Funktionale Programmierung

Putting the FUN in programming

Christian Kellermann

ckeen@pestilenz.org
mastodon.social/@ckeen

Was ist denn jetzt ‘funktionale Programmierung’?

- Funktionen sind nur abhängig von Eingabeparametern
- Funktionen sind selbst ein Teil der Sprache
- Beinhaltet immer ein Typsystem:
  - Dynamisch (LISP, Scheme)
  - Statisch (Haskell, F#, ML, Rust, ...)

Funktionen sind nur abhänging von den Eingangsparametern

```
(define (square n) (* n n))
```

```
square :: Int -> Int
```

```
square n = n * n
```

Funktionen sind nur abhängig von den Eingangsparametern

- Keine Abhängigkeiten heisst kein global state
- Leicht zu testen, weil kaum mockups nötig
- Haskell hat QuickCheck, das kann Ein-/Ausgaben selbst generieren
- Scheme / Lisp tut sich da schwerer.
-

Funktionen kann man verketten

```
(define (double n) (* 2 n))
```

```
double :: Int -> Int
double n = 2 * n
```

Funktionen kann man verketten

(double (square 2)) → 8

double (square 2) → 8

Funktionen kann man verketten

- Die Verkettung is so häufig, daß Haskell dafür nen Operator hat: '.'

```
doublensquare :: Int → Int
doublensquare = double . square
```

Funktionen sind selbst ein Datentyp

- Ich kann also Funktionen generieren:

```
(define (adder n)
    (lambda (x) (+ n x)))
(adder 2) → #<procedure>
((adder 2) 2) → 4
```

Funktionale patterns

- Auslagern der “Dinge, die getan werden”
- Beispiel: Maximum einer Liste von Zahlen berechnen

Funktionale patterns

```
int max = 0;
int *list[] = { 1, 2, 3, 4 };
int list_len = 4;
for (int i=0;i<list_len;i++){
    if (list[i] > max)
        max = list[i];
}
```

Funktionale patterns

```
(define l '(1 2 3 4))
(define (maximum input-list
   (let loop
      ((max 0)
       (l input-list))
     (cond ((null? l) max)
           ((< max (car l))
              (loop (car l) (cdr l)))
           (else
```

Funktionale patterns

- Scheme Version hat einige Vorteile gegenüber der C Version:
  - Dynamische Speicherverwaltung
  - Variable Länge der Liste
- Ist aber ganz schön lang!
- Und wenn wir jetzt das Minimum wollen?

Funktionale patterns

- Also die Aktion in eine Funktion packen!

```
(define l '(1 2 3 4))
(define (find what input-list)
  (let loop ((acc 0) (l input-list))
    (cond ((null? l) acc)
          (else (loop (what (car l)) (cdr l))))))
(define (max a b) (if (<= a b) b a))
(find max l) → 4
```

Funktionale patterns

- Das find-what ist also nur noch ein Iterator, der einen Wert aufsammelt. Das gibt es schon:

```
(define (maximum lst)
  (fold max 0 lst))
```

Funktionale patterns

- Die “Aktion” kann natürlich auch die Fehlerbehandlung sein

```
(define (foo some input error-handler)
  (when (invalid? some input)
    (error-handler some input))
  ...)
```

Funktionale patterns

- Warum also nicht in beiden Fällen?

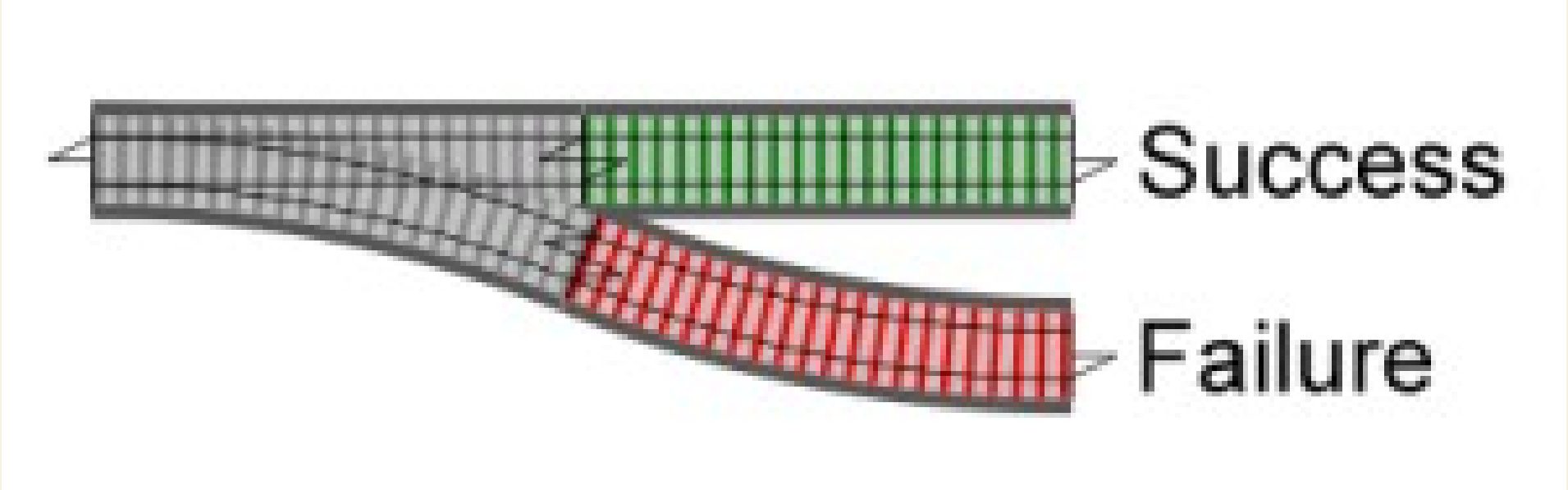
```
(define (parser input success failure)
  (cond ((is-valid? input) (success input))
        (else (failure input))))
```

Funktionale patterns

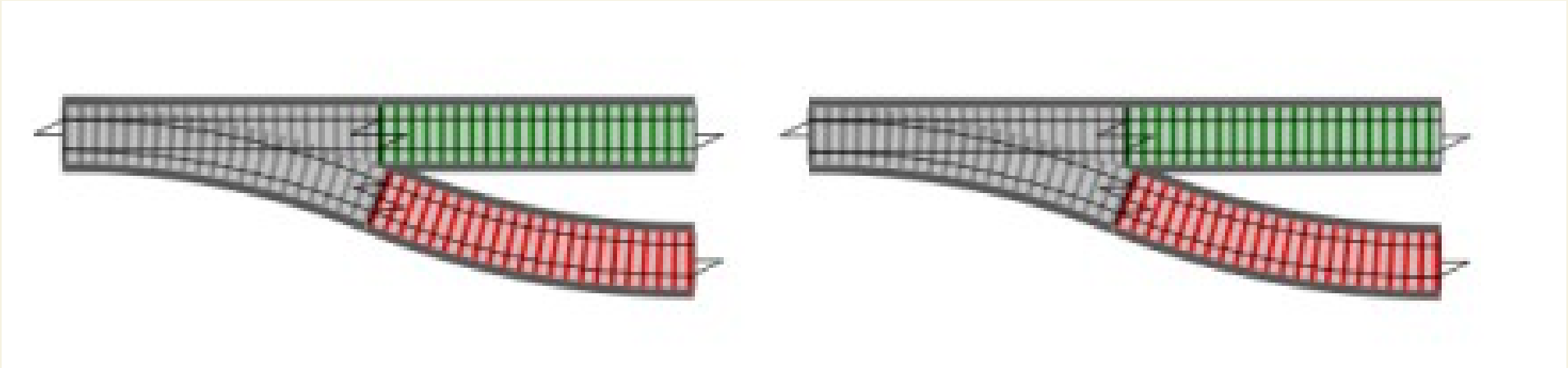
- Das explizite Übergeben der Kontrolle nennt man auch: continuation passing style (CPS)
- Die “continuation” ist der Teil des Programms, der den Wert einer Funktion erwartet
- Scheme kann diese continuation als Funktion “einfangen”

```
(call-with-current-continuation
  (lambda (continuation) …))
```

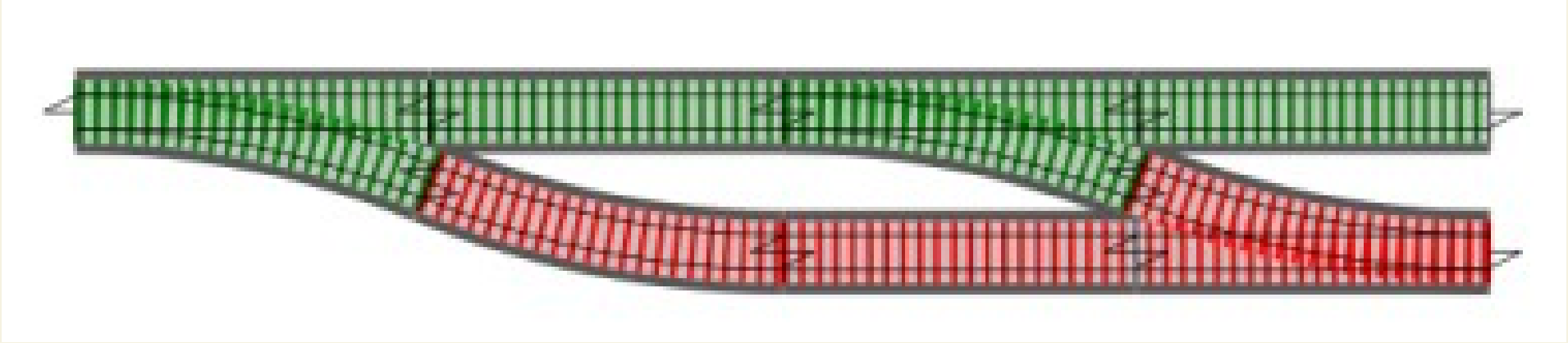
Continuation Passing Style



CPS – 2 Funktionen



Verbundene Funktionen



Funktionale patterns

- Beispiel: Finde das erste Element einer Liste, für die pred? true zurückgibt.

```
(define (find-first pred? lst)
  (call-with-current-continuation
    (lambda (return)
      (fold
        (lambda (r element)
          (when (pred? element) (return element)))
```

Funktionale patterns

- Dinge, die man damit tun kann:
  - Kooperatives Scheduling
  - Exceptions, restarts
  - Stateful webservices
  - Kontrollfluss (v.A. im Fehlerfall!) → Haskells Monaden

Now to something completely different!

Typsysteme

Scheme hat ein dynamisches Typsystem:

- Type checks passieren zur Laufzeit
- Der compiler kann einem nur bedingt helfen, z.B. durch flow analysis
- Man kann sich zwar behelfen, hat aber seinen Preis, z.B. durch design-by-contract

Typsysteme

- Es wäre natürlich cool, wenn der compiler vorab den Typ kennt → statische Typsysteme
- Einer der prominenten Vertreter davon ist Haskell
- Die folgenden Dinge lassen sich dadurch so gut wie gar nicht in Scheme abbilden

Haskell Datentypen

Warnung!

Die Mathematiker haben sich das zuerst ausgedacht, deswegen sind die Namen für die Dinge unverständlich^Wunintuitiv^W eben aus der Mathematik.

Haskell Datentypen

- Haskell hat mehrere Datentypen
  - Einfache: Int, Float, Bool, Char, [Char], String,…
  - Strukturierte (records in Scheme)
  - Tagged datatypes
  - Algebraic Datatypes

Tagged Datatypes

```
data Shape = Circle Float Float Float |
                    Rectangle Float Float Float
   Float
surface :: Shape → Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) =  (abs $ x2 -
   x1) * (abs $ y2 – y1)
```

Tagged Datatypes

- Jeder ‘tag’ ist der Konstruktor:

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
```

Algebraische Datentypen

```
data Tree = Empty | Leaf Int | Node Tree Tree
  deriving (Eq, Show)
```

- Oder auch:

```
data Maybe a = Nothing | Just a
  deriving (Eq, Show)
```

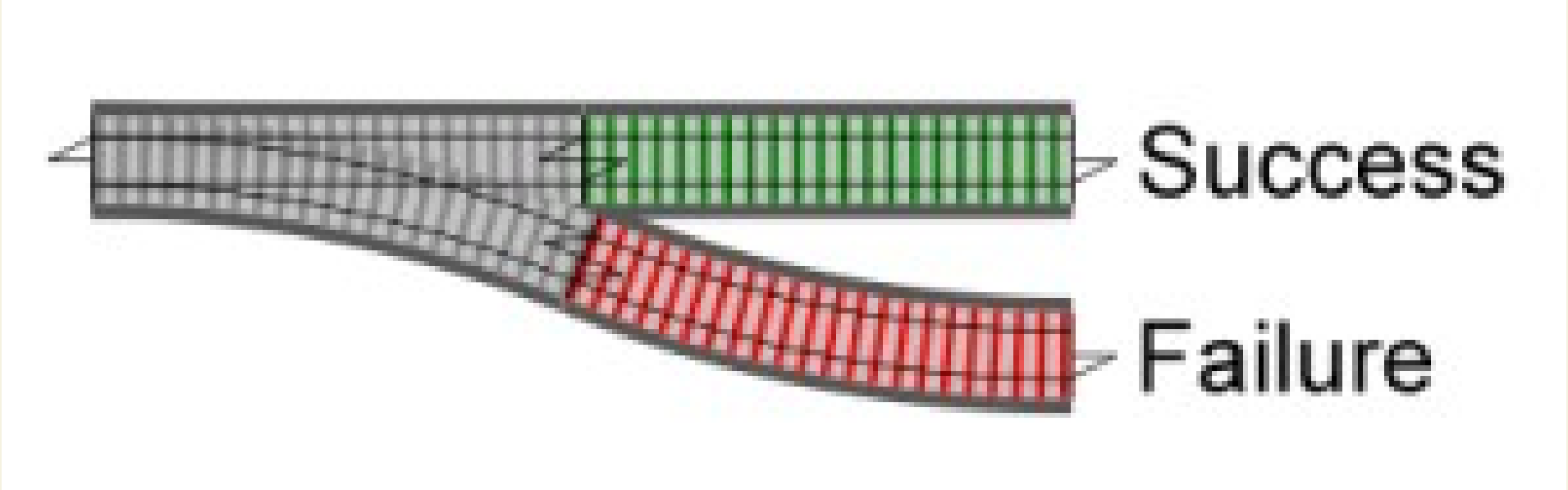
Maybe?

```
divide :: Int → Int → Float
divide a b = a / b
```

Und die 0?

```
divide a 0 = Nothing
divide a b = Just (a/b)
```

Maybe (aka implizites CPS)



Fehlerbehandlung mit GADTs (Maybe)

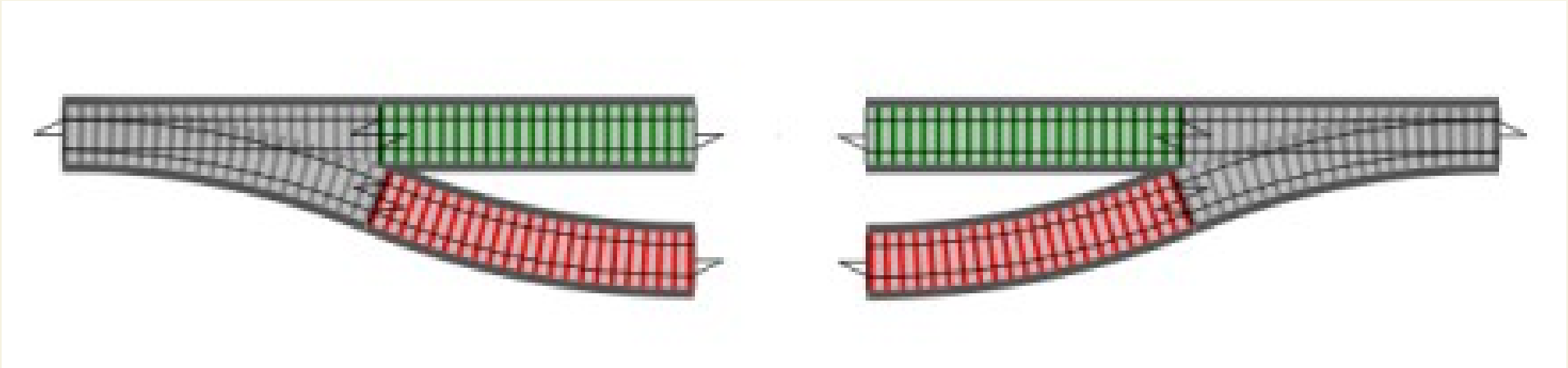
```
validInput :: Maybe a → Bool
validInput (Just a) = True
validInput Nothing = False

validInput $ divide 3 0 → False
```

CPS – 2 Funktionen

Parse

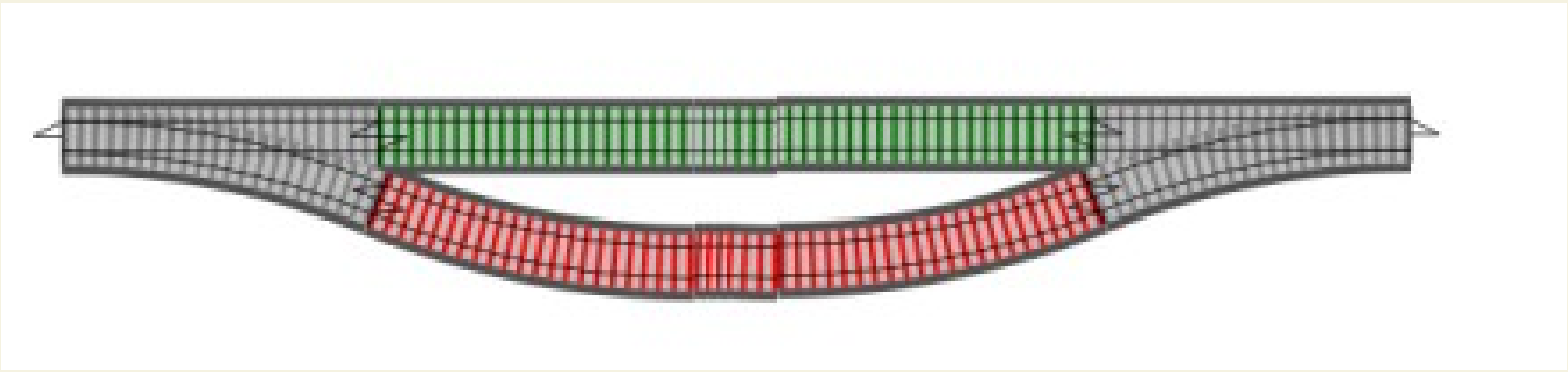
validInput



Verbundene 2 Funktionen

Parse

validInput



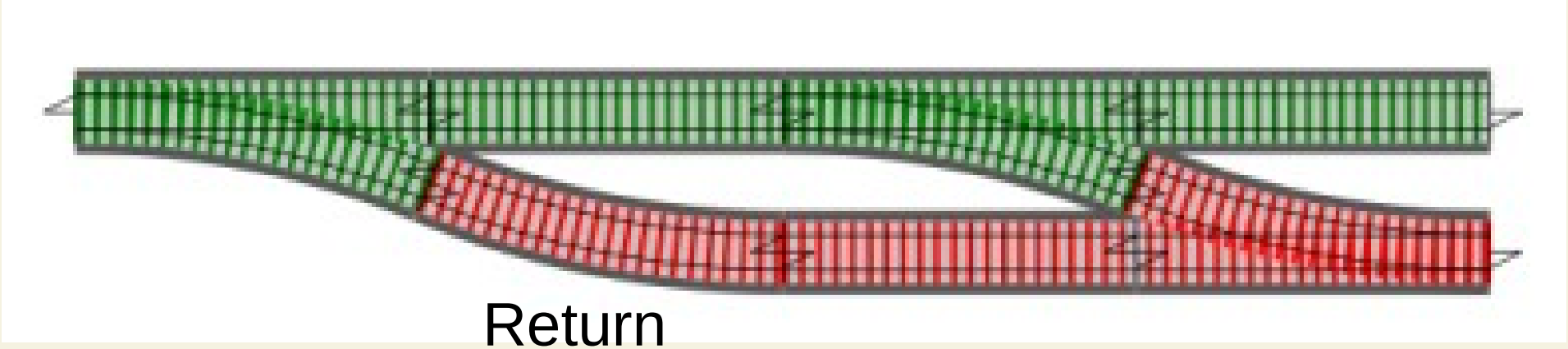
“Continuations” in Haskell

- Haskell Typen können mit einem Interface von Funktionen verknüpft werden: Type classes
- Wenn wir dann diese Funktionen voraussetzen:
  - Fehlerfall: return
  - Erfolgsfall: bind aka. >>=
  - Chain aka >>
  - Fehlermeldung: fail
- Damit haben wir einen Monaden geschaffen

Haskell Monade

Bind baut 2 Funktionen zusammen

Success



Return

Seiteneffekte

- Haskell ist eine sog. rein funktionale Sprache
- I/O ist als Berechnung innerhalb eines Monaden Typs definiert: IO()
- Haskell hello world:

```
putStrLn “Hello World!”

ghci> :t putStrLn

putStrLn :: string → IO()
```

Seiteneffekte

- Der compiler weiss also welche Teile des Programms I/O erzeugen
- Lazy evaluation möglich

Monaden

- Irgendwann landed fast alles im IO Monad: Monaden / Funktionen zusammenstecken ist essenzielles Handwerk
- Nicht abschrecken lassen von den Namen oder den Tutorials (seien es Bahnschienen, Buritos oder Space suites)
- Im Zweifelsfall einfach auf die Definition schaun

Scheme vs Haskell

- Pros Haskell:
  - Der compiler kann schnelleren code erzeugen
  - Der compiler kann einem helfen Fehler zu finden, wenn man Funktionen falsch zusammensteckt
- Cons Haskell:
  - Laufzeitverhalten schwer optimierbar (für Anfänger)
  - Bottom up artet schnell in rewrite Orgien aus
  - Monaden zu kombinieren ist am Anfang gar nicht so leicht

Scheme vs. Haskell

- Pros Scheme:
  - Schnelles Prototyping (“explorative Programming”)
  - Bottom up is leicht
  - Seiteneffekte kann man einbaun, wo man sie braucht
- Cons Scheme:
  - Der compiler kann einem oft nicht helfen
  - Laufzeit checks generieren exceptions zur Laufzeit

Haskell Anwendungen zum Lesen

- Xmonad, ein tiling window manager
- Darcs, dezentrales Versionskontrollsystem
- Yi, Ein Editor in Haskell

Scheme Anwendungen zum Lesen

- nsfwm, ein cwm like window manager (wip)
- vandusen, ein IRC bot
- Pastiche, ein paste bin in CHICKEN Scheme (Achtung: von mir selber)
- Ein wiki clone (auch von mir):

  http://pestilenz.org/~ckeen/wiki.scm

Referenzen

- Learn you a haskell: http://learnyouahaskell.com
- Scheme hat meist nur Bücher, der Sprachstandard hat aber auch nur ~60 Seiten und ist lesbar. Z.B.: 'The Scheme programming language'
- Die Eisenbahnweichen sind geklaut von

  https://fsharpforfunandprofit.com/posts/recipe-part2/