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

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

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 vermischt
- es wird zuviel ausgewertet ⇒ Beobachtung ändert Verhalten

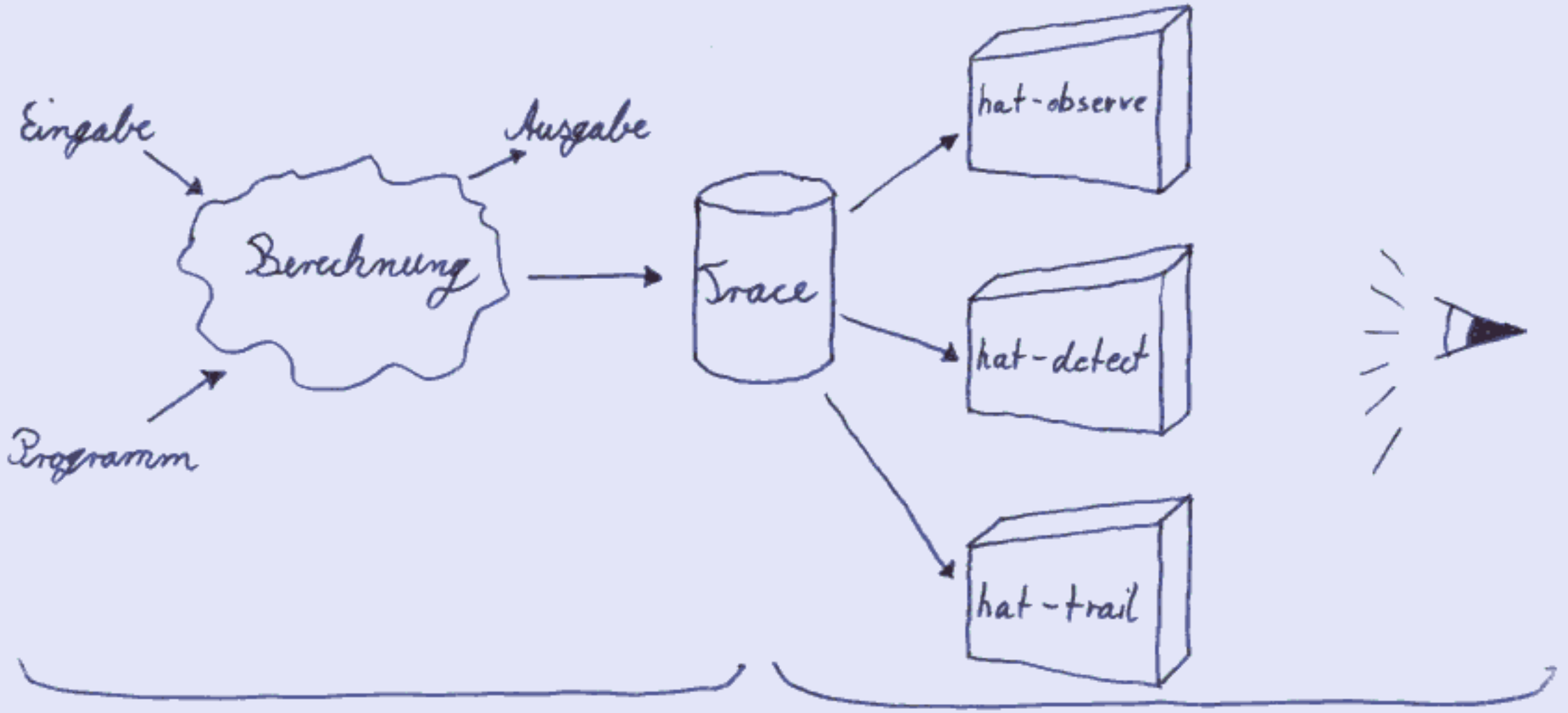
Debugger für imperative Sprachen

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

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:_
```

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])
```

Output:

```
[3,3]
```

Trail:

```
← take 2 (3:3:_) = [3,3]
← insert 3 [2,2] | if True
← insert 2 [1] | if True = [2,2]
← sort [2,1] = [2,2]
← sort [3,2,1] = [2,2]
← main
```

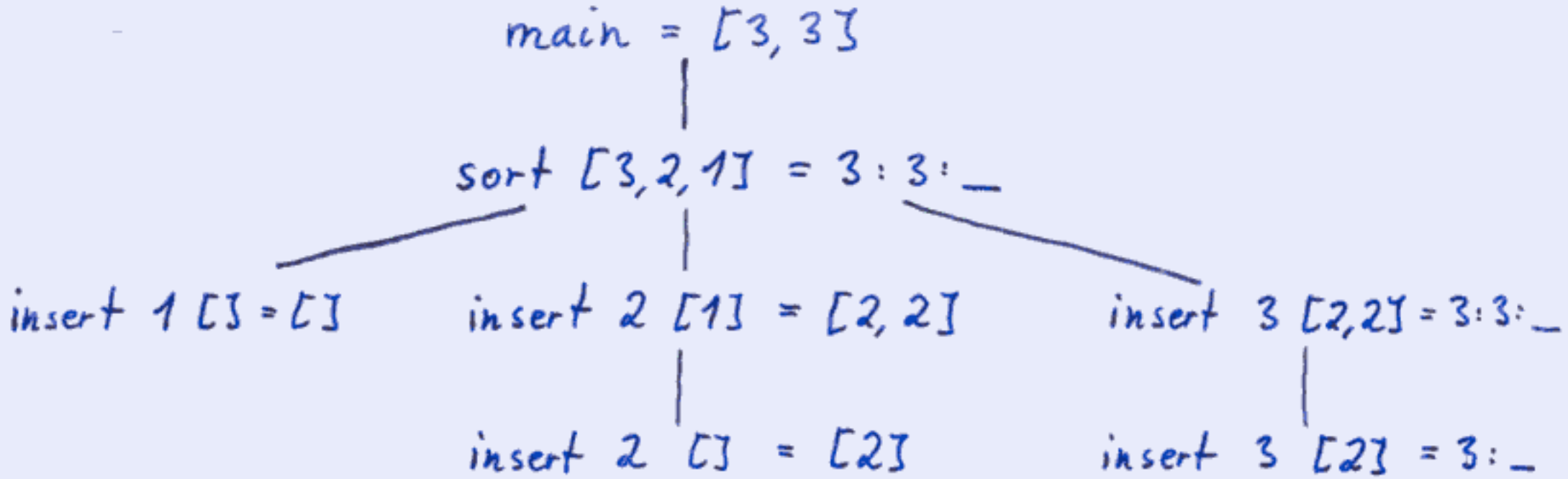
vgl. Graphersetzungssemantik

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]
```

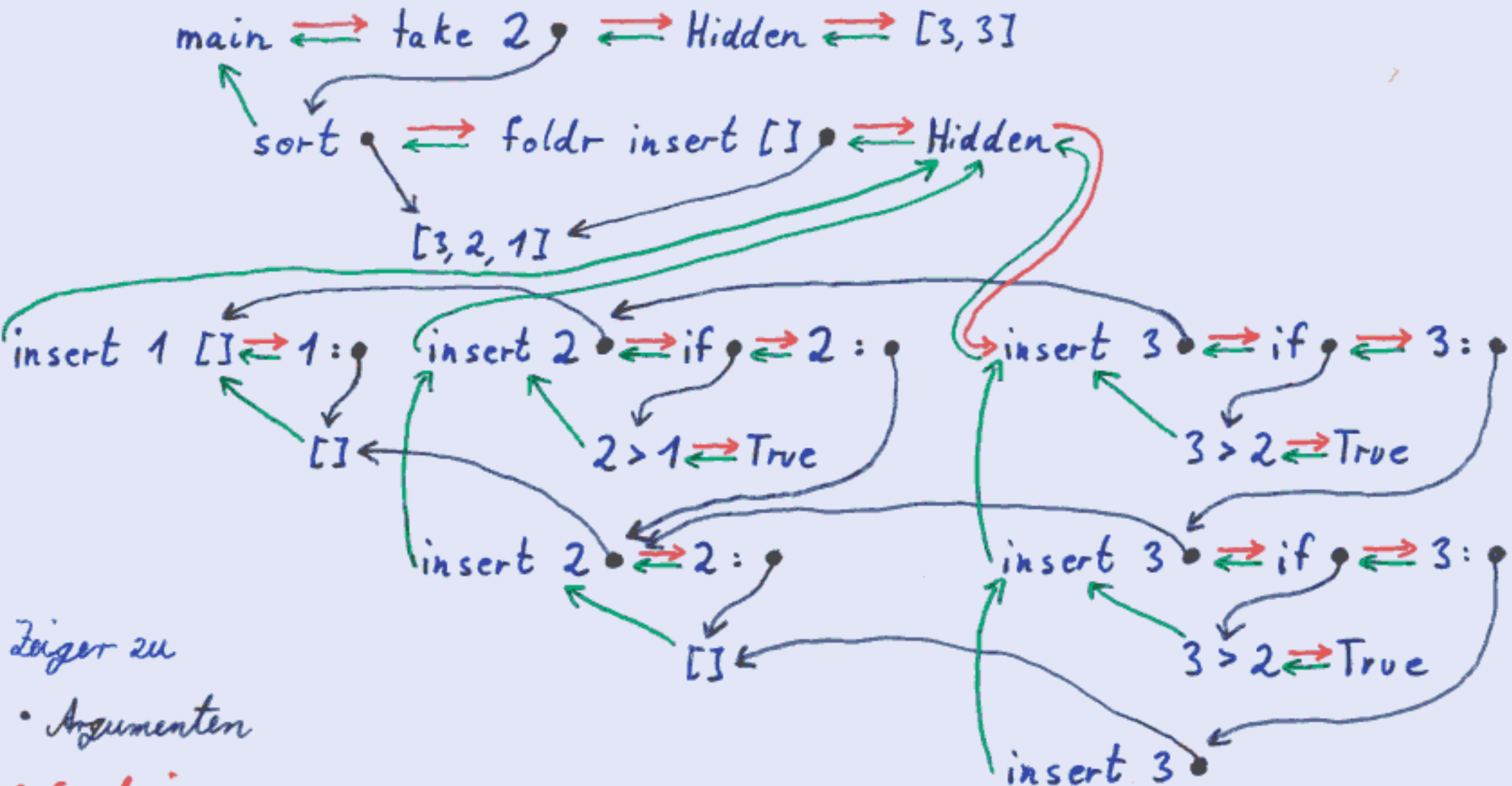
Der Berechnungsabhängigkeitsbaum



vgl.

- Shapiro ; Henrik Nilsson's Freya
- Großschrittsemantik

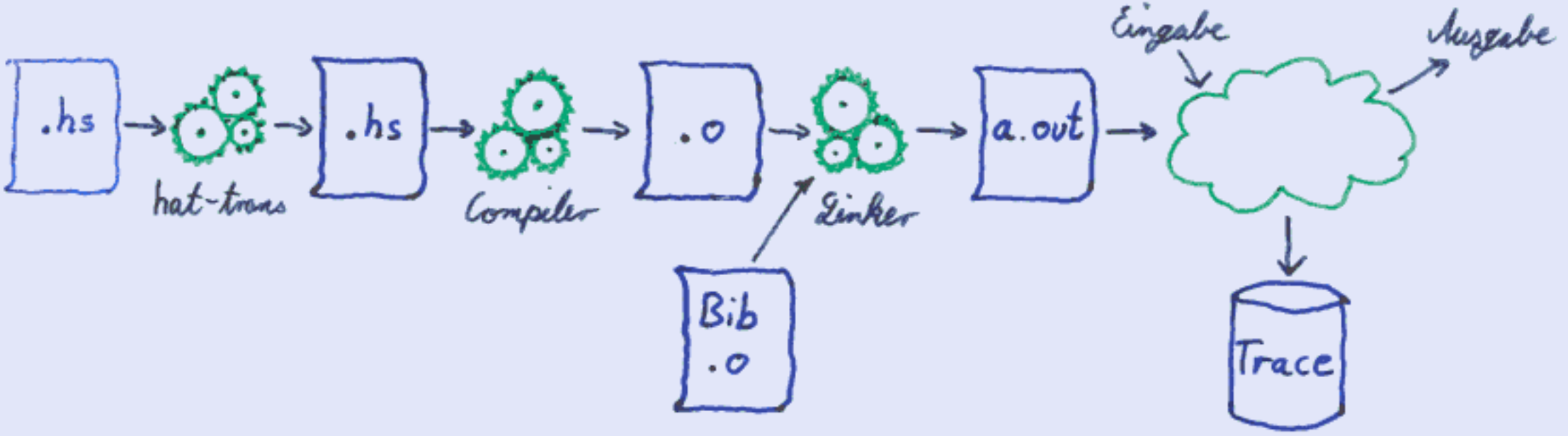
Der Trace



Zeiger zu

- Argumenten
- Ergebnissen
- Erzeugern

Trace-Erzeugung durch Programmtransformation



- Bibliothek à la Hood unmöglich
- Programmtransformation portabel: getrennte Definition und Implementierung
- Abstrakte Maschine à la Freja effizienter

Ausdrücke anreichern

```
data R a = R a RefExp
```

```
Bool ⇝ R Bool        Int ⇝ R Int
```

```
data Maybe a = Just a | Nothing
⇝ data Maybe a = Just (R a) | Nothing
```

```
⇝ data List a = Cons (R a) (R (List a)) | Nil
```

```
⇝ data Fun a b = Fun (RefExp → R a → R b)
```

Transformierte Typsignaturen

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

```
ginsert :: Ref SrcPos -> Ref Exp -> R (Fun Int ( Fun (List Int) (List Int)))

hinsert :: R Int -> R (List Int) -> Ref Exp -> R (List Int)
```

Syntaxe-gesteuerte Transformation

```
ginsert p j = fun2 ainsert p j hinsert

hinsert fx (R Nil _) j = con2 p11v16 j Cons aCons fx
                              (con0 p11v17 j Nil aNil)
hinsert fx (R (Cons fy fys) _) j =
  cif p13v3 j (ap2 p13v8 j ((!>) p13v8 j) fx fy)
    (\v -> con2 p13v18 v Cons aCons fx
             (ap2 p13v19 v (ginsert p13v19 v) fx fys))
    (\v -> ...)

ainsert = mkVariable tMain 11 1 3 2 "insert" False
```

Separate Bibliothek mit Kombinatoren

```
fun1 :: RefAtom -> RefSrcPos -> RefExp
     -> (R a -> RefExp -> R z) -> R (Fun a z)
fun2 :: RefAtom -> RefSrcPos -> RefExp
     -> (R a -> R b -> RefExp -> R z) -> R (Fun a (Fun b z))

ap1 :: RefSrcPos -> RefExp
    -> R (Fun a z) -> R a -> R z

ap2 :: RefSrcPos -> RefExp
    -> R (Fun a (Fun b z)) -> R a -> R b -> R z
```

Definitionen zweier Kombinatoren

```
fun1 var sr p f = R (Fun (\r a -> f a r))
                    (mkValueUse p sr var)

ap1 sr p (R f rf) a@(R _ ra) =
   let Fun f' = f
       r = mkApp1 p sr rf ra
       R y ry = f' a r
   in R y r
```

Redaktionsergebnis aktualisieren

```
ap1 sr p (R f rf) a@(R _ ra) =
  let Fun f' = f
      r = mkApp1 p sr rf ra
  in wrapReduction (f' r a) r

wrapReduction :: R a -> RefExp -> R a
wrapReduction x r = R (enterRedex r `seq`
                       case x of R y ry ->
                         updReduct r ry `seq` y)
                      r
```

Verbindung von traced und untraced Code

```
toChar :: RefExp -> R Char -> Char
fromChar :: RefExp -> Char -> R Char

toList :: (RefExp -> R a -> b) -> RefExp -> R (List a) -> [b]
fromList :: (RefExp -> a -> R b) -> RefExp -> [a] -> R (List b)

    toString :: RefExp -> R String -> Prelude.String
    toString = toList toChar

toFun :: (RefExp -> c -> R a) -> (RefExp -> R b -> d) -> RefExp
      -> R (Fun a b) -> (c -> d)
fromFun :: ...
```

Zugang zu primitiven Funktionen

. :

```
isUpper :: Char -> Bool
```

⇝

```
gisUpper p j = fun1 aisUpper p j hisUpper

hisUpper z1 k = fromBool k (isUpper (toChar k z1))
```

Kombinatoren für Tracing

vap1, vap2, vap3, ... : vfun1, vfun2, vfun3, ...

foldr insert [] [3,2,1] → Hidden → 3 : 3 : _

insert 1 [] → ... insert 2 [1] → ... insert 3 [2,2] → ...

In den Trace :
- Aufruf
- Endergebnis
- Berechnungen, denen wir nicht vertrauen

Auch trusted Code muß transformiert werden

1. `(++) :: [a] -> [a] -> [a]`

   ⇝ `(!++) p j = fun2 (<++) p j (>++)`

   ```
   (>++) z1 z2 k = fromList fromId k
                     (toList told k z1 ++ toList told k z2)
   ```

   `[1,2,3] ++ ([4,5,6] ++ [7,8,9])`

2. `elem :: Eq a => a -> [a] -> Bool`

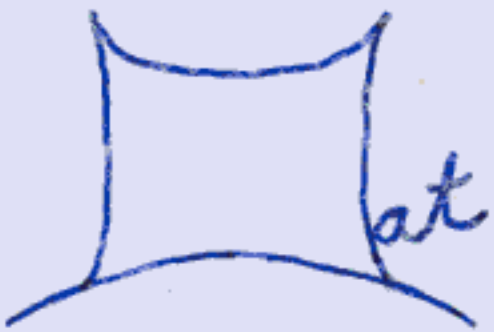
   ⇝ `gelem :: Eq a => Ref SrcPos -> RefExp -> R (Fun a (Fun (List a) Bool))`

Zusammenfassung

2 Phasen befreien vom Zeitpfeil der Berechnung

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

Programmtransformation:
- separiert Tracen
- syntaxe-gerichtet
- Bibliothek mit Kombinatoren



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