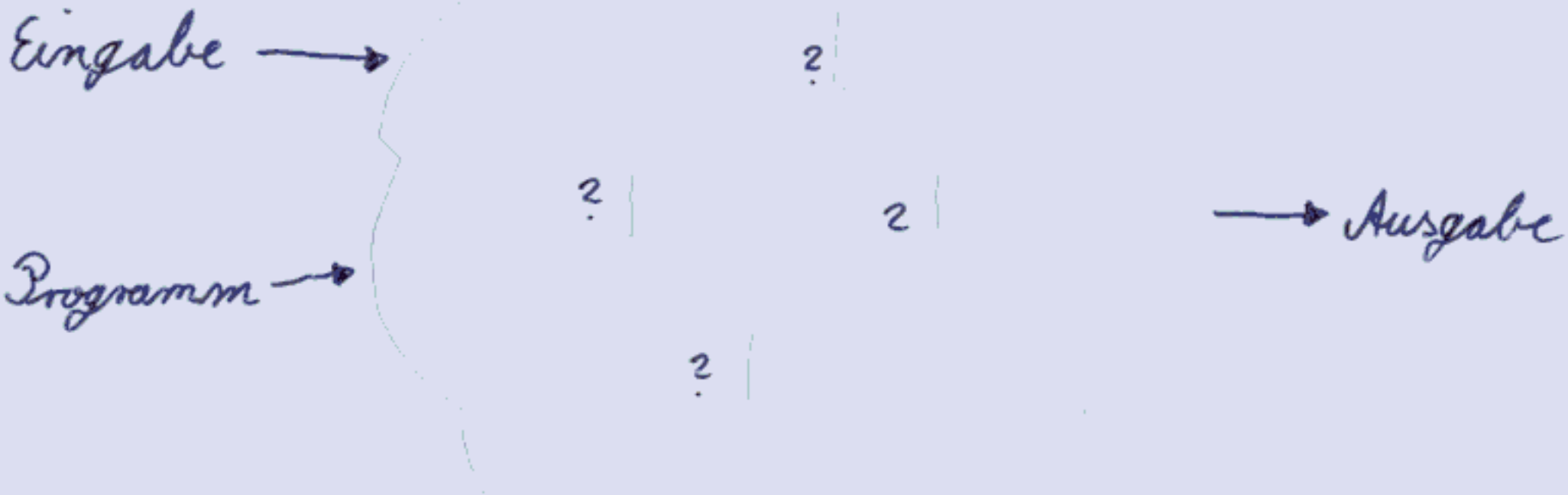
# Tracen und Debuggen von funktionalen Programmen mit verzögerter Auswertung

Olaf Chitil

The University of York

Colin Runciman, Malcolm Wallace, Thorsten Brehm

# Tracen




Berechnung

# Ziele

- Lokalisieren von Fehlern, die falsches Verhalten verursachen
  - falsche Ein- / Ausgabe - Aktionen
  - Abbruch mit Fehlermeldung
  - Nicht- Terminierung
- Verstehen, wie ein Programm funktioniert
- Enge Verbindung zum Quellcode
- Sicht des Programmierers, nicht der Implementierung

# Verzögerte Auswertung

```
factorial :: Int -> Int

factorial n = prod [1..n]
```

```
primes :: [Int]

primes = filter isPrime [2..]
```

```
all :: (a -> Bool) -> [a] -> Bool

all p xs = and (map p xs)
```

```
isPrime :: Int -> Bool

isPrime n =
  all ((/=0) . (n `mod`))
    (takeWhile ((<=n).(^2)) primes)
```

# Verzögerte Auswertung

```
all :: (a -> True) -> [a] -> Bool
all p xs  = and (map p xs)

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

```
   all even [4, 3, 1/0]
⇒ and (map even [4, 3, 1/0])
⇒ and (even 4 : map even [3, 1/0])
⇒ even 4 && and (map even [3, 1/0])
⇒ True && and (map even [3, 1/0])
⇒ and (map even [3, 1/0])
⇒ and (even 3 : map even [1/0])
⇒ even 3 && and (map even [1/0])
⇒ False && and (map even [1/0])
⇒ False
```

# Debugger für imperative Sprachen

zeigen zu einem Zeitpunkt der Berechnung
Teil des aktuellen Zustands

ungeeignet für funktionale Sprachen mit verzögerter Auswertung

- Auswertungsreihenfolge verwirrend
- unausgewertete Teilausdrücke groß ⇒ schwer lesbar
- Stack entspricht nicht der statischen Aufrufreihenfolge
- entspricht nicht der Sicht des Programmierers

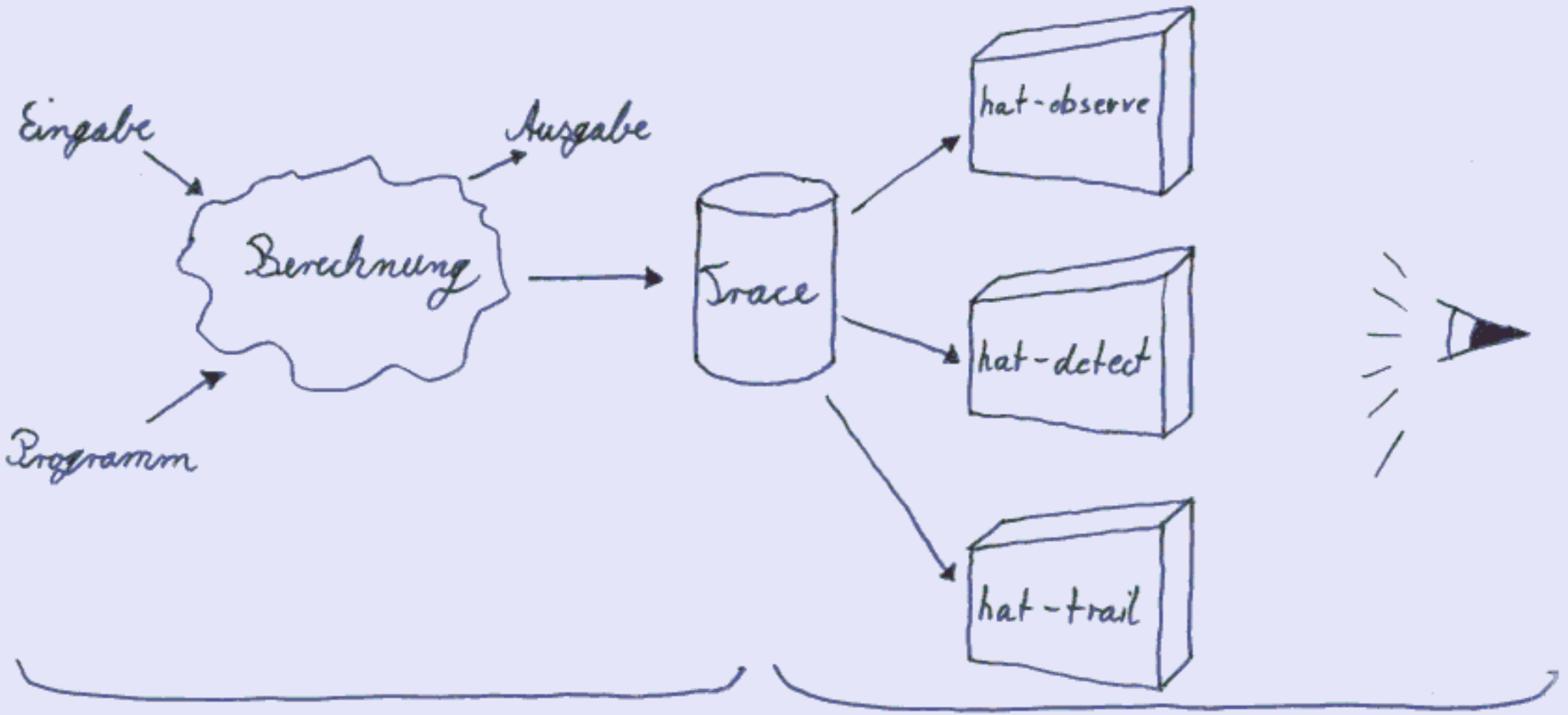
# Die print-Methode: traceShow

```
insert :: Int -> [Int] -> [Int]

insert x []     = [x]
insert x (y:ys) = if x > y then x : (traceShow ">" (insert x ys))
                           else x : y : ys

main = take 5 (insert 4 [1..])
```

Ausgabe: `>[4>[4>[4,4,5,6,7,8,9,10,11,...`

- Ausgabe verwirrt
- es wird zuviel ausgewertet ⇒ Beobachtung ändert Verhalten

# 2-Phasen-Methode




befreit vom Zeitpfeil der Berechnung

# Beobachten von Funktionen : hat-observe

```
insert :: Int -> [Int] -> [Int]

insert x []     = [x]
insert x (y:ys) =
    if x>y then  x : insert x ys
           else  x : y : ys

main = take 5 (insert 4 [1..])
```

Beobachte insert:

```
insert 4 (1:2:3:4:_) = 4:4:4:4:4:_
insert 4 (2:3:4:_)   = 4:4:4:4:_
insert 4 (3:4:_)     = 4:4:4:_
insert 4 (4:_)       = 4:4:_
```

vgl.:

- Andy Gill's Hood
- denotationelle Semantik

# Informationsreduktion in hat-observe

- keine redundanten Reduktionen

```
insert 5 (1:3:6:9:_) = 1:3:5:_
insert 5 (1:3:6:_)   = 1:3:5:6:_
```

- nur / nicht Aufrufe von einer Funktionsdefinition

```
insert 4 (1:2:3:4:_) = 1:2:3:4:_
```

- Muster für Reduktionen

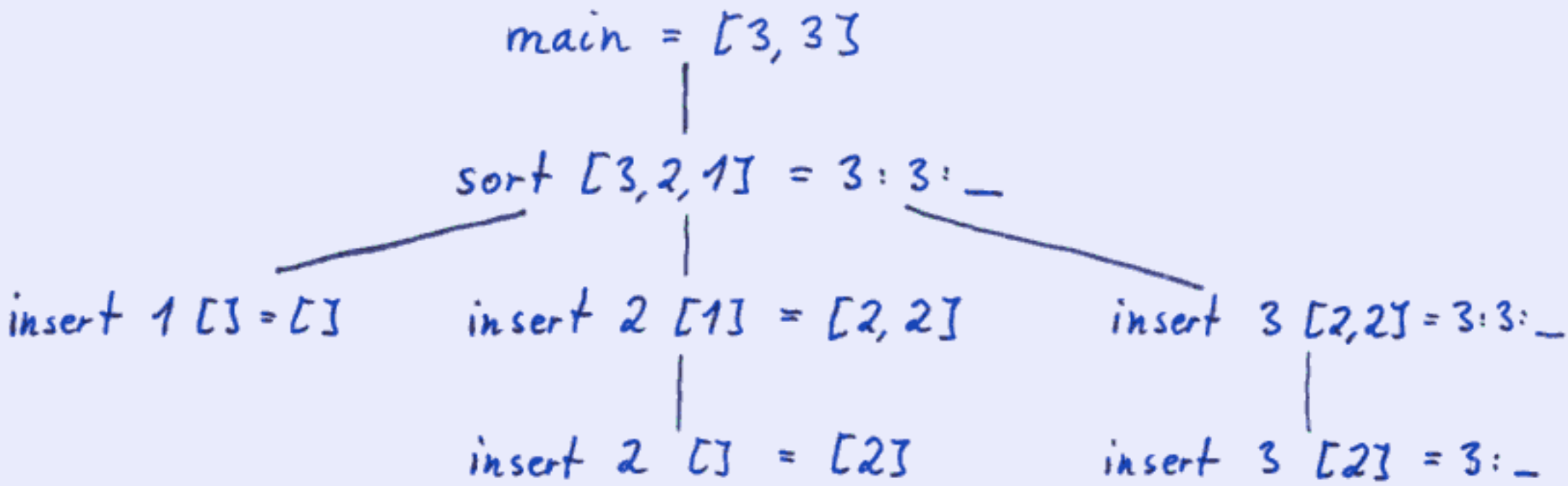
```
insert _ (3:_) = 3:_
```

# Algorithmisches Debuggen: hat-detect

```
insert x []     = [x]
insert x (y:ys) =
  if x > y then x : insert x ys
           else x : y : ys

sort xs = foldr insert [] xs

main = take 2 (sort [3,2,1])
```

```
main = [3,3]      ?  n

sort [3,2,1] = 3:3:_  ?  n

insert 1 [] = [1]  ?  y

insert 2 [1] = [2,2]  ?  n

insert 2 [] = [2]  ?  y

Error located:

  insert 2 [1] = [2,2]
```

# Der Berechnungsabhängigkeitsbaum




vgl.

- Shapiro ; Henrik Nilsson's Freja
- Großschrittsemantik

# Informationsreduktion in hat-detect

- vertraue Funktionen / Modulen

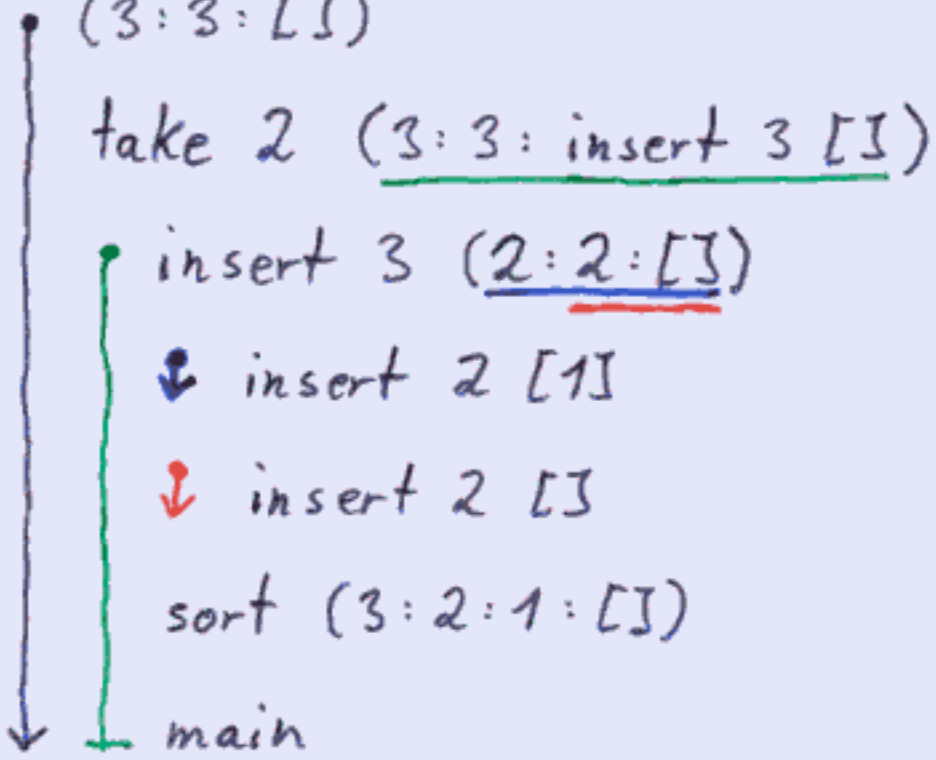
  take, foldr

- nie dieselbe / spezifischere Frage nochmal

- Antwort vielleicht richtig / falsch

# Erzeuger zurückverfolgen : hat-trail

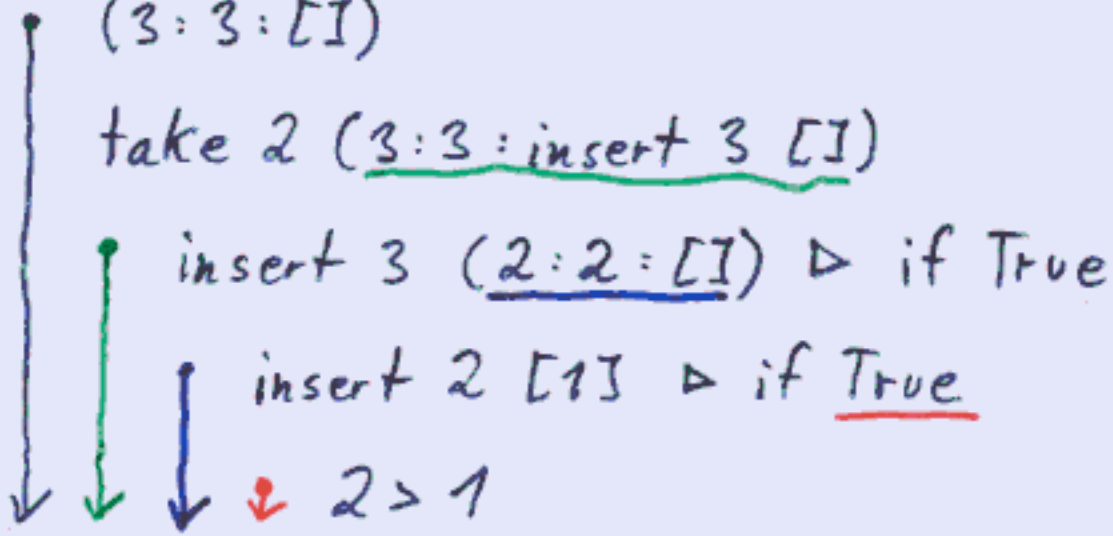
```
insert x []     = []
insert x (y:ys) =
   if x>y then x:insert x ys
          else x:y:ys

sort xs = foldr insert [] xs

main = take 2 (sort [3,2,1])
```




vgl. Graphersetzungssemantik

# Kontrollfluß : if, case, guard

```
insert x []     = []
insert x (y:ys) =
    if x>y then  x:insert x xs
           else  x:y:ys
```



# Wechsel zwischen den Sichten

hat-observe → hat-trail

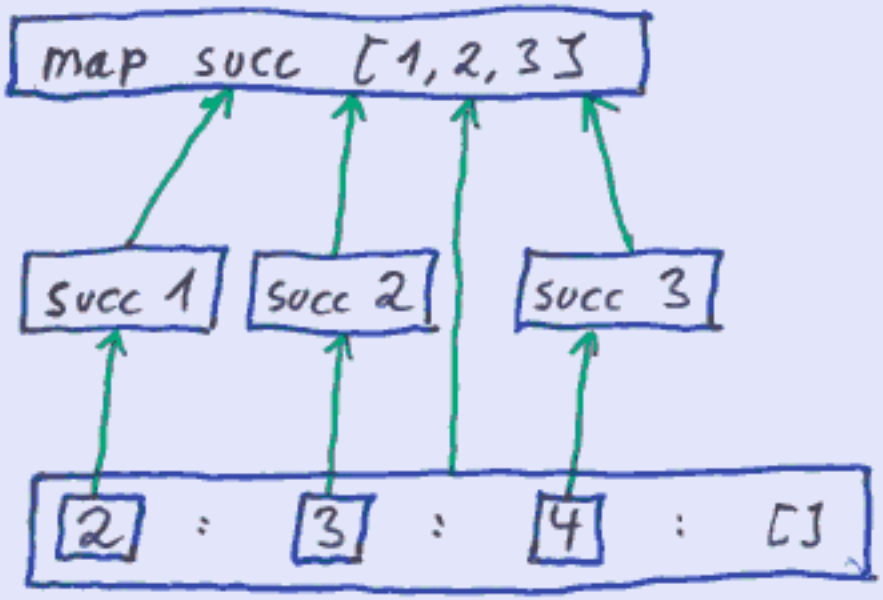
factorial (-1) = 1

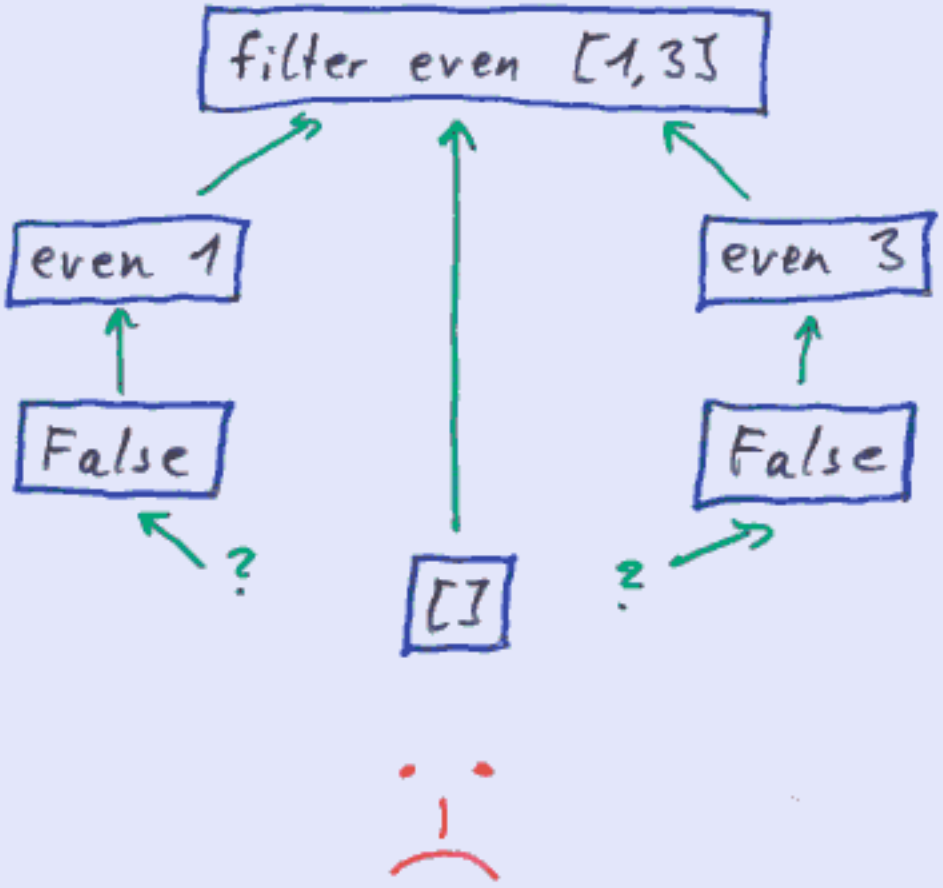
hat-detect → hat-observe

beobachte Funktion, bevor wir ihr trauen

# Vertrauen von Modulen

direkt bei Traceerzeugung spart Speicherplatz



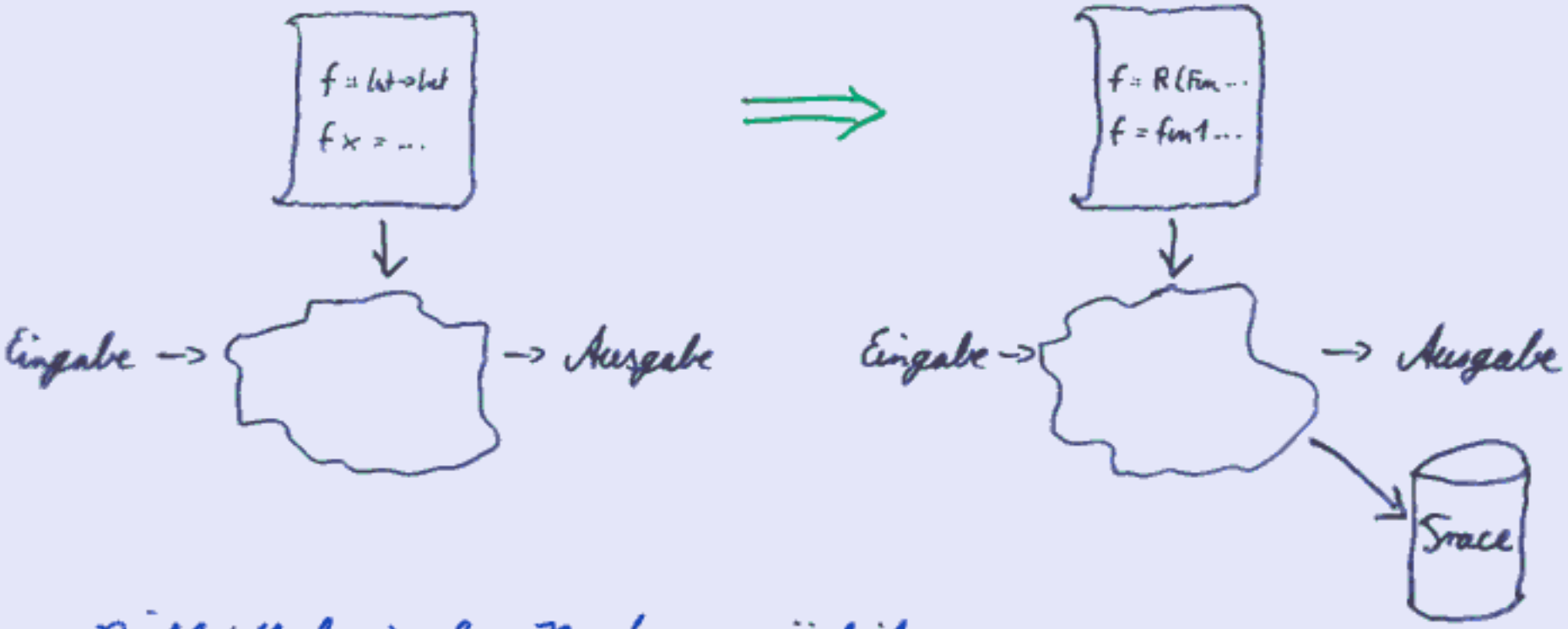
# Freie Variablen in lokal definierten Funktionen

```
mults x ys = map g ys
  where
  g y = x * y
```

```
g 1 = 2
g 2 = 4
g 1 = 3   ??
```

```
(g where x=2) 1 = 2
(g where x=2) 2 = 4
(g where x=3) 1 = 3
```

# Implementierung durch Programmtransformation



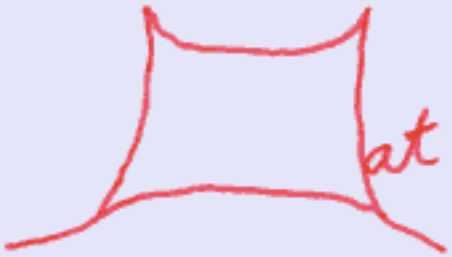

- Bibliothek à la Hood unmöglich
- Programmtransformation portabel
- Abstrakte Maschine à la Freja effizienter

# Zusammenfassung

2 Phasen
- befreien vom Zeitpfuhl der Berechnung
- modularisieren Tracerentwicklung

3 Sichten
- hat-observe : Beobachten von Funktionen
- hat-detect : Algorithmisches Debuggen
- hat-trail : Erzeuger zurückverfolgen

Implementierung durch Programmtransformation




http://www.cs.york.ac.uk/fp/hat