR>(tS source, Func<tS, tR> f)
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}

return Guard(GetCustomer(5), customer =>
  Guard(GetOrdersFor(customer), order => {
    // ...
  }));
```

Syntax-Zucker

```
public static tR Guard<tS, tR>(this tS source, Func<tS, tR>
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}
```

Syntax-Zucker

```
public static tR Guard<tS, tR>(this tS source, Func<tS, tR>
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}

return GetCustomer(5).Guard(customer =>
  GetOrdersFor(customer).Guard(order => {
    // ...
  }));
```

können wir das eleganter lösen?

Maybe

Erfolg represäntieren

Wenn eine Funktion **partiell** ist, dann sollte sie es auch *zugeben*

Erfolg represäntieren

Wenn eine Funktion **partiell** ist, dann sollte sie es auch *zugeben*

```
Customer TryGetCustomer(int customerNumber)
```

Erfolg represäntieren

Wenn eine Funktion **partiell** ist, dann sollte sie es auch *zugeben*

```
Customer TryGetCustomer(int customerNumber)
```

davon *merkt* der Compiler nichts

nutze die Typen

Der Compiler kann mit **Datentypen** umgehen

nutze die Typen

Der Compiler kann mit **Datentypen** umgehen

```
Maybe<Customer> TryGetCustomer(int customerNumber)

Maybe<Orders> TryGetOrdersFor(Customer customer)
```

nutze die Typen

Der Compiler kann mit **Datentypen** umgehen

```
Maybe<Customer> TryGetCustomer(int customerNumber)

Maybe<Orders> TryGetOrdersFor(Customer customer)
```

```
type Maybe<'a> =
  | Just of 'a
  | Nothing
```

Hinweis: tatsächlich würde man `Option<'a>` verwenden

sieht nicht anders aus

```
var customer = TryGetCustomer(5);
if (customer.IsJust)
{
    var orders = TryGetOrdersFor(customer.Value);
    if (orders.IsJust)
    {
        // ... berechne irgendwas vom Typ Rückgabe
        return irgendwas;
    }
    else
    {
        return Maybe.Nothing<Rückgabe>();
    }
}
else
{
    return Maybe.Nothing<Rückgabe>();
}
```

nochmal: refactor

```
return Bind(TryGetCustomer(5), customer =>
  Bind(TryGetOrdersFor(customer), order => {
    // ...
  }));
```

```
public Maybe<tR> Bind<tS, tR>(Maybe<tS> source, Func<tS, Ma
```

gesucht: Implementation davon

let the TYPES guide you luke

```
Maybe<tR> Bind<tS, tR>(Maybe<tS> source, Func<tS, Maybe<tR>
```

```
Maybe<tR> Bind<tS, tR>(Maybe<tS> source, Func<tS, Maybe<tR>

{
  if (source.IsNothing) return Maybe.Nothing<tR>();
```

```
Maybe<tR> Bind<tS, tR>(Maybe<tS> source, Func<tS, Maybe<tR>

{
  if (source.IsNothing) return Maybe.Nothing<tR>();

  else return f(source.Value);
}
```

Zucker . . . mehr Zucker

C#: LINQ

```
return
from customer in TryGetCustomer(5)
from orders in TryGetOrdersFor(customer)
select ...
```

F#: Computation Expressions

```
maybe {
    let! customer = tryGetCustomer 5
    let! orders   = tryGetOrders customer
    return ...
}
```

Haskell: Do-Notation

```haskell
beispiel :: Maybe Orders
beispiel = do
  customer <- tryGetCustomer 5
  orders   <- tryGetOrdersFor customer
  return orders
```

was müssen wir dafür tun?

LINQ

SelectMany implementieren:

```
public static Maybe<tR> SelectMany<tS, tR>
  (this Maybe<tS> source,
   Func<tS, Maybe<tR>> selector)
{
    return Bind(source, selector);
}

public static Maybe<tR> SelectMany<tS, tCol, tR>
  (this Maybe<tS> source,
   Func<tS, Maybe<tCol>> collectionSelector,
   Func<tS, tCol, tR> resultSelector)
{
    if (source.IsNothing) return Nothing<tR>();
    var col = collectionSelector(source.Value);
    if (col.IsNothing) return Nothing<tR>();
    return resultSelector(source.Value, col.Value);
}
```

Builder implementieren:

```
type MaybeBuilder() =
    member x.Bind(m, f)     = bind m f
    member x.Return(v)      = Just v

let maybe = MaybeBuilder()
```

Haskell

`Monad` Typklasse instanzieren:

```haskell
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  Just a  >>= f = f a
```

List Monad

nicht-deterministische Ergebnise repräsentieren

. . . das ist nur ein komplizierter Ausdruck für *mehrere Ergebnisse gleichzeitig*

Motivation

`Maybe` gibt ein oder kein Ergebnis zurück.

Listen geben beliebig viele Ergebnise zurück!

was ist zu tun?

Listen `[a]` sollen zum Monaden werden

was ist zu tun?

Listen `[a]` sollen zum Monaden werden

Gesucht: Implementation von

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

und

```
return :: a -> [a]
```

Return

```
return :: a -> [a]
```

Intuition: einen Wert in eine Liste **packen**

Return

Haskell:

```
return :: a -> [a]
return x = [x]
```

F#:

```
let return (x : 'a) : 'a list = [x]
```

Bind

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Intuition:

- jedes Element gibt wieder eine Liste
- zusammen also eine Liste von Listen
- diese gilt es **plattzumachen**

Bind

Haskell:

```haskell
(>>=) :: [a] -> (a -> [b]) -> [b]
xs (>>=) f = concatMap f xs
```

F#:

```fsharp
let bind (xs : 'a list) (f : 'a -> 'b list) : 'b list =
  xs |> List.collect f
```

Beispiel

Permutation einer Liste/Aufzählung

Rezept:

- Nimm ein Element `x` heraus
- Nimm eine Permuation `xs` der restlichen Elemente
- `x:xs` ist eine Permutation
- Liste alle solchen Permuationen auf

F#

```
let ohne x =
    List.filter ((<>) x)

let rec permutationen = function
  | [] -> [[]]
  | xs -> [ for x in xs do
              let rest = xs |> ohne x
              for xs' in permutationen rest do
              yield (x::xs') ]
```

Haskell

```haskell
import Data.List (delete)

perm :: (Eq a) => [a] -> [[a]]
perm [] = [[]]
perm xs = do
  x <- xs
  xs' <- perm $ delete x xs
  return $ x:xs'
```

C#/LINQ

```
IEnumerable<a> Vor<a>(this a value, IEnumerable<a> rest)
{
    yield return value;
    foreach (var v in rest)
        yield return v;
}

IEnumerable<IEnumerable<a>> Permuationen<a>(IEnumerable<a>
{
    if (!items.Any()) return new[] {new a[] {}};

    return
        from h in items
        from t in Permuationen(items.Where(x => !Equals(x,
        select h.Vor(t);
}
```

Wahrscheinlich ein Monade

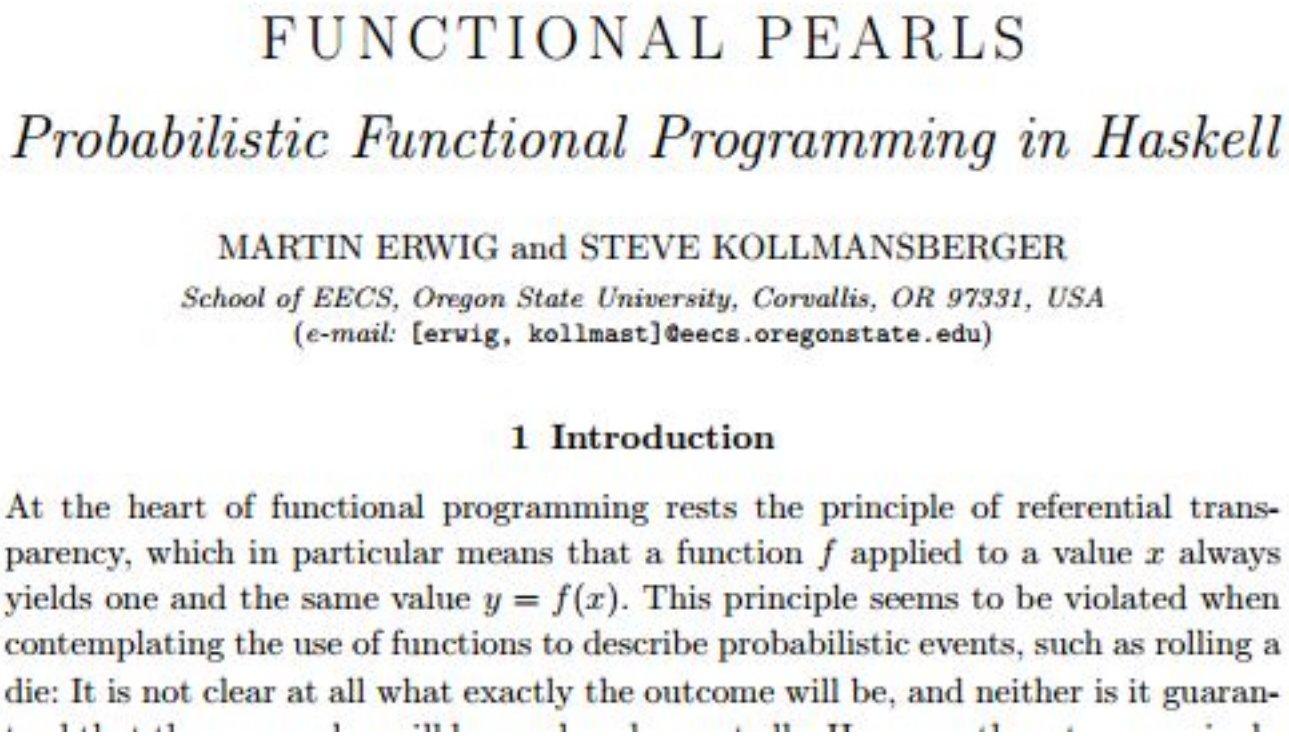
Ziel:

```
let würfel : Verteilung<int> =
    gleichVerteilung [1..6]

let rec nWürfel (n : int) : Verteilung<int list> =
    prob {
        if n <= 0 then return [] else
        let! w = würfel
        let! rest = nWürfel (n-1)
        return (w::rest)
    }

let ``WS: mindestens 2 Sechser in 4 Würfeln`` =
    let hatZweiSecher (augen : int list) : bool =
        augen |> List.filter ((=) 6)
              |> (fun l -> List.length l >= 2)
    nWürfel 4 >? hatZweiSecher
> 0.1319444444
```

inspieriert durch



FUNCTIONAL PEARLS

Probabilistic Functional Programming in Haskell

MARTIN ERWIG and STEVE KOLLMANSBERGER

School of EECS, Oregon State University, Corvallis, OR 97331, USA

(*e-mail:* [erwig, kollmast]@eecs.oregonstate.edu)

1 Introduction

At the heart of functional programming rests the principle of referential transparency, which in particular means that a function f applied to a value x always yields one and the same value $y = f(x)$. This principle seems to be violated when contemplating the use of functions to describe probabilistic events, such as rolling a die: It is not clear at all what exactly the outcome will be, and neither is it guaran-

Figure 1: Functional Pearls: Probabilistic Functional Programming in Haskell

FsCheck Generator

FsCheck?

- Testet **Eigenschaften** von Funktionen/Programmen automatisch.
- Eigenschaft: `prop : 'input -> bool`
- FsCheck generiert zufällige Eingaben und testet ob die Eigenschaft jeweils `true` liefert.

Doku/Info https://fsharp.github.io/FsCheck/QuickStart.html

Beispiel

```
let ``zweimal List.rev ist Identität``(xs : int list) =
    List.rev (List.rev xs) = xs

Check.Quick ``zweimal List.rev ist Identität``
> Ok, passed 100 tests.

let ``List.rev ist Identität``(xs : int list) =
    List.rev xs = xs

Check.Quick ``List.rev ist Identität``
> Falsifiable, after 3 tests (6 shrinks) (StdGen (12336011
> [1; 0]
```

Generatoren

Um die Testfälle zu generieren verwendet FsCheck *Generatoren*

```
type Würfel = internal W of int
let würfel n =
    if n < 1 || n > 6
    then failwith "ungültiger Wert"
    else W n

let würfelGen =
    gen {
        let! n = Gen.choose (1,6)
        return würfel n
    }
```

Definition

vereinfacht: Generator ist eine Aktion, die einen zufälligen Wert liefert

```
type Gen<'a> = Gen of (unit -> 'a)

let sample (Gen g : Gen<'a>) : 'a =
    g()
```

make it monadic

Gesucht: Implementation von

```
(>>=) : Gen<'a> -> ('a -> Gen<'b>) -> Gen<'b>
```

und

```
return : 'a -> Gen<'a>
```

Return

```
return : 'a -> Gen<'a> = 'a -> (unit -> 'a)
```

Intuition: zufällige Wahl aus einem Element ...

Return

```
return : 'a -> Gen<'a> = 'a -> (unit -> 'a)
```

Intuition: zufällige Wahl aus einem Element ...

```
let returnM (a : 'a) : Gen<'a> =
    Gen <| fun () -> a
```

Bind

```
bind : ('a -> Gen<'b>) -> Gen<'a> -> Gen<'b> =
     = ('a -> (unit -> 'b)) -> (unit -> 'a) -> (unit -> 'b)
```

Bind

```
bind : ('a -> Gen<'b>) -> Gen<'a> -> Gen<'b> =
     = ('a -> (unit -> 'b)) -> (unit -> 'a) -> (unit -> 'b)
```

F#:

```
let bind (f : 'a -> Gen<'b>) (g : Gen<'a>) : Gen<'b> =
    Gen <| fun () -> sample (sample g |> f)
```

Async

MSDN example

```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length
        with
            | ex -> printfn "%s" (ex.Message);
    }
```

MSDN example

```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length
        with
            | ex -> printfn "%s" (ex.Message);
    }
```

zu komplex

simplify

verkettete Continuations/Callback

Continuation

```
type Cont<'a>  = 'a -> unit
```

- Nebeneffekt behaftete Aktion
- wird mit dem Ergebnis einer (asynchronen) Berechnung aufgerufen

Continuation monad

```
type ContM<'a> = { run : Cont<'a> -> unit }
```

- gib mir eine 'a Continution c
- ich berechne ein 'a und rufe c damit auf
- kann verzögert/asychron erfolgen

Hilfsfunktionen

```
let mkCont (f : Cont<'a> -> unit) : ContM<'a> =
    { run = f }

let runWith (f : 'a -> 'b) (m : ContM<'a>) =
    m.run (f >> ignore)
```

example

```
let delay (span : TimeSpan) : ContM<unit> =
fun f ->
    let timer = new Timer()
    timer.Interval <- int span.TotalMilliseconds
    timer.Tick.Add (fun _ -> timer.Dispose(); f())
    timer.Start()
|> mkCont
```

make it monadic

```
let mReturn : 'a -> ContM<'a> = ...

let bind : ('a -> ContM<'b>) -> ContM<'a> -> ContM<'b> =
```

Return

```
let mReturn (a : 'a) : ContM<'a> =
    mkCont (fun f -> f a)
```

rufe die übergebene Continuation sofort mit a auf

Bind

```
let bind (f : 'a -> ContM<'b>) (m : ContM<'a>) : ContM<'b>
  fun (c : Cont<'b>) ->
      m.run <| fun a -> (f a).run c
  |> mkCont
```

- wenn eine Continuation `c` übergeben wird:
  - berechne erst den Effekt von `m`
  - nutze dessen Ergebnis um mit `f` ein `Cont<'b>` zu bekommen
  - berechne diese und gib das Ergebnis an `c` weiter

example

```
let verzögertesHallo() =
    cont {
        do! delay <| TimeSpan.FromSeconds 2.0
        return "Hello, World!"
    } |> runWith MessageBox.Show
```

State

Motivation

Berechnungen mit Zustand:

```
f :: zustand -> a -> (b, zustand)
g :: zustand -> b -> (c, zustand)
```

Motivation

Berechnungen mit Zustand:

```
f :: zustand -> a -> (b, zustand)
g :: zustand -> b -> (c, zustand)
```

Komposition:

```
let (b, z1) = f z0 a
let (c, _)  = g z1 b
c
```

unschön - geht das mit Monaden besser?

to much freedom

```
f :: zustand -> a -> (b, zustand)
```

- **Bisher:** immer ein generischer Parameter im Monaden
- **jetzt:** drei: `zustand`, `a` und `b`

to much freedom

```
f :: zustand -> a -> (b, zustand)
```

- **Bisher:** immer ein generischer Parameter im Monaden
- **jetzt:** drei: `zustand`, `a` und `b`

`zustand` *ändert* sich nicht - aber was ist mit den anderen?

functional trickster

ein wenig umformen:

```
f :: a -> (zustand -> (b, zustand))
```

hier ist ein Monade versteckt

functional trickster

ein wenig umformen:

```
f :: a -> (zustand -> (b, zustand))
```

hier ist ein Monade versteckt

Hint: `putStrLn :: String -> IO ()`

and the candiate is

Haskell:

```
data State s a = State { runState :: s -> (a, s) }
f :: a -> State zustand b
```

F#:

```
type State<'s,'a> = { runState : 's -> ('a*'s) }
let mkState f = { runState = f }
```

make it monadic

```
return :: a -> State s a
(>>=)  :: State s a -> (a -> State s b) -> State s b
```

Return

Haskell:

```
return :: a -> State s a
return a =  State $ \s -> (a, s)
```

F#

```
let return (a : 'a) : State<'s, 'a> =
  mkState <| fun s -> (a, s)
```

reiche den Zustand weiter und gib den Wert zurück

Bind

Haskell

```haskell
(>>=)  :: State s a -> (a -> State s b) -> State s b
State f >>= g =
    State $ \s -> let (a,s') = f s
                  in runState (g a) s'
```

F#

```fsharp
let bind (g : 'a -> State<'s,'b>) (f : State<'s,'a>) : Stat
    fun s ->
        let (a,s') = f.runState s
        (g a).runState s'
    |> mkState
```

Sample