Frank – eine Programmiersprache mit Effekten

Tim Baumann

Curry Club Augsburg

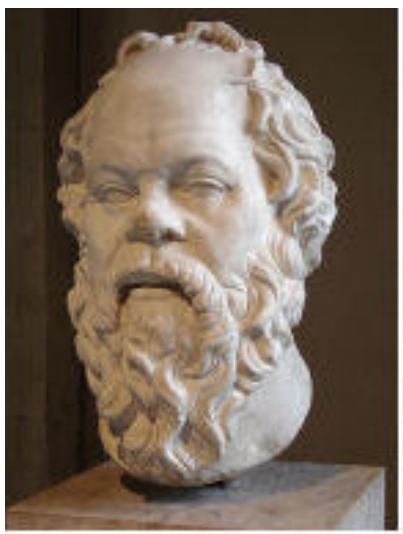
23. Februar 2017

Paper:	Do Be Do Be Do
	von Sam Lindley, Conor McBride und Craig McLaughlin
GitHub:	`https://github.com/cmcl/frankjnr`

„To be is to do“ – Socrates

„To do is to be“ – Sartre

„Do Be Do Be Do“ – Sinatra



Socrates

J. P. Sartre

Frank Sinatra

Berechnungen und Werte

Grundprinzip in Frank: Es werden

- Werte, die *sind*, und
- Berechnungen, die *tuen*,

voneinander getrennt.

Berechnungen und Werte

```
data List X = nil | cons X (List X)

map : {{X -> Y} -> List X -> List Y}
map f nil          = nil
map f (cons x xs) = cons (f x) (map f xs)
```

Berechnungen und Werte

```
data List X = nil | cons X (List X)

map : {{X -> Y} -> List X -> List Y}
map f nil          = nil
map f (cons x xs) = cons (f x) (map f xs)
```

- Regel: Typen von Berechnungen sind von der Form `{t1 -> ... -> tn -> r}` mit null oder mehr Argumenten.

Berechnungen und Werte

```
data List X = nil | cons X (List X)

map : {{X -> Y} -> List X -> List Y}
map f nil          = nil
map f (cons x xs) = cons (f x) (map f xs)
```

- Regel: Typen von Berechnungen sind von der Form `{t1 -> ... -> tn -> r}` mit null oder mehr Argumenten.
- Berechnungen werden unausgewertet übergeben, Werte ausgewertet übergeben. Argumente werden in der Reihenfolge von links nach rechts ausgewertet.

Berechnungen und Werte

```
data List X = nil | cons X (List X)

map : {{X -> Y} -> List X -> List Y}
map f nil          = nil
map f (cons x xs) = cons (f x) (map f xs)
```

- Regel: Typen von Berechnungen sind von der Form `{t1 -> ... -> tn -> r}` mit null oder mehr Argumenten.
- Berechnungen werden unausgewertet übergeben, Werte ausgewertet übergeben. Argumente werden in der Reihenfolge von links nach rechts ausgewertet.
- Berechnungen mit Argumenten führt man aus, indem man Argumente übergibt

Berechnungen und Werte

```
data List X = nil | cons X (List X)

map : {{X -> Y} -> List X -> List Y}
map f nil          = nil
map f (cons x xs) = cons (f x) (map f xs)
```

- Regel: Typen von Berechnungen sind von der Form `{t1 -> ... -> tn -> r}` mit null oder mehr Argumenten.
- Berechnungen werden unausgewertet übergeben, Werte ausgewertet übergeben. Argumente werden in der Reihenfolge von links nach rechts ausgewertet.
- Berechnungen mit Argumenten führt man aus, indem man Argumente übergibt
- Etwas Syntaxzucker: Wir werden die ganz äußeren Klammern bei Funktionsdefinitionen weglassen:

```
map : {X -> Y} -> List X -> List Y
```

Berechnungen und Werte

```
data Bool = tt | ff

if : Bool -> {X} -> {X} -> X
if tt t _ = t!
if ff _ f = f!
```

Berechnungen und Werte

```
data Bool = tt | ff

if : Bool -> {X} -> {X} -> X
if tt t _ = t!
if ff _ f = f!
```

- Regel: Eine nullstellige Berechnung `f : {X}` führt man mit einem Ausrufezeichen aus: `f! : X`

Berechnungen und Werte

```
data Bool = tt | ff

if : Bool -> {X} -> {X} -> X
if tt t _ = t!
if ff _ f = f!
```

- Regel: Eine nullstellige Berechnung `f : {X}` führt man mit einem Ausrufezeichen aus: `f! : X`

```
if fire! {launch missiles} {unit}
```

Berechnungen und Werte

```
data Bool = tt | ff

if : Bool -> {X} -> {X} -> X
if tt t _ = t!
if ff _ f = f!
```

- Regel: Eine nullstellige Berechnung `f : {X}` führt man mit einem Ausrufezeichen aus: `f! : X`

```
if fire! {launch missiles} {unit}
```

- Regel: Ist `x : X` ein Wert, so ist `{x} : {X}` die Berechnung, die diesen Wert produziert.

Berechnungen und Werte

Man kann Case-Style-Pattern-Matching in der Sprache definieren:

```
on : X -> {X -> Y} -> Y
on x f = f x

shortAnd : Bool -> {Bool} -> Bool
shortAnd x c = on x { tt -> c! | ff -> ff }
```

Effekte

```
interface Send X = send : X -> Unit

interface Receive X = receive : X

chatbot : {[Send String, Receive String]Unit}
chatbot! =
  send "Hallo! Wie heißt du?";
  send ("'" ++ receive! ++ "' ist ein schöner Name!");
  chatbot!

data Unit = unit

(;) : X -> Y -> Y
(;) x y = y
```

Effekte

```
interface Send X = send : X -> Unit

interface Receive X = receive : X

chatbot : {[Send String, Receive String]Unit}
chatbot! =
  send "Hallo! Wie heißt du?";
  send ("'" ++ receive! ++ "' ist ein schöner Name!");
  chatbot!
```

- Regel: Jede Berechnung, die Interfaces verwendet, muss diese in eckigen Klammern vor dem Rückgabetyp angeben:
  `{t1 -> ... -> tn -> [I1, ..., Im]}`

Effekte

```
data Void =

interface Abort = aborting : Void

abort : [Abort]X
abort! = on aborting! {}
```

Effekte

```
interface State S = get : S
                  | put : S -> Unit

next : [State Int]Int
next! = fst get! (put (get! + 1))

fst : X -> Y -> X
fst x y = x
```

Effekte

```
sends : List X -> [Send X]Unit
sends xs = map send xs; unit

indexer : List X -> [State Int]List (X, Int)
indexer xs = map {x -> (x, next!)} xs

-- map : {X ->   Y} -> List X ->   List Y
-- map : {X -> []Y} -> List X -> []List Y
```

Effekte

```
sends : List X -> [Send X]Unit
sends xs = map send xs; unit

indexer : List X -> [State Int]List (X, Int)
indexer xs = map {x -> (x, next!)} xs

-- map : {X ->   Y} -> List X ->   List Y
-- map : {X -> []Y} -> List X -> []List Y
```

- Regel: Jede Berechnung ist implizit polymorph in ihren Effekten. Man kann `map` mit einer Berechnung, die das `Send`-Interface verwendet, als Argument aufrufen. Das Ergebnis ist eine Berechnung, die ebenfalls das `Send`-Interface verwendet.

Effekt-Handler

```
state : S -> <State S>X -> X
state _ x                = x
state s <get -> k>       = state s (k s)
  -- k : X -> <State S>X
state _ <put s -> k> = state s (k unit)
  -- k : Unit -> <State S>X
```

Effekt-Handler

```
state : S -> <State S>X -> X
state _ x                = x
state s <get -> k>       = state s (k s)
  -- k : X -> <State S>X
state _ <put s -> k> = state s (k unit)
  -- k : Unit -> <State S>X
```

- Regel: Jede Berechnung hat die Möglichkeit, die Effekte ihrer übergebenen Berechnungen zu behandeln. Im Typ werden die behandelten Interfaces in eckigen Klammern angegeben.

Effekt-Handler

```
state : S -> <State S>X -> X
state _ x                = x
state s <get -> k>       = state s (k s)
  -- k : X -> <State S>X
state _ <put s -> k> = state s (k unit)
  -- k : Unit -> <State S>X
```

- Regel: Jede Berechnung hat die Möglichkeit, die Effekte ihrer übergebenen Berechnungen zu behandeln. Im Typ werden die behandelten Interfaces in eckigen Klammern angegeben.

```
index : List X -> List (X, Int)
index xs = state 0 (indexer xs)
  -- indexer : List X -> [State Int]List (X, Int)
  -- e.g. index "abc" == [('a', 0), ('b', 1), ('c', 2)]
```

Effekt-Handler

```
catch : <Abort>X -> {X} -> X
catch x                 _ = x
catch <aborting -> _> f = f!
```

Effekt-Handler

```
feed : List R -> <Receive R>X -> [Abort]X
feed _           x                  = x
feed (cons x xs) <receive -> k>     = feed xs (k x)
feed nil         <receive -> _>     = abort!
```

Effekt-Handler

```
pipe : <Send X>Unit -> <Receive X>Y -> [Abort]Y
pipe <send x -> s> <receive -> r> = pipe (s unit) (r x)
pipe _             y              = y
pipe <_>           y              = y
pipe unit          <_>            = abort!
```

```
spacer : [Send String, Receive String]Unit
spacer! = send receive!; send " "; spacer!

catter : [Receive (List X)]List X
catter! =
  on receive!
    { nil -> nil
    | xs  -> xs ++ catter!
    }

pipe
  (sends ["do", "be", "do", ""])
  (pipe spacer! catter!)
-- evaluates to ["do", " ", "be", " ", "do", ""]
```

```
interface Choice = choice : Bool

data Toss = Heads | Tails

toss : [Choice]Toss
toss! = if choice! {Heads} {Tails}

drunkToss : [Choice, Abort]Toss
drunkToss! = if choice! toss abort

drunkTosses : Int -> [Choice, Abort]List Toss
drunkTosses 0 = nil
drunkTosses n = cons drunkToss! (drunkTosses (n-1))
```

```
allChoices : <Choice>X -> List X
allChoices x                = cons x nil
allChoices <choice -> k> =
  allChoices (k true) ++ allChoices (k false)

data Maybe X = nothing | just X

maybeAbort : <Abort>X -> Maybe X
maybeAbort x                  = just x
maybeAbort <aborting -> k> = nothing

t5 : Maybe (List (List Toss))
t5! = maybeAbort (allChoices (drunkTosses 2))


t6 : List (Maybe (List Toss))
t6! = allChoices (maybeAbort (drunkTosses 2))
```

```
allChoices : <Choice>X -> List X
allChoices x                = cons x nil
allChoices <choice -> k> =
  allChoices (k true) ++ allChoices (k false)

data Maybe X = nothing | just X

maybeAbort : <Abort>X -> Maybe X
maybeAbort x                  = just x
maybeAbort <aborting -> k> = nothing

t5 : Maybe (List (List Toss))
t5! = maybeAbort (allChoices (drunkTosses 2))

-- nothing

t6 : List (Maybe (List Toss))
t6! = allChoices (maybeAbort (drunkTosses 2))

-- [just [Heads, Heads], just [Heads, Tails], nothing,
--  just [Tails, Heads], just [Tails, Tails], nothing, nothing
```