

Caught in a Vicious Circle

Haskell als reine Spezifikationsprache

Baltasar Trancón y Widemann

Universität Bayreuth

HaL 4

12. Juni 2009

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Caught in a Vicious Circle

Motto

*You're caught in a vicious circle
Surrounded by your so-called friends
You're caught in a vicious circle
And it looks like it will never end*

*You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
Surrounded by all of your friends*

Lou Reed, *Rock and Roll Heart*, 1976.

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Kontext

Das Projekt

- Mittelgroßes Haskell-Programm (1.3 kLoC)
- Verwendet viele der üblichen Sprachfeatures
 - ▶ Module (18)
 - ▶ Typ-, Konstruktorklassen, Generizität
 - ▶ Funktoren, Monaden, Monaden-Transformer
 - ▶ Eingebettete domänenspezifische Sprachen
- Intensive Debugging- und Testphase
- Ausgeführt: Niemals
 - ▶ Weder als Hauptprogramm noch als Bibliothek
 - ▶ Zweck?

Kontext

Das Projekt

- Mittelgroßes Haskell-Programm (1.3 kLoC)
- Verwendet viele der üblichen Sprachfeatures
 - ▶ Module (18)
 - ▶ Typ-, Konstruktorklassen, Generizität
 - ▶ Funktoren, Monaden, Monaden-Transformer
 - ▶ Eingebettete domänenspezifische Sprachen
- Intensive Debugging- und Testphase
- **Ausgeführt: Niemals**
 - ▶ Weder als Hauptprogramm noch als Bibliothek
 - ▶ Zweck?

Kontext

Abgedruckt und kommentiert in



B. Trancón y Widemann:

Strikte Verfahren Zyklischer Berechnung.

Dissertation, Technische Universität Berlin, 2007.

- Arbeit zu Semantik und Compilerbau
- Kein inhaltlicher Bezug zu Haskell

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Das Problem

Rekursion auf zyklischen Eingabedaten

waltz = 1 : 2 : 3 : *waltz*

...

Das Problem

Rekursion auf zyklischen Eingabedaten

waltz = 1 : 2 : 3 : *waltz*

Kein Problem für Haskell!

song = *map syll waltz*

where

syll 1 = "Tra"

syll 2 = "li"

syll 3 = "la"

take 11 (*concat song*) \rightsquigarrow "TralilaTral"

Das Problem

Rekursion auf zyklischen Eingabedaten

waltz = 1 : 2 : 3 : *waltz*

...oder doch?

<i>4 'elem' waltz</i>	$\rightsquigarrow \perp$	— einstellige Prädikate
<i>waltz == cycle [1..3]</i>	$\rightsquigarrow \perp$	— mehrstellige Prädikate
<i>filter (> 3) waltz</i>	$\rightsquigarrow \perp$	— nichtproduktive Funktionen
<i>sort waltz</i>	$\rightsquigarrow \perp$	— nichtlineare Funktionen

Das Problem

Rekursion auf zyklischen Eingabedaten

waltz = 1 : 2 : 3 : *waltz*

...oder doch?

4 'elem' <i>waltz</i>	$\rightsquigarrow \perp$	— einstellige Prädikate
<i>waltz</i> == <i>cycle</i> [1..3]	$\rightsquigarrow \perp$	— mehrstellige Prädikate
<i>filter</i> (> 3) <i>waltz</i>	$\rightsquigarrow \perp$	— nichtproduktive Funktionen
<i>sort waltz</i>	$\rightsquigarrow \perp$	— nichtlineare Funktionen

Das Problem

Erstaunliche Behauptung

Strikte Sprachen sind hier überlegen!

Leitmotiv

Those who cannot remember the past are condemned to repeat it.

J.A.N. Ruiz de Santayana, *The Life of Reason*, 1905.

Das Problem

Erstaunliche Behauptung

Strikte Sprachen sind hier überlegen!

Leitmotiv

Those who cannot remember the past are condemned to repeat it.

J.A.N. Ruiz de Santayana, *The Life of Reason*, 1905.

Übersicht

1 Einführung

- Kontext
- Das Problem
- Lösungsansatz

2 Spezifikation von Strategien

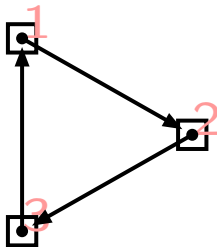
- Daten und Signaturen
- Speicherzustand
- Auswertung

3 Spezifikation einer Virtuellen Maschine

4 Zusammenfassung

Erster Lösungsansatz

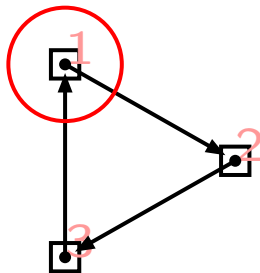
Lazy



$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

Lazy

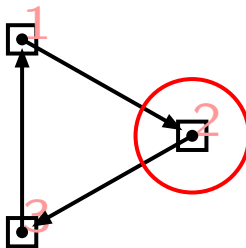


$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

Lazy

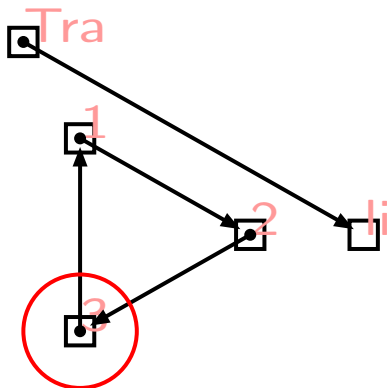
□ Tra



$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

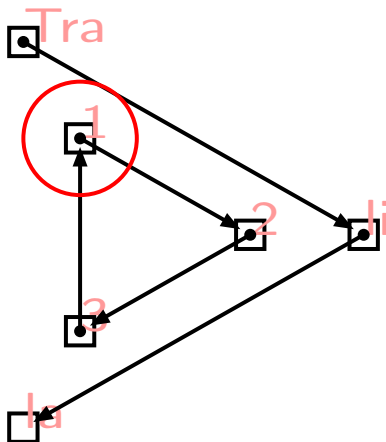
Lazy



$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

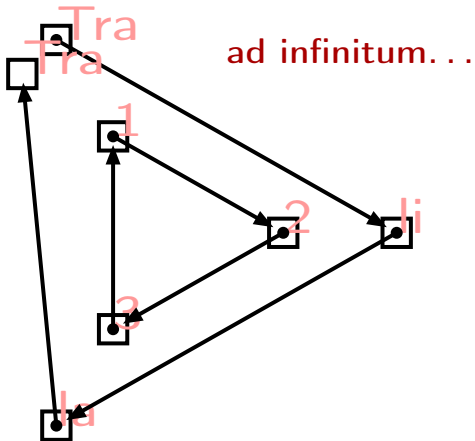
Lazy



$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

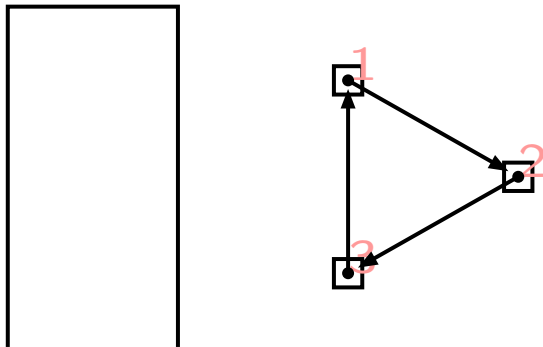
Lazy



$$\text{map } f \ (x : ys) = f \ x : \text{map } f \ ys$$

Erster Lösungsansatz

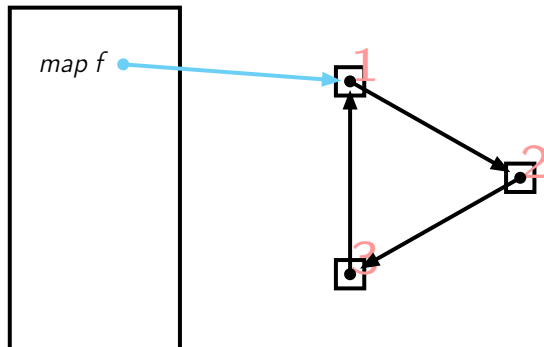
Strikt



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

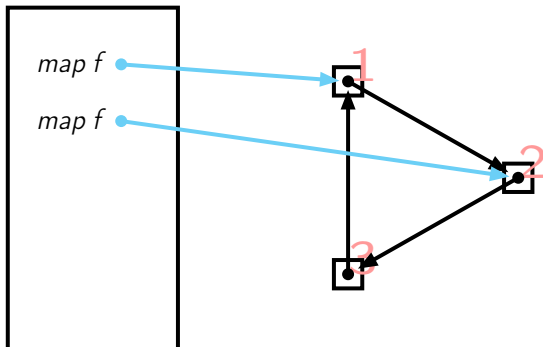
Strikt



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

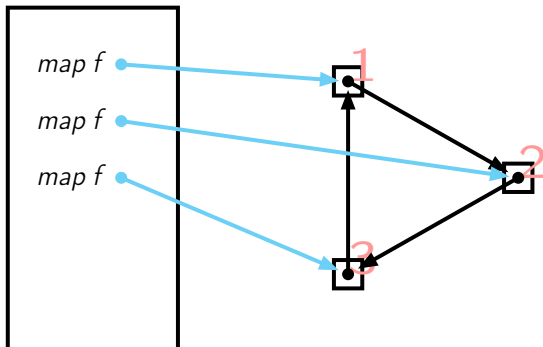
Strikt



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

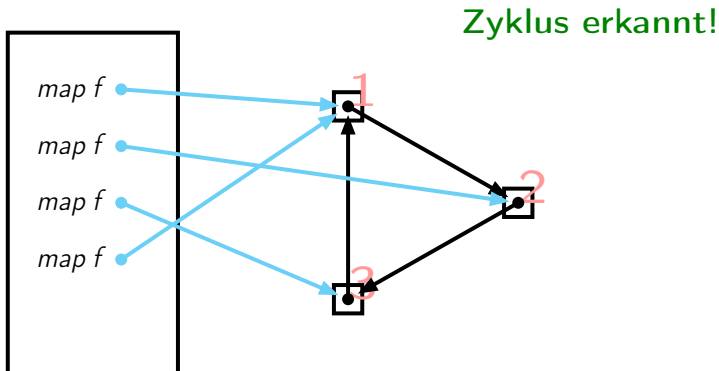
Strikt



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

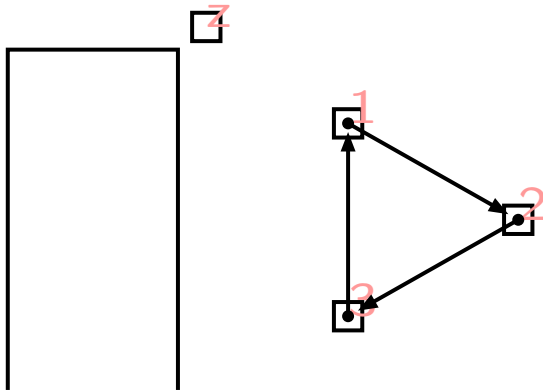
Strikt



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

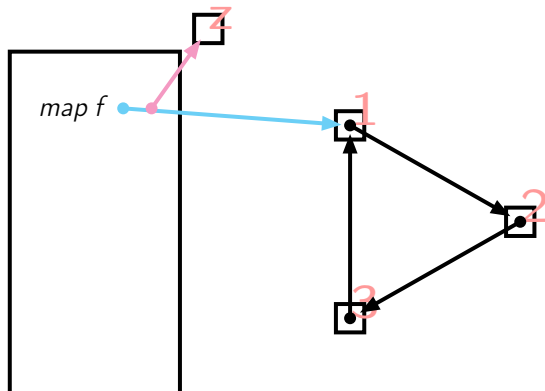
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

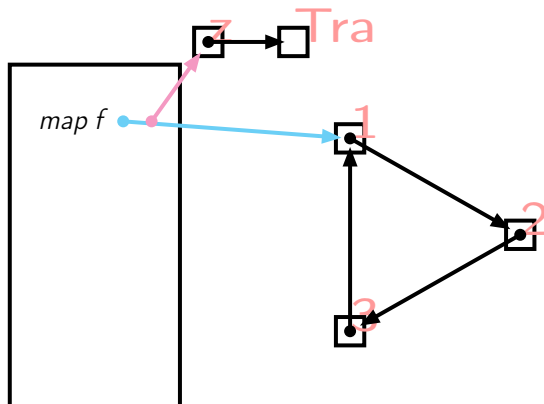
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

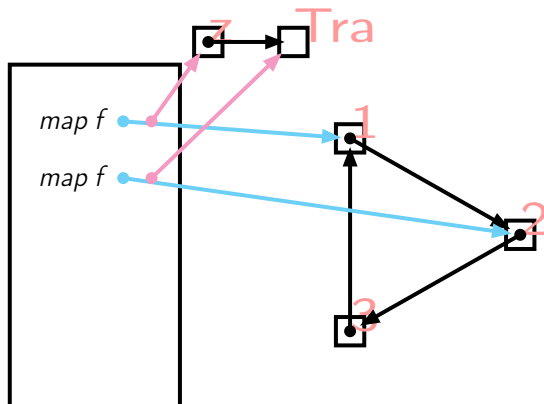
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

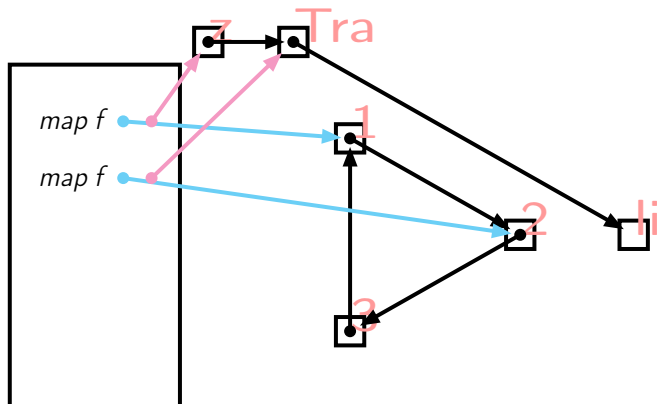
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

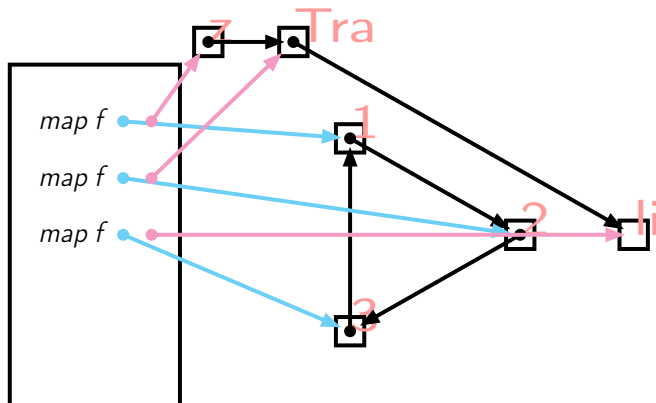
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

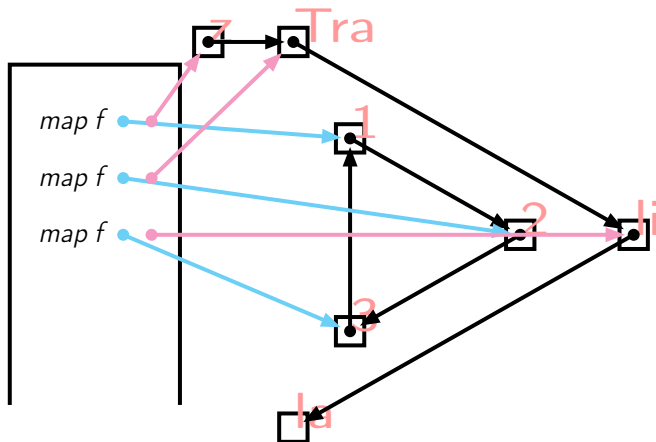
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

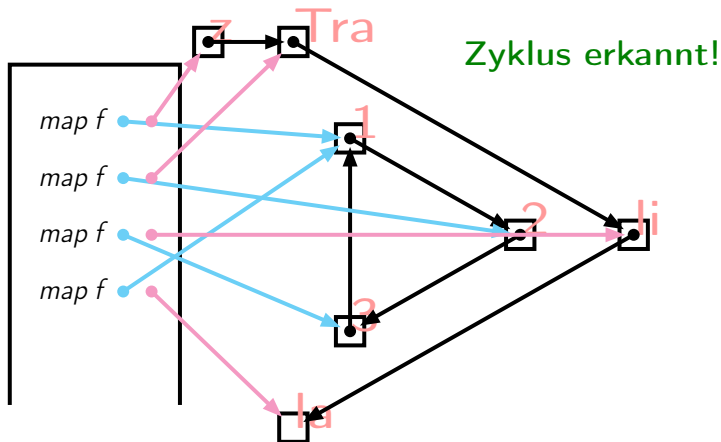
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

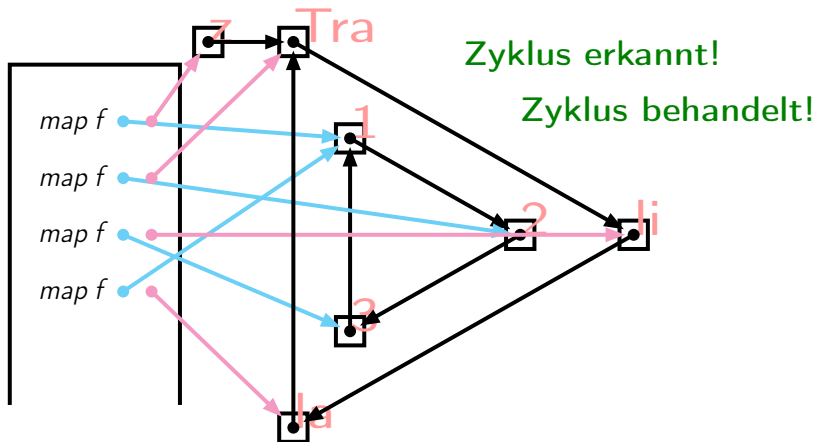
Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Erster Lösungsansatz

Strikt, Ausgabe per Referenz



$$\text{map } f (x : ys) = f x : \text{map } f ys$$

Bewertung des Lösungsansatzes

Erzeugen zyklischer Datenstrukturen

- Geeignete Aufrufkonventionen
- Zyklenerkennung und -behandlung
- Durch lazy unendliche Datenstrukturen approximiert

Durchsuchen zyklischer Datenstrukturen

- Monotone Suchprobleme effektiv lösbar
- Zyklenerkennung und Abbruchfall
 - ▷ Existentielle Probleme (kleinste Fixpunkte)
 - ▷ Universelle Probleme (größte Fixpunkte)
- Mehrstellige Prädikate, Filter, Quicksort uvm. analog
- Durch lazy Rekursion allgemein nur semi-entscheidbar

Bewertung des Lösungsansatzes

Erzeugen zyklischer Datenstrukturen

- Geeignete Aufrufkonventionen
- Zyklenerkennung und -behandlung
- Durch lazy unendliche Datenstrukturen approximiert

Durchsuchen zyklischer Datenstrukturen

- Monotone Suchprobleme effektiv lösbar
- Zyklenerkennung und Abbruchfall
 - ▶ Existentielle Probleme (kleinste Fixpunkte)
 - ▶ Universelle Probleme (größte Fixpunkte)
- Mehrstellige Prädikate, Filter, Quicksort uvm. analog
- **Durch lazy Rekursion allgemein nur semi-entscheidbar**

Bewertung des Lösungsansatzes

Offene Fragen

- 1 Semantik der Datenstrukturen?
- 2 Spezifikation der Strategien?
- 3 Korrektheit der Strategien?
- 4 Unterstützende Umgebung?

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Daten und Signaturen

Prinzip

Generische Handhabung rekursiver Datenstrukturen

- Trennung von lokaler Struktur und Rekursion

Lokale Struktur spezifisch durch Typkonstruktor (besser: *Funktor*)
Typvariable = Blatt-Typ

```
class Functor  $\phi$  where  
  fmap ::  $(\alpha \rightarrow \beta) \rightarrow \phi \alpha \rightarrow \phi \beta$ 
```

Rekursion universell durch Fixpunktoperator

```
newtype Fix  $\phi = In \{ out :: \phi (Fix \phi) \}$ 
```

Daten und Signaturen

Beispiel

Beispiel: Listen

```
data Cell  $\alpha$   $\beta$  = Nil | Cons  $\alpha$   $\beta$   
instance Functor (Cell  $\alpha$ ) where  
    fmap f Nil = Nil  
    fmap f (Cons x y) = Cons x (f y)  
type List  $\alpha$  = Fix (Cell  $\alpha$ )
```

$[] \simeq \text{In } \text{Nil}$

$[1] \simeq \text{In } \$ \text{Cons } 1 \$ \text{In } \text{Nil}$

$[1, 2] \simeq \text{In } \$ \text{Cons } 1 \$ \text{In } \$ \text{Cons } 2 \$ \text{In } \text{Nil}$

Daten und Signaturen

Beispiel

Beispiel: Listen

```
data Cell  $\alpha$   $\beta$  = Nil | Cons  $\alpha$   $\beta$   
instance Functor (Cell  $\alpha$ ) where  
    fmap f Nil           = Nil  
    fmap f (Cons x y) = Cons x (f y)  
type List  $\alpha$  = Fix (Cell  $\alpha$ )
```

$[] \simeq \text{In } \text{Nil}$

$[1] \simeq \text{In } \$ \text{Cons } 1 \$ \text{In } \text{Nil}$

$[1, 2] \simeq \text{In } \$ \text{Cons } 1 \$ \text{In } \$ \text{Cons } 2 \$ \text{In } \text{Nil}$

Signaturen

Intuition

Funktoren für Datenstrukturen

- Benötigen einige zusätzliche Methoden
- Realisiert in der Klasse der *Signaturen*

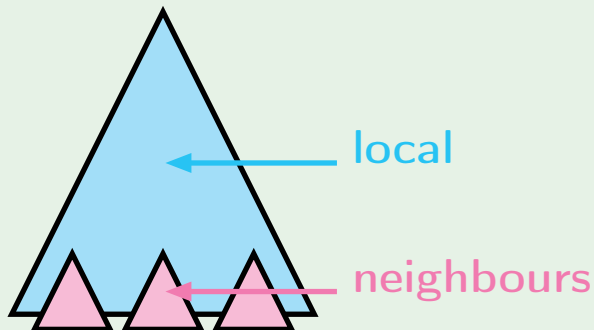
Intuition: Graphstruktur

$$local \quad :: (Signature \sigma) \Rightarrow \sigma \alpha \rightarrow \sigma ()$$
$$neighbours :: (Signature \sigma) \Rightarrow \sigma \alpha \rightarrow [\alpha]$$
$$x \simeq (local\ x, neighbours\ x)$$

Signaturen

Definition

Visualisierung



Signaturen

Definition

Definition

```
class (Functor  $\sigma$ )  $\Rightarrow$  Signature  $\sigma$  where
  (==*)      :: (Eq  $\alpha$ )  $\Rightarrow$   $\sigma$   $\alpha$   $\rightarrow$   $\sigma$   $\alpha$   $\rightarrow$  Bool
  fmapM      :: (Monad  $\mu$ )  $\Rightarrow$  ( $\alpha \rightarrow \mu$   $\beta$ )  $\rightarrow$   $\sigma$   $\alpha$   $\rightarrow$   $\mu$  ( $\sigma$   $\beta$ )
  ffold      :: ( $\alpha \rightarrow \beta \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \sigma$   $\beta \rightarrow \alpha$ 
  fzipWithM_ :: (Monad  $\mu$ )  $\Rightarrow$  ( $\alpha \rightarrow \beta \rightarrow \mu$  ())  $\rightarrow$   $\sigma$   $\alpha \rightarrow \sigma$   $\beta \rightarrow \mu$  ()
```

Abgeleitete Operationen

```
local      = fmap (const ())
neighbours = ffold ( $\lambda$  xs y  $\rightarrow$  xs ++ [y]) []
```

Signaturen

Definition

Definition

```
class (Functor  $\sigma$ )  $\Rightarrow$  Signature  $\sigma$  where
  (==*)      :: (Eq  $\alpha$ )  $\Rightarrow$   $\sigma$   $\alpha$   $\rightarrow$   $\sigma$   $\alpha$   $\rightarrow$  Bool
  fmapM      :: (Monad  $\mu$ )  $\Rightarrow$  ( $\alpha \rightarrow \mu$   $\beta$ )  $\rightarrow$   $\sigma$   $\alpha$   $\rightarrow$   $\mu$  ( $\sigma$   $\beta$ )
  ffold      :: ( $\alpha \rightarrow \beta \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow \sigma$   $\beta \rightarrow \alpha$ 
  fzipWithM_ :: (Monad  $\mu$ )  $\Rightarrow$  ( $\alpha \rightarrow \beta \rightarrow \mu$  ())  $\rightarrow$   $\sigma$   $\alpha \rightarrow \sigma$   $\beta \rightarrow \mu$  ()
```

Abgeleitete Operationen

```
local      = fmap (const ())
neighbours = ffold ( $\lambda$  xs y  $\rightarrow$  xs ++ [y]) []
```

Signaturen

Konstruktoren

Prinzip der Generischen Programmierung

- 4 Konstruktoren genügen für gewöhnliche Datentypen:
Variable, Konstanten, Produkt $((,))$, Coprodukt (*Either*)
- Alle Methoden festgelegt
- Punktfreie Notation

Signaturen

Konstruktoren

Konstruktor: Variable

instance *Signature Identity* **where**

Identity $x ==^* Identity\ y$ $= x == y$

fmapM $f\ (Identity\ x)$ $= liftM\ Identity\ (f\ x)$

ffold $f\ e\ (Identity\ x)$ $= f\ e\ x$

fzipWithM $_ f\ (Identity\ x)\ (Identity\ y)$ $= f\ x\ y$

Signaturen

Konstruktoren

Konstruktor: Konstanten

`newtype` *Const* α $\beta = \text{Const } \alpha$

`instance` *Functor* (*Const* α) **`where`**

fmap *f* = *id*

`instance` (*Eq* α) \Rightarrow *Signature* (*Const* α) **`where`**

Const *c* ==* *Const* *d* = *c* == *d*

fmapM *f* = *return*

ffold *f* = *const*

fzipWithM _ *f* (*Const* *c*) (*Const* *d*) | *c* == *d* = *return* ()

Signaturen

Konstruktoren

Konstruktor: Punktweises Produkt

data $(\otimes) \sigma \tau \alpha = (\otimes) (\sigma \alpha) (\tau \alpha)$

instance (*Functor* σ , *Functor* τ) \Rightarrow *Functor* $(\sigma \otimes \tau)$ **where**

$fmap\ f\ (x \otimes y) = fmap\ f\ x \otimes fmap\ f\ y$

instance (*Signature* σ , *Signature* τ) \Rightarrow *Signature* $(\sigma \otimes \tau)$ **where**

$(w \otimes x) ==^* (y \otimes z) \quad \quad \quad = w ==^* y \ \&\& \ x ==^* z$

$fmapM\ f\ (x \otimes y) \quad \quad \quad =$

$liftM2\ (\otimes)\ (fmapM\ f\ x)\ (fmapM\ f\ y)$

$ffold\ f\ e\ (x \otimes y) \quad \quad \quad = ffold\ f\ (ffold\ f\ e\ x)\ y$

$fzipWithM_f\ (w \otimes x)\ (y \otimes z) =$

$fzipWithM_f\ w\ y \gg fzipWithM_f\ x\ z$

Signaturen

Konstruktoren

Konstruktor: Punktweises Coprodukt

data $(\oplus) \sigma \tau \alpha = Lt (\sigma \alpha) \mid Rt (\tau \alpha)$

instance (*Functor* σ , *Functor* τ) \Rightarrow *Functor* $(\sigma \oplus \tau)$ **where**

$fmap\ f\ (Lt\ x) = Lt\ (fmap\ f\ x)$

$fmap\ f\ (Rt\ y) = Rt\ (fmap\ f\ y)$

instance (*Signature* σ , *Signature* τ) \Rightarrow *Signature* $(\sigma \oplus \tau)$ **where**

\vdots

$fzipWithM_ f\ (Lt\ x)\ (Lt\ y) = fzipWithM_ f\ x\ y$

$fzipWithM_ f\ (Rt\ x)\ (Rt\ y) = fzipWithM_ f\ x\ y$

Signaturen

Konstruktoren

Beispiel: Listen (Wiederholung)

type $Cell\ \alpha = Const\ () \oplus (Const\ \alpha \otimes Identity)$

type $List\ \alpha = Fix\ (Cell\ \alpha)$

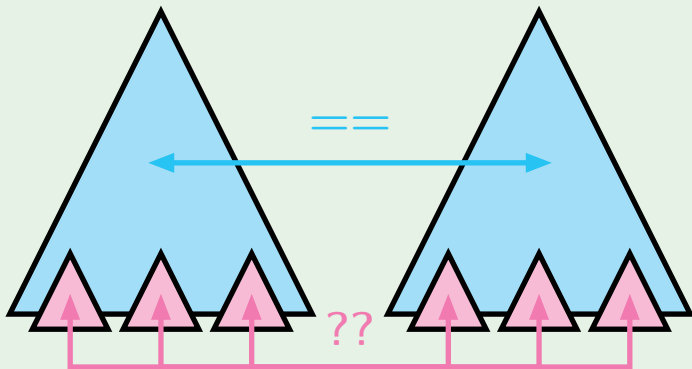
$[] \simeq In\ \$\ Lt\ \$\ Const\ ()$

$[1] \simeq In\ \$\ Rt\ \$\ (Const\ 1 \otimes Identity\ \$\ (In\ \$\ Lt\ \$\ Const\ ()))$

Signaturen

Relationen auf Daten

Individualtransformationen



Daten

Individualtransformtionen

type *ITrans* $\alpha \beta = [(\alpha, \beta)]$

trans :: (*Signature* σ) $\Rightarrow \sigma \alpha \rightarrow \sigma \beta \rightarrow ITrans \alpha \beta$

trans $x y = execState (fzipWithM_upd x y) []$

where

upd $v w = modify (+ + [(v, w)])$

cover :: (*Monad* μ) $\Rightarrow (\alpha \rightarrow \beta \rightarrow \mu ()) \rightarrow ITrans \alpha \beta \rightarrow \mu ()$

cover = *mapM_* \circ *uncurry*

Daten

Individualtransformtionen

type *ITrans* $\alpha \beta = [(\alpha, \beta)]$

trans :: (*Signature* σ) $\Rightarrow \sigma \alpha \rightarrow \sigma \beta \rightarrow$ *ITrans* $\alpha \beta$

trans $x y = \text{execState } (\text{fzipWithM_upd } x y) []$

where

upd $v w = \text{modify } (++) [(\mathbf{v}, w)]$

cover :: (*Monad* μ) $\Rightarrow (\alpha \rightarrow \beta \rightarrow \mu ()) \rightarrow$ *ITrans* $\alpha \beta \rightarrow \mu ()$

cover = *mapM_* \circ *uncurry*

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Speicherzustand

Aufrufkonventionen

Strikte Auswertung mit Ausgabe per Referenz

- Prozedural
 - Eingabe per Referenz auf Zelle
 - Ausgabe per Referenz auf Feld
 - Seiteneffekt Initialisieren der Ausgabefelder
- Reihenfolge kritisch
 - ▶ Erst Ausgabe initialisieren, dann Rekursion
- Ein Fall für (Zustands-)Monaden!

Speicherzustand

Struktur

Speicher als Datenstruktur

- Zwei Adressräume: Zellen, Felder
- Belegung explizit
- Zellenformat als Signatur gegeben

newtype *Cell* = *Cell Int*

newtype *Field* = *Cell Int*

data (*Signature* σ) \Rightarrow *Heap* σ =

Heap { *nextCell* :: *Cell*,
 nextField :: *Field*,
 cells :: *Map Cell* (σ *Field*),
 fields :: *Map Field Cell*}

Speicherzustand

Operationen

Elementarer Speicherzugriff als monadische Operationen

```
type HeapOp  $\sigma$  = State (Heap  $\sigma$ )  
freshCell :: (Signature  $\sigma$ )  $\Rightarrow$  HeapOp  $\sigma$  Cell  
freshField :: (Signature  $\sigma$ )  $\Rightarrow$  HeapOp  $\sigma$  Field  
setCell :: (Signature  $\sigma$ )  $\Rightarrow$  Cell  $\rightarrow$   $\sigma$  Field  $\rightarrow$  HeapOp  $\sigma$  ()  
setField :: (Signature  $\sigma$ )  $\Rightarrow$  Field  $\rightarrow$  Cell  $\rightarrow$  HeapOp  $\sigma$  ()  
getCell :: (Signature  $\sigma$ )  $\Rightarrow$  Cell  $\rightarrow$  HeapOp  $\sigma$  ( $\sigma$  Field)  
getField :: (Signature  $\sigma$ )  $\Rightarrow$  Field  $\rightarrow$  HeapOp  $\sigma$  Cell
```

Speicherzustand

Operationen

Zentrale Operation 1: Erzeugen von Daten

- Kontextfrei

$newCell \quad :: (Signature \sigma) \Rightarrow \sigma \alpha \rightarrow HeapOp \sigma (Cell, ITrans Field \alpha)$

$newCell \ s = \text{do}$

$\quad x \leftarrow freshCell$

$\quad t \leftarrow fmapM (const freshField) \ s$

$\quad setCell \ x \ t$

$\quad return \ (x, trans \ t \ s)$

Speicherzustand

Operationen

Zentrale Operation 2: Speichern von Daten

- Im Kontext einer Ausgabevariable (Aufrufkonvention)

```

gen      :: (Signature  $\sigma$ )  $\Rightarrow$  Field  $\rightarrow$   $\sigma$   $\alpha \rightarrow$  HeapOp  $\sigma$  (ITrans Field  $\alpha$ )
gen  $\nu$  s = do
              (y, fs)  $\leftarrow$  newCell s
              setField  $\nu$  y
              return fs
  
```

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - **Auswertung**
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Gerüst einer Funktion

Interpretation

Interpretation

- Vergleiche E. Meijer, *FP With Bananas, Lenses et al.*

$$\begin{aligned}
 \text{map}_0 (f, []) &= [] \\
 \text{map}_0 (f, x : ys) &= f \ x : \underbrace{(f, ys)}_{\leftarrow} \\
 \text{map} &= \llbracket \text{map}_0 \rrbracket
 \end{aligned}$$

- Monadisch über Speicherzustand, Aufrufkonvention

type *Interp* $\sigma \ \tau = \tau \ \text{Cell} \rightarrow \text{HeapOp} \ \sigma \ (\sigma \ (\tau \ \text{Cell}))$

type *Function* $\sigma \ \tau = \text{Field} \rightarrow \tau \ \text{Cell} \rightarrow \text{HeapOp} \ \sigma \ ()$

Funktionsberechnung

Strategie

Funktionsberechnung

$eval :: (Signature\ \sigma, Signature\ \tau) \Rightarrow Interp\ \sigma\ \tau \rightarrow Function\ \sigma\ \tau$

$eval\ i = recur\ []$

where

$recur\ q\ v\ x =$

$let\ recur' = recur\ ((v, x) : q)$

$cycle\ (_, y) = x ==> y$

in case find cycle q of

$Just\ (u, _) \rightarrow getField\ u \gg= setField\ v$

$Nothing \rightarrow i\ x \gg= gen\ v \gg= cover\ recur'$

Funktionsberechnung

Strategie

Funktionsberechnung mit Stack

$eval :: (Signature\ \sigma, Signature\ \tau) \Rightarrow Interp\ \sigma\ \tau \rightarrow Function\ \sigma\ \tau$

$eval\ i = recur\ []$

where

$recur\ q\ v\ x =$

let $recur'$ $= recur\ ((v, x) : q)$

$cycle\ (_, y) = x ==^* y$

in $case\ find\ cycle\ q\ of$

$Just\ (u, _) \rightarrow getField\ u \gg= setField\ v$

$Nothing \rightarrow i\ x \gg= gen\ v \gg= cover\ recur'$

Funktionsberechnung

Strategie

Funktionsberechnung mit Zyklenerkennung

$eval :: (Signature\ \sigma, Signature\ \tau) \Rightarrow Interp\ \sigma\ \tau \rightarrow Function\ \sigma\ \tau$
 $eval\ i = recur\ []$

where

$recur\ q\ v\ x =$

let $recur' = recur\ ((v, x) : q)$

$cycle\ (_, y) = x ==^* y$

in case find cycle q of

$Just\ (u, _) \rightarrow getField\ u \gg= setField\ v$

$Nothing \rightarrow i\ x \gg= gen\ v \gg= cover\ recur'$

Funktionsberechnung

Strategie

Funktionsberechnung mit Zyklenerkennung & -behandlung

$eval :: (Signature\ \sigma, Signature\ \tau) \Rightarrow Interp\ \sigma\ \tau \rightarrow Function\ \sigma\ \tau$
 $eval\ i = recur\ []$

where

$recur\ q\ v\ x =$

let $recur' = recur\ ((v, x) : q)$

$cycle\ (_, y) = x ==^* y$

in case find cycle q of

$Just\ (u, _) \rightarrow getField\ u \gg= setField\ v$

$Nothing \rightarrow i\ x \gg= gen\ v \gg= cover\ recur'$

Funktionsberechnung

Korrektheit

Korrektheitsnachweis

- Führe $eval\ i\ v\ x$ aus
- Berechnung terminiert für endlich zyklische Eingabe x
- Betrachte den Nachzustand des Speichers als σ -Coalgebra
- Nun hat Ausgabe v finale Semantik $\llbracket i \rrbracket(x)$ (Anamorphismus)
- Beweis durch Graphfärbung & Coinduktion

Funktionsberechnung

Korrektheit

Beispiel: Listen (Demonstration)

```
singWaltz = do  
    v ← freshField  
    x ← createWaltz  
    eval (map0 song) v (Identity x)  
    getField v
```

Suchprobleme

Überblick

Suchprobleme

- Ähnliche Strategie
 - ▶ Erzeugt aus nichtrekursivem „Gerüst“
 - ▶ Spezieller Rekursionsoperator
 - ▶ Zyklenerkennung & -behandlung
- Zur Vereinfachung hier nicht-monadisch

Gerüst einer Suche

Kalkül

Kalkül

- Hilbert-Kalkül definiert rekursive Prädikate
- Auf zyklischen Daten inhärent mehrdeutig
 - ▶ Monoton \implies Verband von Fixpunktsemantiken (Tarski)
 - ▶ Ausgezeichneter kleinster & größter Fixpunkt
 - ▶ Zyklenbehandlung durch Abbruchwert f/w
- Repräsentierbar als Resolutionsschritt in DNF

type $Calc\ \alpha = \alpha \rightarrow [[\alpha]]$
 $dnf :: (\alpha \rightarrow Bool) \rightarrow [[\alpha]] \rightarrow Bool$
 $dnf = any \circ all$

Gerüst einer Suche

Kalkül

Beispiel: Listen (Endlichkeit)

$$\frac{}{fin([])} \qquad \frac{fin(ys)}{fin(x : ys)} \qquad \frac{inf(ys)}{inf(x : ys)}$$

Kleinsten Fixpunkt für *fin*, größter für *inf*

$$\begin{array}{ll} k(Fin []) &= [[]] & e(Fin_)=False \\ k(Fin (x : ys)) &= [[Fin ys]] \\ k(Inf []) &= [] & e(Inf_)=True \\ k(Inf (x : ys)) &= [[Inf ys]] \end{array}$$

Suche

search :: (*Eq* α) \Rightarrow *Calc* $\alpha \rightarrow (\alpha \rightarrow \text{Bool}) \rightarrow \alpha \rightarrow \text{Bool}$

search *k* *e* = *recur* []

where

recur *q* *x* =

let *recur'* = *recur* (*x* : *q*)

in case *any* (*==* *x*) *q* of

True \rightarrow *e* *x*

False \rightarrow *dnf recur' \$ k x*

Suche mit Stack

$search \quad :: (Eq \alpha) \Rightarrow Calc \alpha \rightarrow (\alpha \rightarrow Bool) \rightarrow \alpha \rightarrow Bool$

$search \ k \ e = recur \ []$

where

$recur \ q \ x =$

let $recur' = recur \ (x : q)$

in $case \ any \ (== \ x) \ q \ of$

$True \rightarrow e \ x$

$False \rightarrow dnf \ recur' \ \$ \ k \ x$

Suche mit Zyklenerkennung

$search \quad :: (Eq \alpha) \Rightarrow Calc \alpha \rightarrow (\alpha \rightarrow Bool) \rightarrow \alpha \rightarrow Bool$

$search \ k \ e = recur \ []$

where

$recur \ q \ x =$

let $recur' = recur \ (x : q)$

in case $any \ (== \ x) \ q$ **of**

$True \rightarrow e \ x$

$False \rightarrow dnf \ recur' \ \$ \ k \ x$

Suche mit Zyklenerkennung & -behandlung

$search \quad :: (Eq \alpha) \Rightarrow Calc \alpha \rightarrow (\alpha \rightarrow Bool) \rightarrow \alpha \rightarrow Bool$

$search \ k \ e = recur \ []$

where

$recur \ q \ x =$

let $recur' = recur \ (x : q)$

in case $any \ (== \ x) \ q$ **of**

$True \rightarrow e \ x$

$False \rightarrow dnf \ recur' \ \$ \ k \ x$

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Laufzeitunterstützung

Warum Virtuelle Maschine?

- Aufrufkonvention von wenigen Sprachen unterstützt
- Manipulation des Stacks problematisch
 - ▶ Verdirbt Optimierungen (*inlining*, *tail calls*)
 - ▶ Sicherheitslücke
 - ▶ von Sprachen nicht unterstützt
- Zyklenerkennung & -behandlung immer gleich
 - ▶ Als primitive Operationen gewünscht

Modellierung in Haskell

- Als eingebettete domänenspezifische Sprache

Laufzeitunterstützung

Warum Virtuelle Maschine?

- Aufrufkonvention von wenigen Sprachen unterstützt
- Manipulation des Stacks problematisch
 - ▶ Verdirbt Optimierungen (*inlining*, *tail calls*)
 - ▶ Sicherheitslücke
 - ▶ von Sprachen nicht unterstützt
- Zyklenerkennung & -behandlung immer gleich
 - ▶ Als primitive Operationen gewünscht

Modellierung in Haskell

- Als eingebettete domänenspezifische Sprache

Virtuelle Maschine

Modellierung (1)

- Universelle Datensignatur

```
data Univ  $\alpha$  = Tuple [ $\alpha$ ]
                | Tag String  $\alpha$ 
                | Closure Function [ $\alpha$ ]
instance Signature Univ where ...
```

- Erweiterung des Speichers um Stack und Konstantenpool

```
data Memory = Memory (Heap Univ) Stack CPool
```

Virtuelle Maschine

Modellierung (1)

- Universelle Datensignatur

```
data Univ  $\alpha$  = Tuple [ $\alpha$ ]
                | Tag String  $\alpha$ 
                | Closure Function [ $\alpha$ ]
instance Signature Univ where ...
```

- Erweiterung des Speichers um Stack und Konstantenpool

```
data Memory = Memory (Heap Univ) Stack CPool
```

Virtuelle Maschine

Modellierung (2)

- Befehlssatz als freier Datentyp

```
data Statement = Return | Pop | Ditto | ...
```

Virtuelle Maschine

Modellierung (2)

- Befehlssatz als freier Datentyp

data *Statement* = *Return* | *Pop* | *Ditto* | ...

- Ausführungszustand als Monade

data *Progress* α = *Halted*
 | *Returned Memory*
 | *Running* α

- Berechnung mit Zustand

type Computation = *StateT Memory Progress*

- Interpreter als monadische Operation

execStatement :: *Statement* \rightarrow *Computation* ()

Virtuelle Maschine

Modellierung (2)

- Befehlssatz als freier Datentyp

data *Statement* = *Return* | *Pop* | *Ditto* | ...

- Ausführungszustand als Monade

data *Progress* α = *Halted*
 | *Returned Memory*
 | *Running* α

- Berechnung mit Zustand

type *Computation* = *StateT Memory Progress*

- Interpreter als monadische Operation

execStatement :: *Statement* \rightarrow *Computation* ()

Virtuelle Maschine

Modellierung (2)

- Befehlssatz als freier Datentyp

data *Statement* = *Return* | *Pop* | *Ditto* | ...

- Ausführungszustand als Monade

data *Progress* α = *Halted*
 | *Returned Memory*
 | *Running* α

- Berechnung mit Zustand

type *Computation* = *StateT Memory Progress*

- Interpreter als monadische Operation

execStatement :: *Statement* \rightarrow *Computation* ()

Übersicht

- 1 Einführung
 - Kontext
 - Das Problem
 - Lösungsansatz
- 2 Spezifikation von Strategien
 - Daten und Signaturen
 - Speicherzustand
 - Auswertung
- 3 Spezifikation einer Virtuellen Maschine
- 4 Zusammenfassung

Zusammenfassung

Haskell als „bessere Mathematik“?

- *Literate Programming* auf den Kopf gestellt
- Strukturiert: Module, Datentypen, Monaden
- Generisch: höhere Ordnung, eingebettete Sprachen
- verschwenderische Abstraktion, Effizienz (hier) irrelevant
- Typcheck **unbezahlbar**
- Unit-Test wertvoll für Intuition und Validierung

Diskussion!