Caught in a Vicious Circle

Haskell als reine Spezifikationssprache

Baltasar Trancón y Widemann

Universität Bayreuth

HaL 4
12. Juni 2009

Übersicht

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

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

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"
```

$$take\ 11\ (concat\ song) \rightsquigarrow \texttt{"TralilaTral"}$$

Das Problem

Rekursion auf zyklischen Eingabedaten

$$waltz = 1 : 2 : 3 : waltz$$

. . . oder doch?

$$4\ `elem`\ waltz \quad \leadsto \bot \quad \text{— einstellige Prädikate}$$
$$waltz == cycle\ [1..3] \quad \leadsto \bot \quad \text{— mehrstellige Prädikate}$$
$$filter\ (> 3)\ waltz \quad \leadsto \bot \quad \text{— nichtproduktive Funktionen}$$
$$sort\ waltz \quad \leadsto \bot \quad \text{— nichtlineare Funktionen}$$

Das Problem

Rekursion auf zyklischen Eingabedaten

$$waltz = 1 : 2 : 3 : waltz$$

. . . oder doch?

$$\begin{array}{lll} 4\ `elem`\ waltz & \rightsquigarrow \bot & \text{— einstellige Prädikate} \\ waltz == cycle\ [1..3] & \rightsquigarrow \bot & \text{— mehrstellige Prädikate} \\ filter\ (>3)\ waltz & \rightsquigarrow \bot & \text{— nichtproduktive Funktionen} \\ sort\ waltz & \rightsquigarrow \bot & \text{— nichtlineare Funktionen} \end{array}$$

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

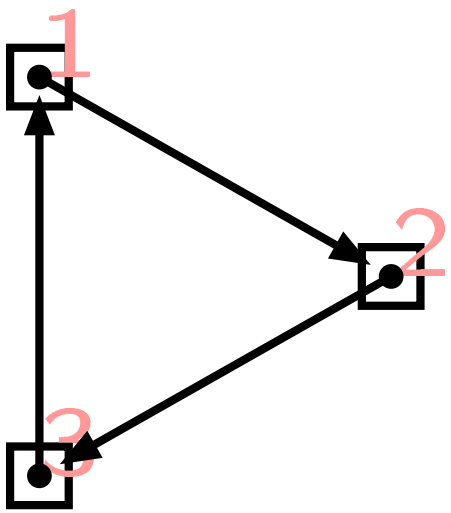
Those who cannot remember the past are condemned to repeat it.

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

Übersicht

Erster Lösungsansatz

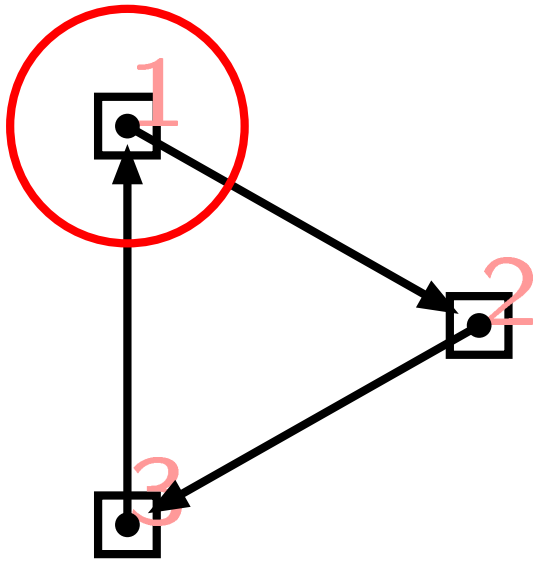
Lazy



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

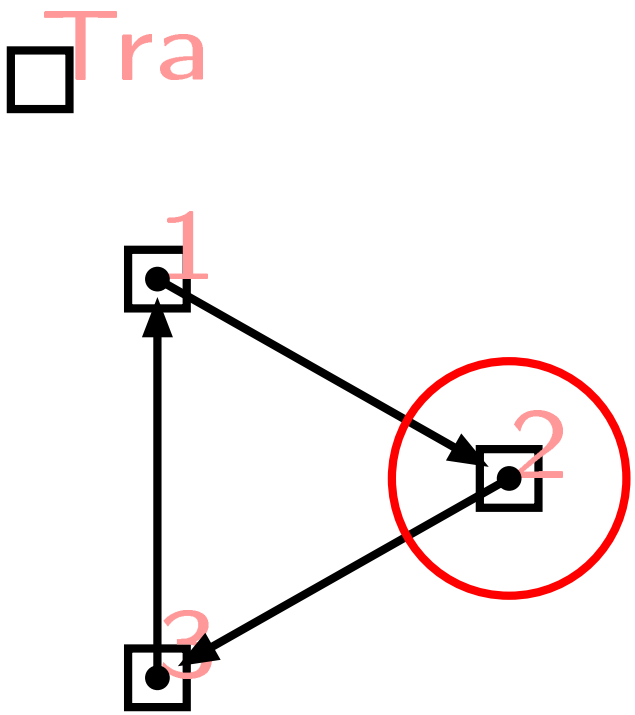
Lazy



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

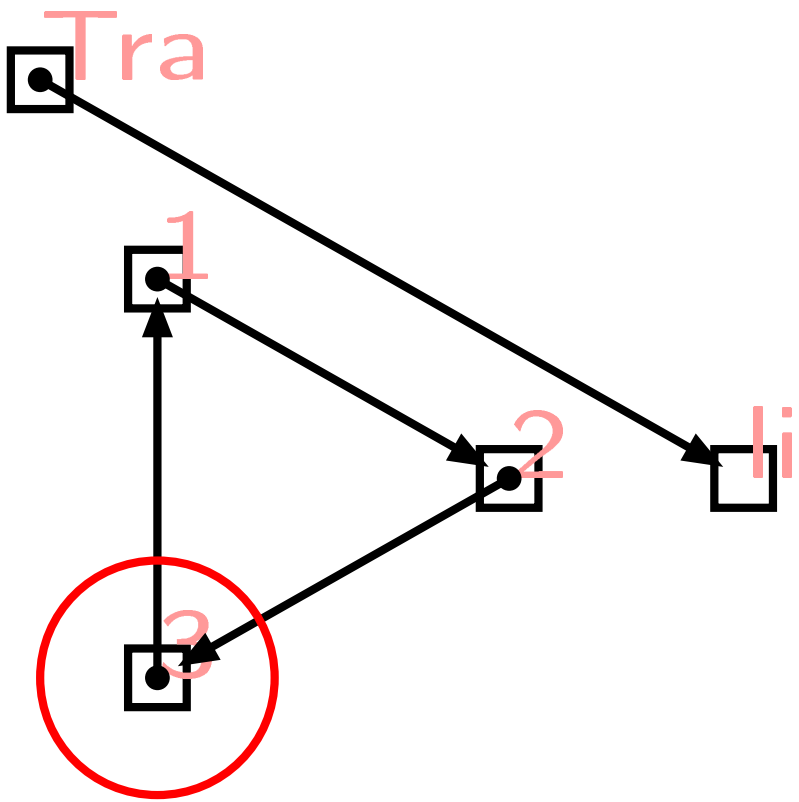
Lazy



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

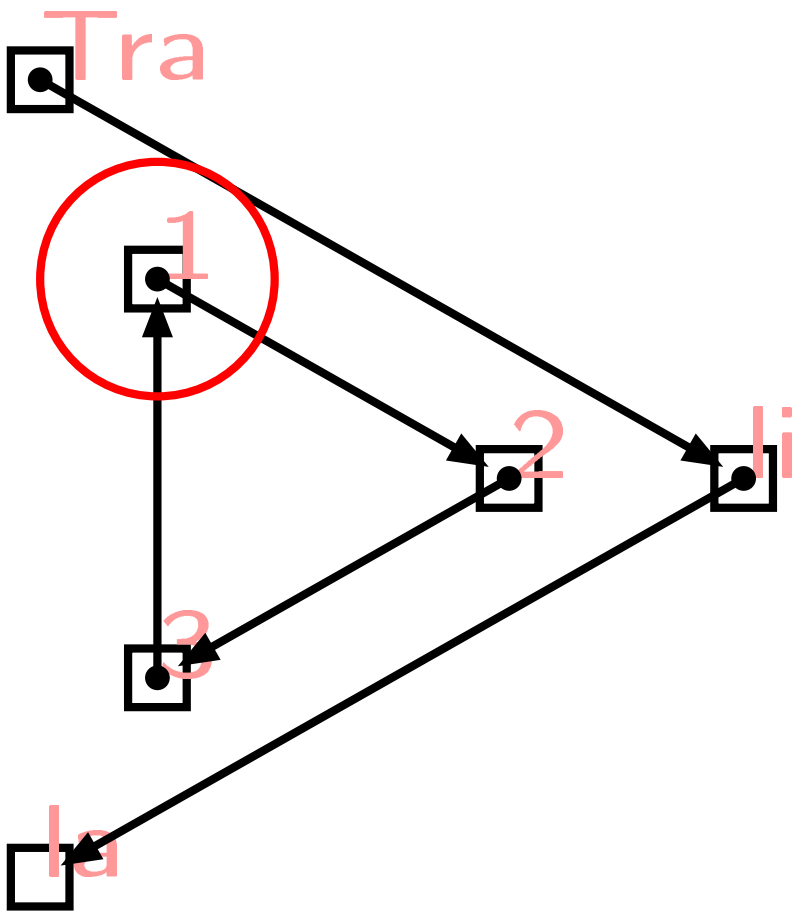
Lazy



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

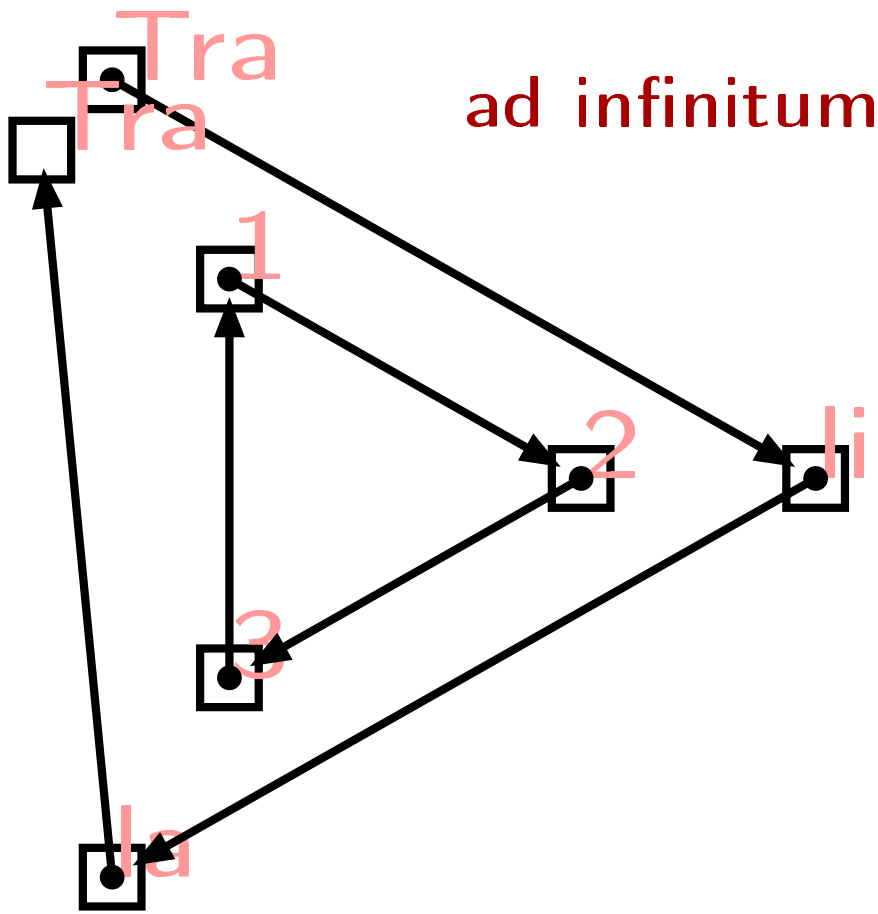
Lazy



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

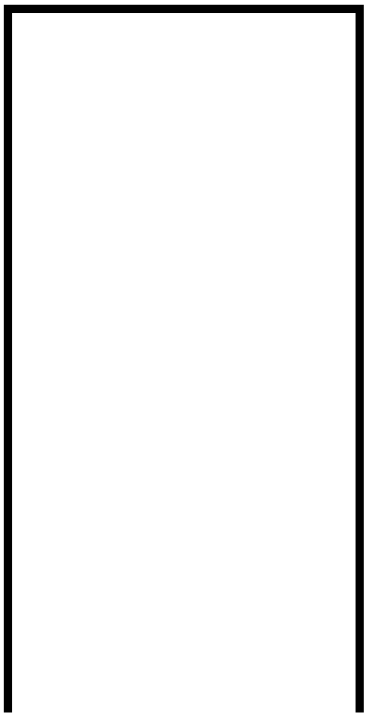
Lazy

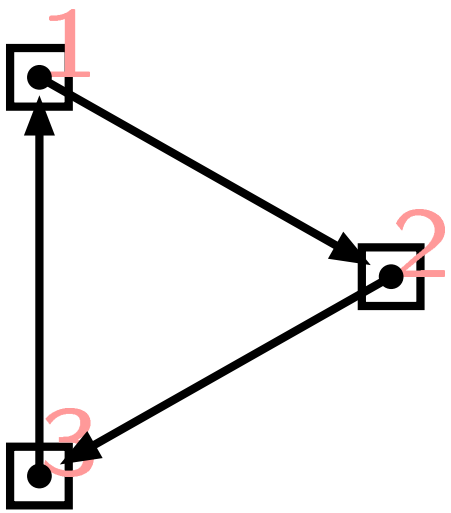


$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

Strikt

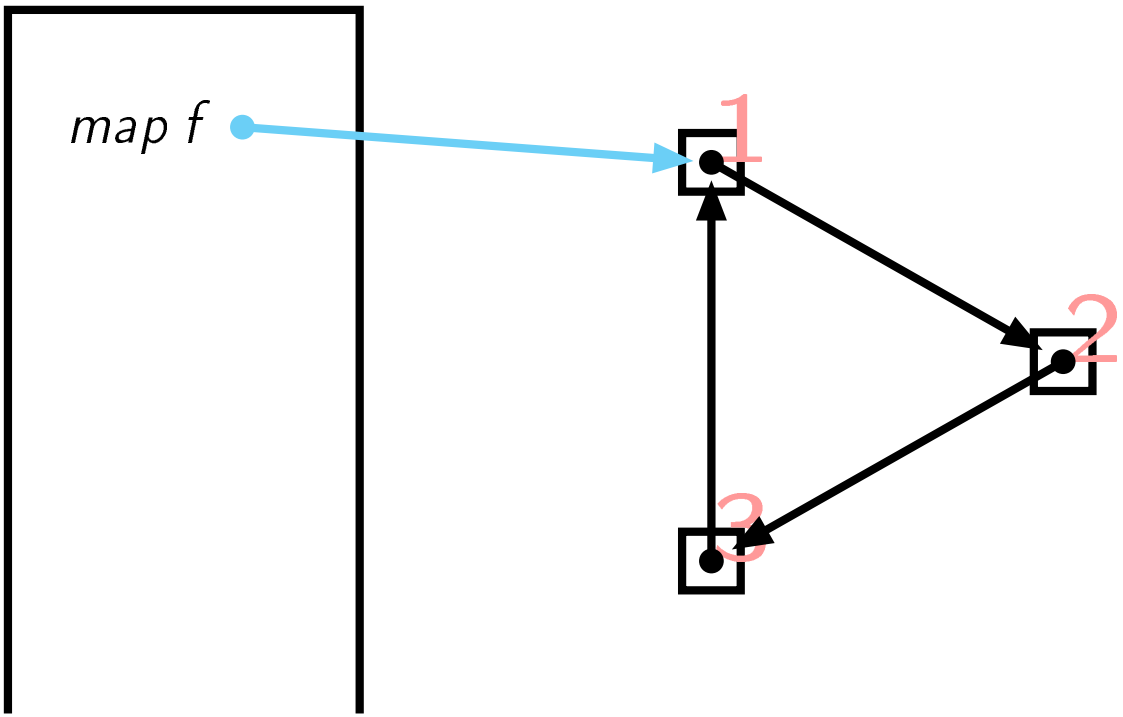




$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

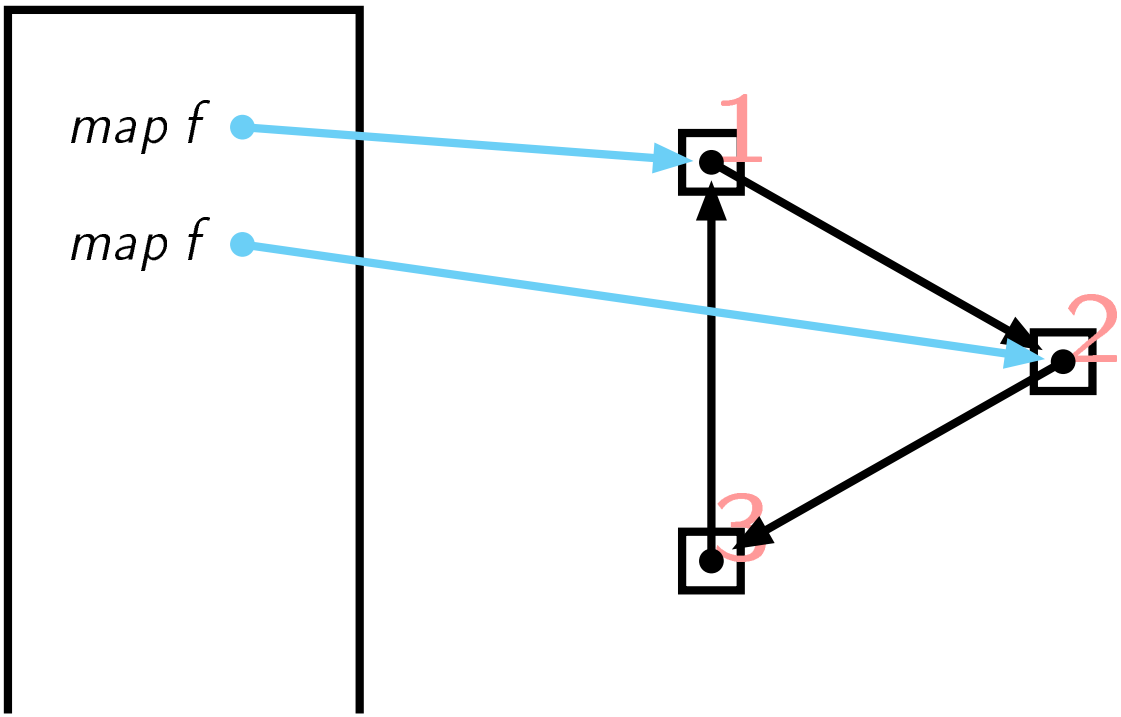
Strikt



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

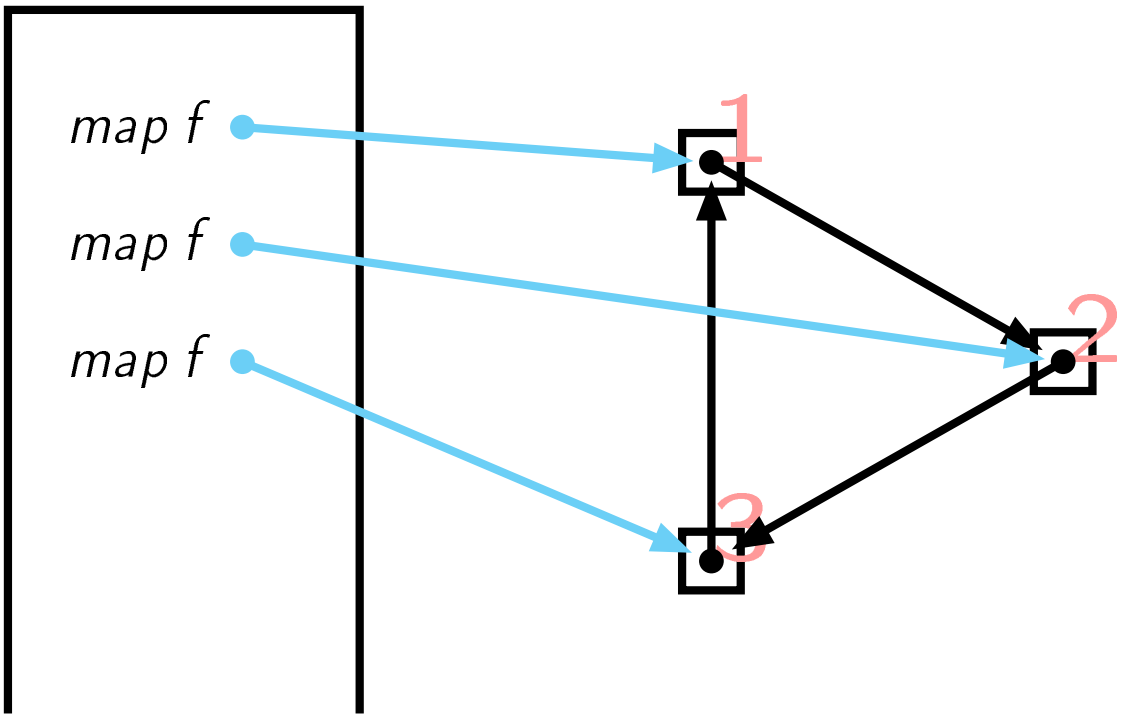
Strikt



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

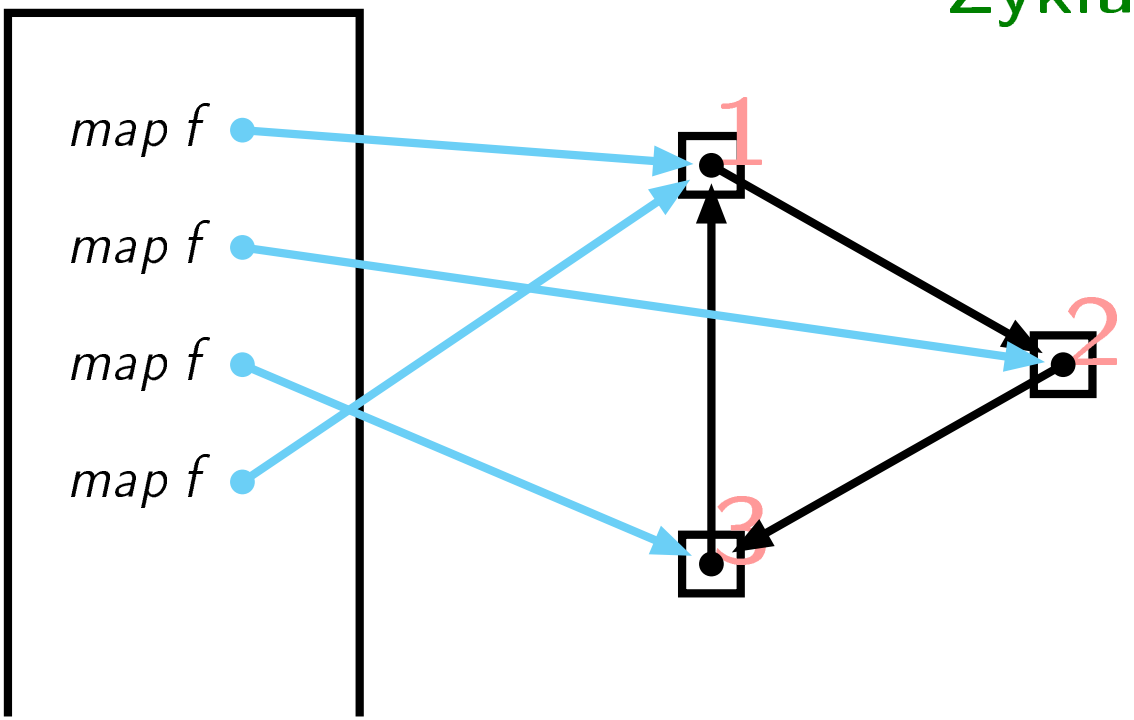
Erster Lösungsansatz

Strikt



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz
Strikt

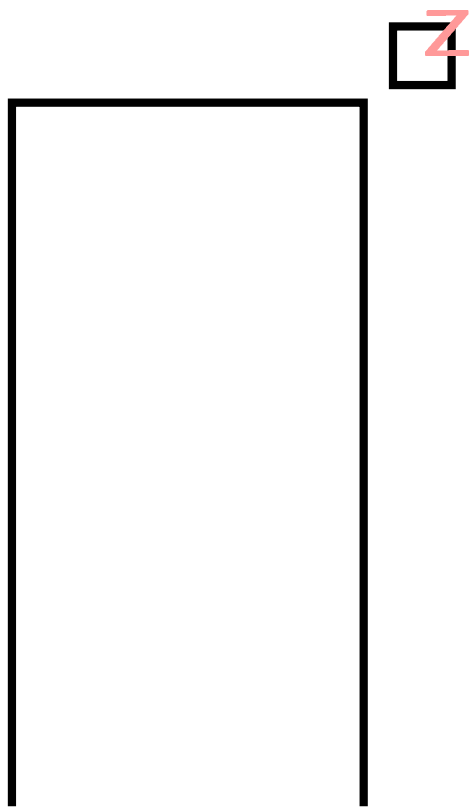


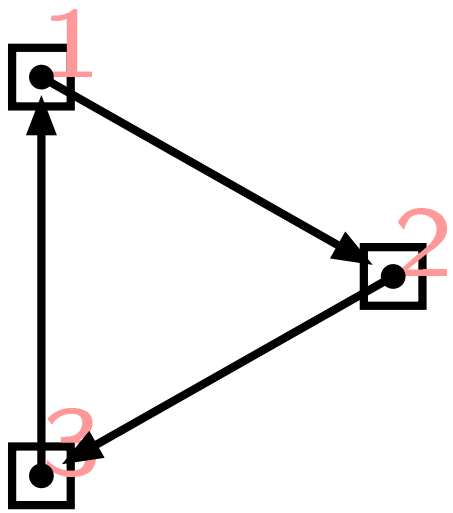
Zyklus erkannt!

$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

Strikt, Ausgabe per Referenz

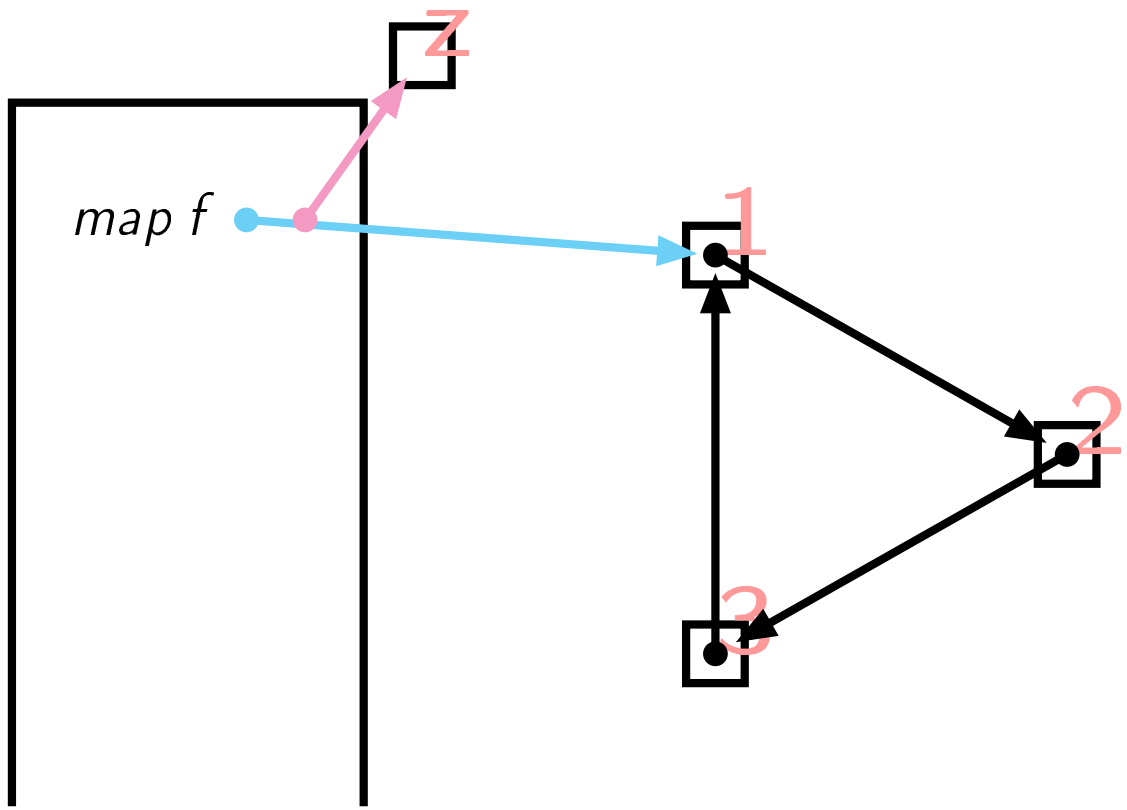




$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

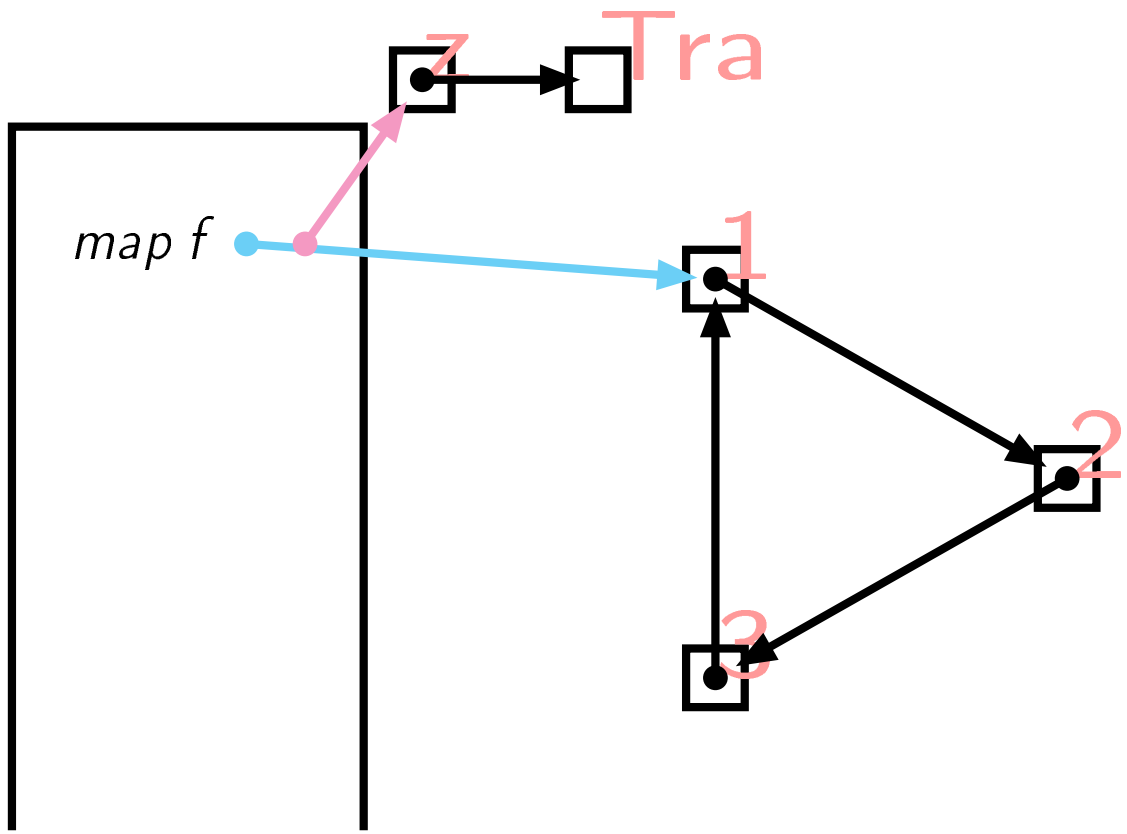
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

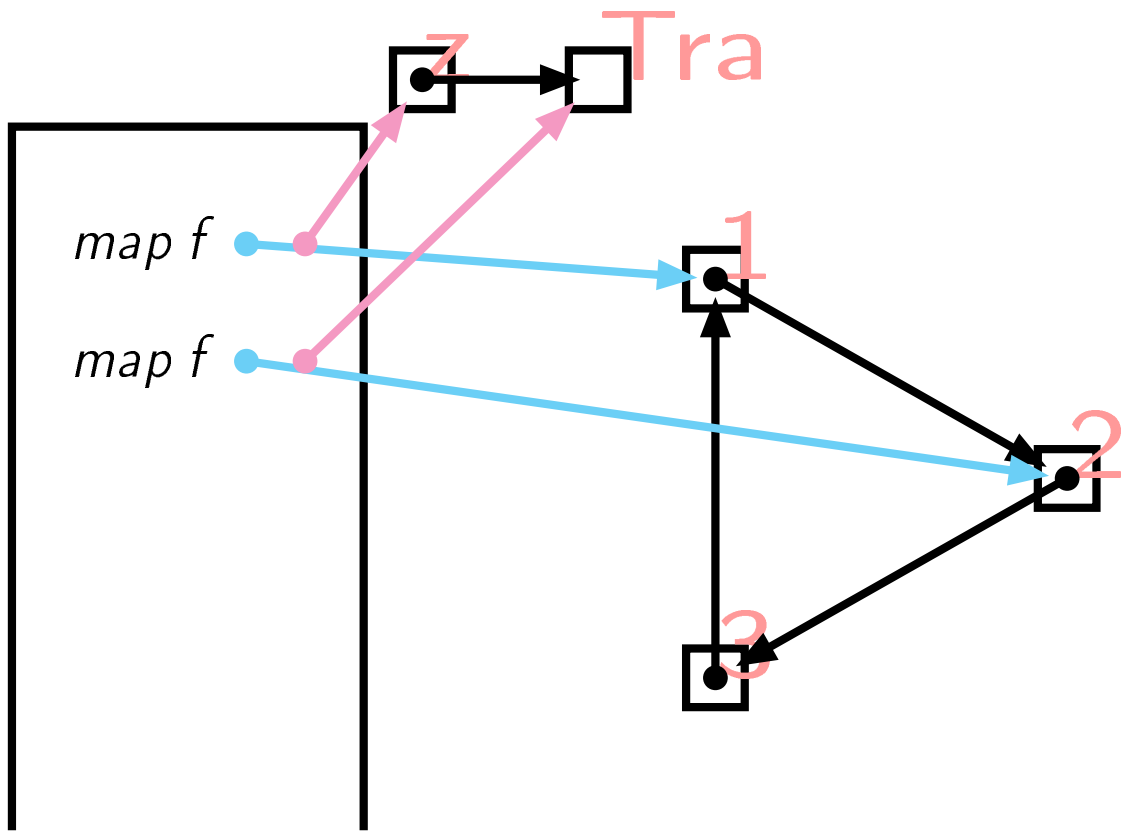
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

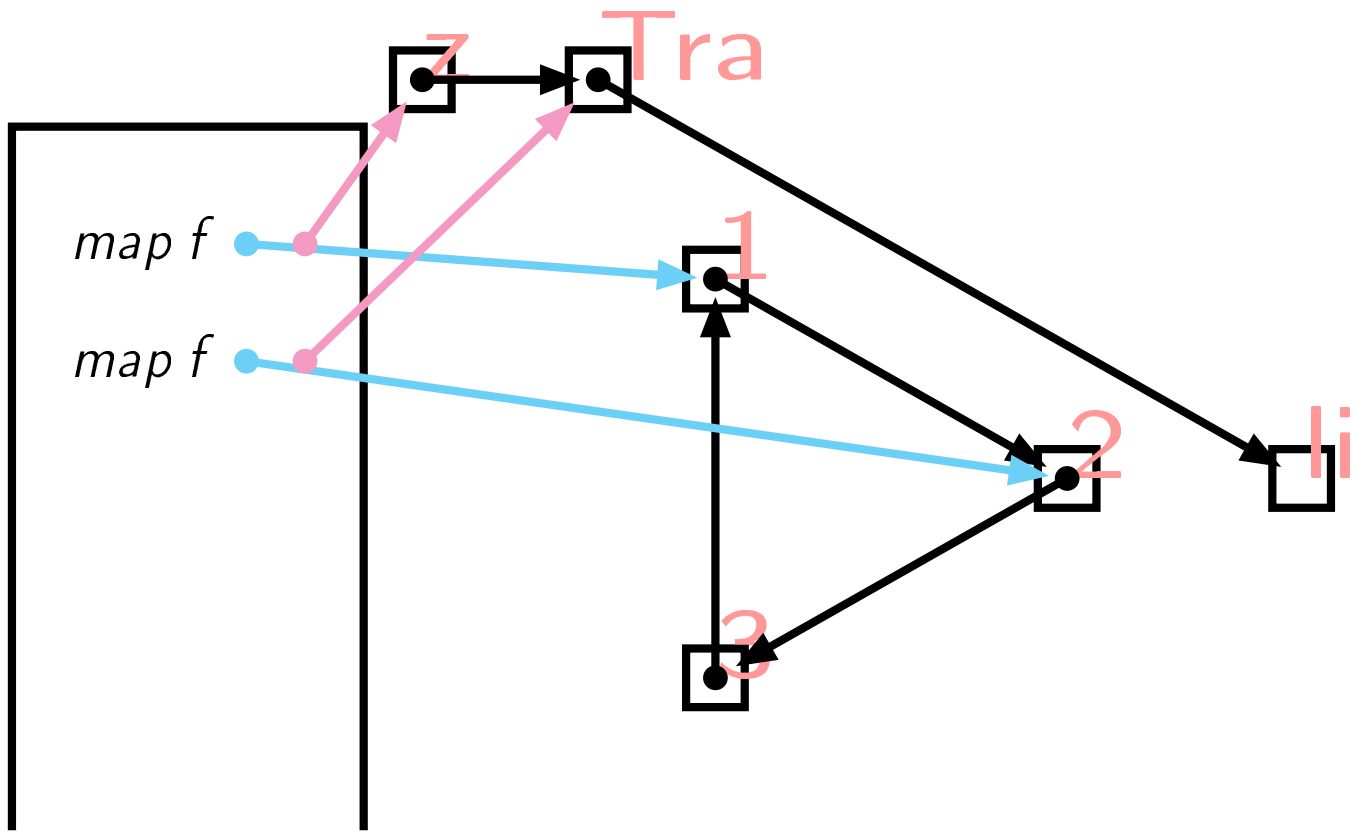
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

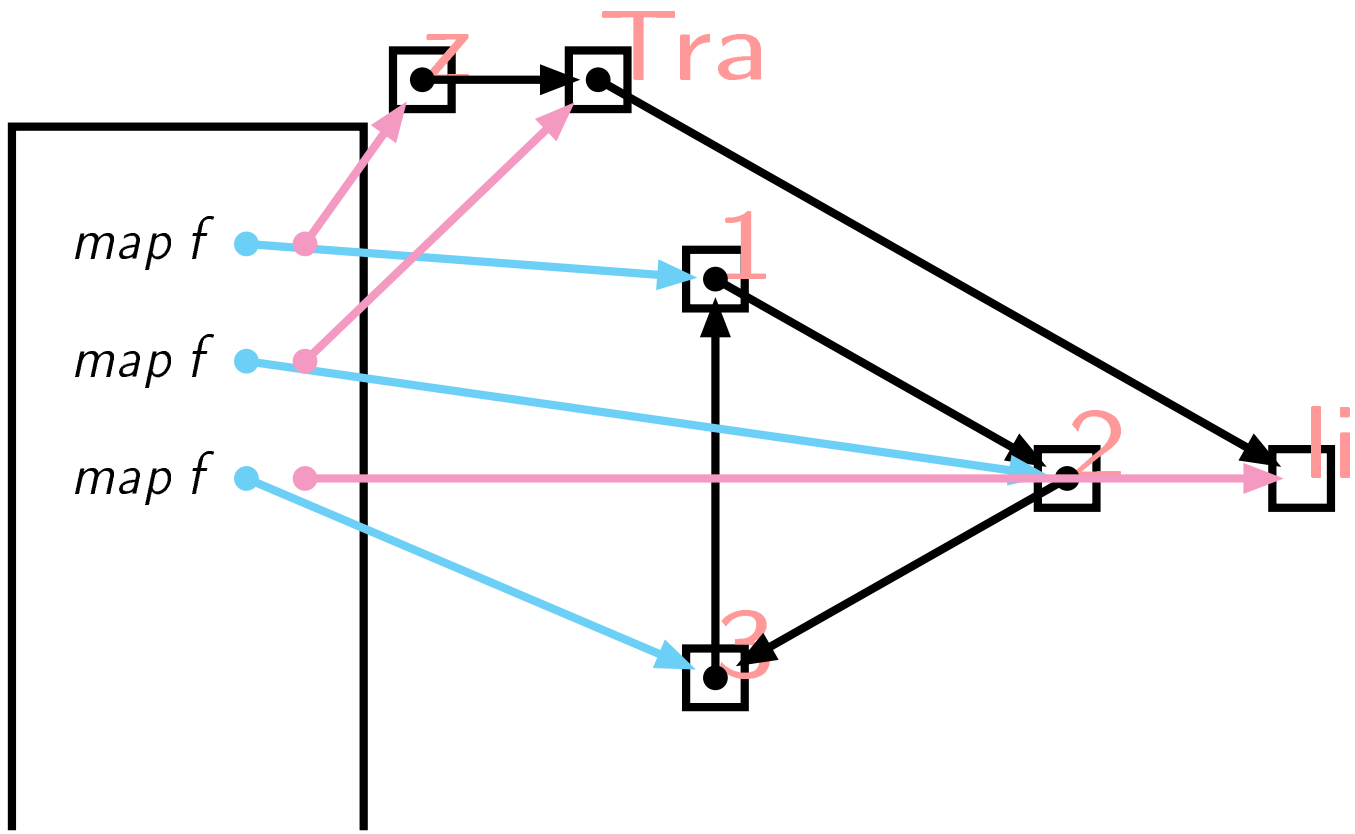
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

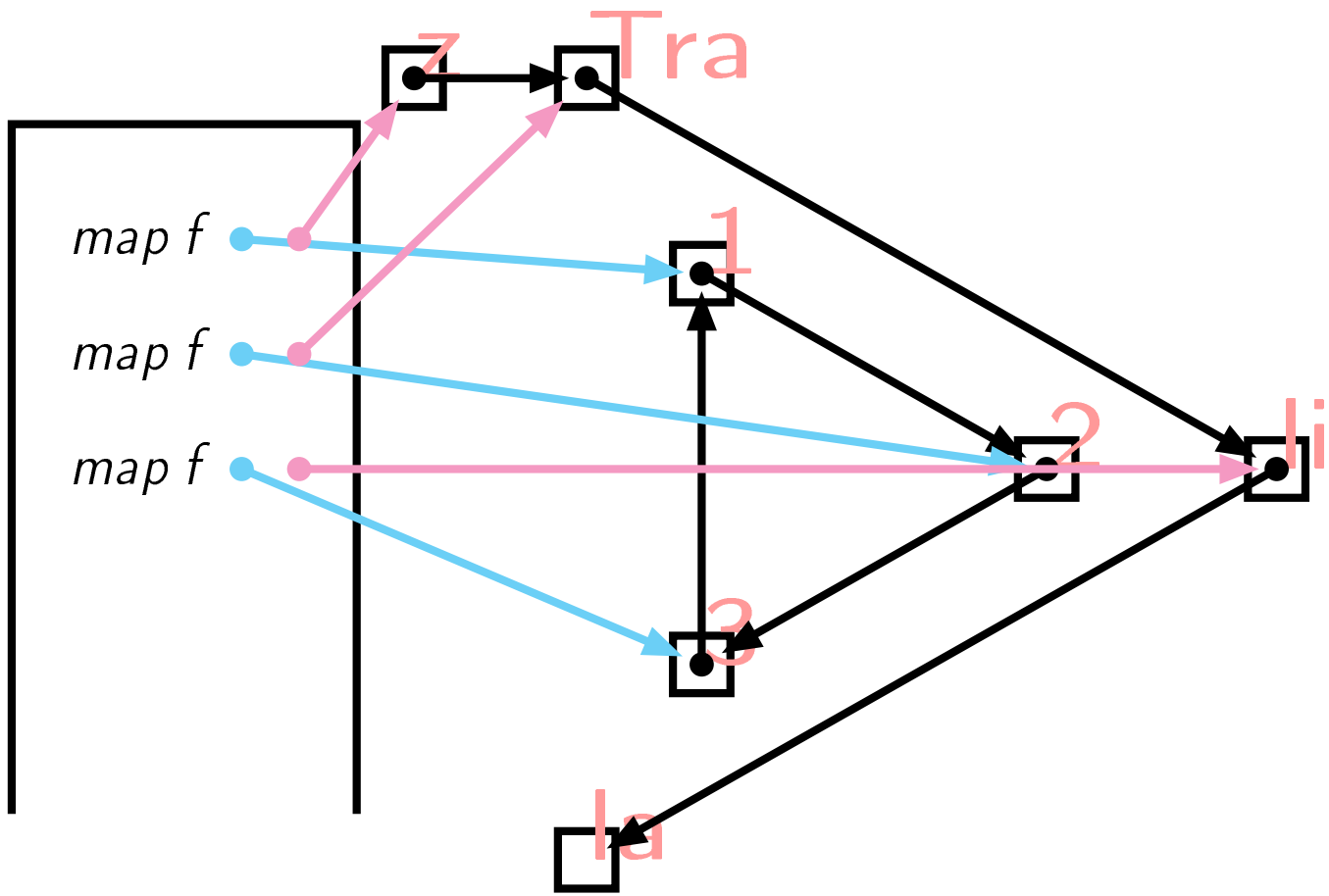
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

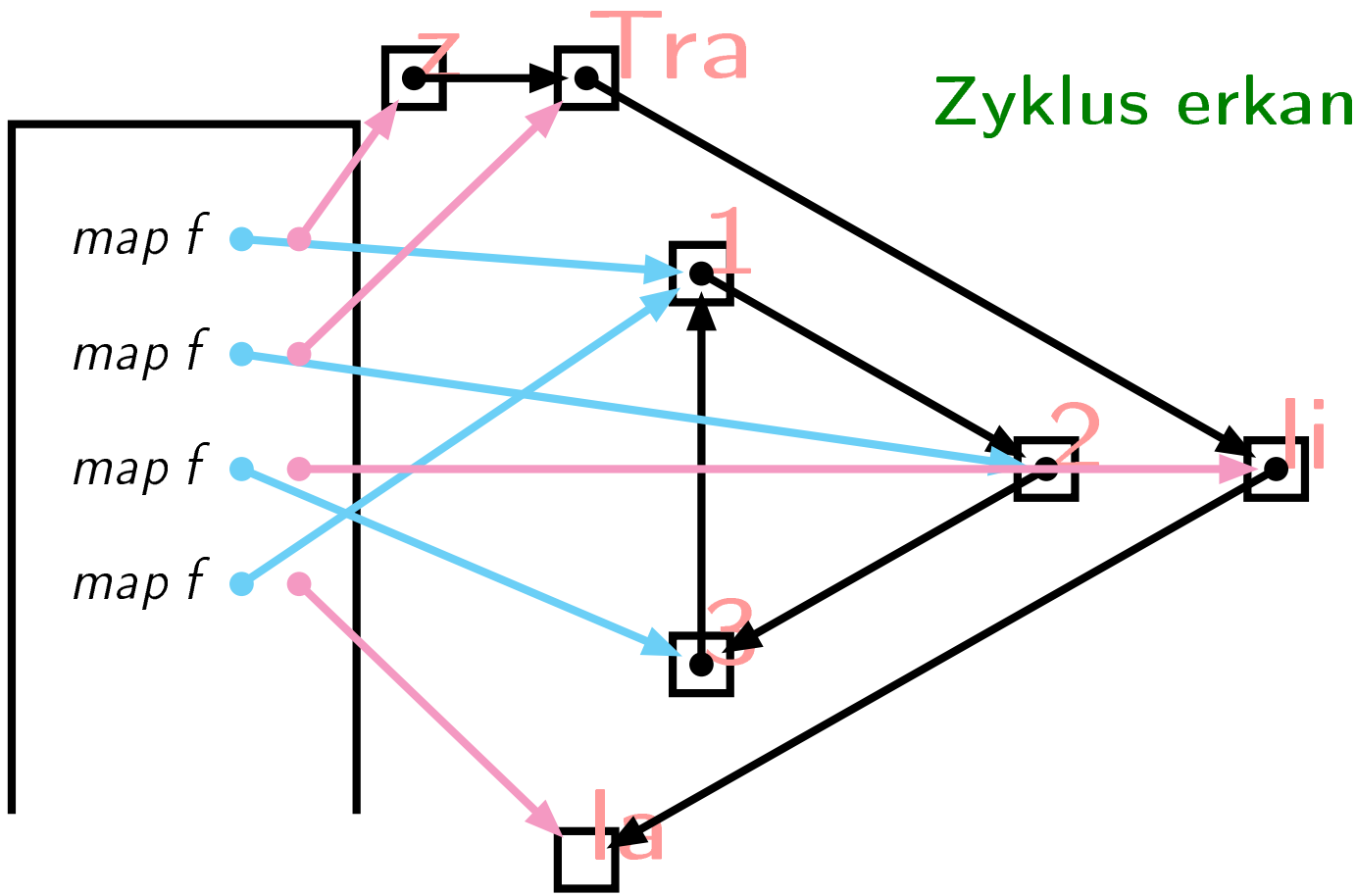
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

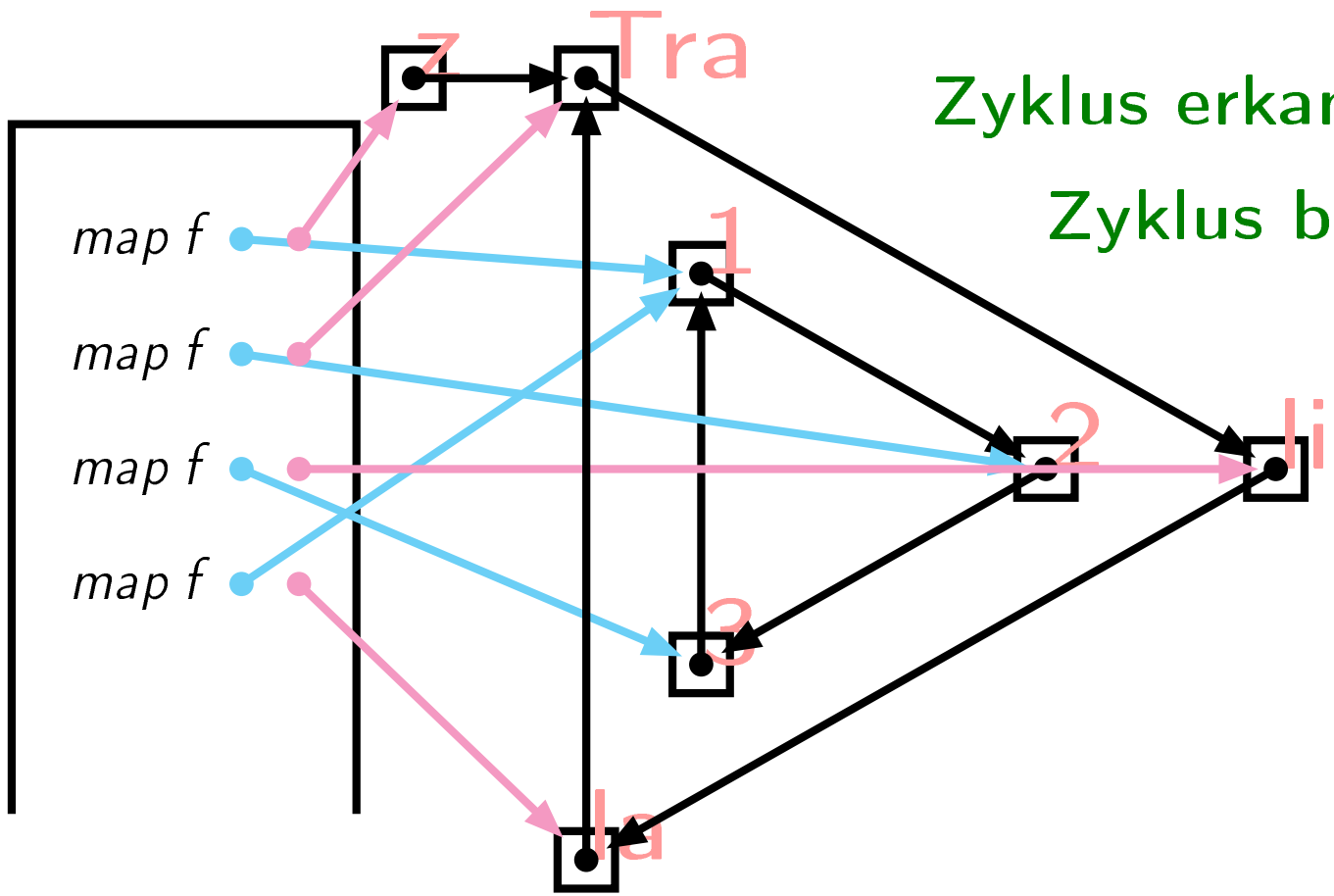
Strikt, Ausgabe per Referenz



$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Erster Lösungsansatz

Strikt, Ausgabe per Referenz



Zyklus erkannt!

Zyklus behandelt!

$$map\ f\ (x : ys) = f\ x : map\ f\ ys$$

Bewertung des Lösungsansatzes

Erzeugung zyklischer Datenstrukturen

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

Traversierung 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

Erzeugung zyklischer Datenstrukturen

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

Traversierung 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

Übersicht

Daten und Signaturen
Prinzip

Generische Handhabung rekursiver Datenstrukturen

- Trennung von lokaler Struktur und Rekursion

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

Rekursion universell durch Fixpunktoperator

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

Daten und Signaturen
Beispiel

Beispiel: Listen

$$\begin{aligned}
&\textbf{data } \mathit{Cell}\ \alpha\ \beta = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ \beta \\
&\textbf{instance } \mathit{Functor}\ (\mathit{Cell}\ \alpha)\ \textbf{where} \\
&\quad \mathit{fmap}\ f\ \mathit{Nil} = \mathit{Nil} \\
&\quad \mathit{fmap}\ f\ (\mathit{Cons}\ x\ y) = \mathit{Cons}\ x\ (f\ y) \\
&\textbf{type } \mathit{List}\ \alpha = \mathit{Fix}\ (\mathit{Cell}\ \alpha)
\end{aligned}$$

$$\begin{aligned}
[] &\simeq \mathit{In}\ \mathit{Nil} \\
[1] &\simeq \mathit{In}\ \$\ \mathit{Cons}\ 1\ \$\ \mathit{In}\ \mathit{Nil} \\
[1,2] &\simeq \mathit{In}\ \$\ \mathit{Cons}\ 1\ \$\ \mathit{In}\ \$\ \mathit{Cons}\ 2\ \$\ \mathit{In}\ \mathit{Nil}
\end{aligned}$$

Daten und Signaturen
Beispiel

Beispiel: Listen

$$\begin{aligned}
&\textbf{data } \mathit{Cell}\ \alpha\ \beta = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ \beta \\
&\textbf{instance } \mathit{Functor}\ (\mathit{Cell}\ \alpha)\ \textbf{where} \\
&\quad \mathit{fmap}\ f\ \mathit{Nil} = \mathit{Nil} \\
&\quad \mathit{fmap}\ f\ (\mathit{Cons}\ x\ y) = \mathit{Cons}\ x\ (f\ y) \\
&\textbf{type } \mathit{List}\ \alpha = \mathit{Fix}\ (\mathit{Cell}\ \alpha)
\end{aligned}$$

$$\begin{aligned}
[] &\simeq \mathit{In}\ \mathit{Nil} \\
[1] &\simeq \mathit{In}\ \$\ \mathit{Cons}\ 1\ \$\ \mathit{In}\ \mathit{Nil} \\
[1,2] &\simeq \mathit{In}\ \$\ \mathit{Cons}\ 1\ \$\ \mathit{In}\ \$\ \mathit{Cons}\ 2\ \$\ \mathit{In}\ \mathit{Nil}
\end{aligned}$$

Signaturen
Intuition

Funktoren für Datenstrukturen

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

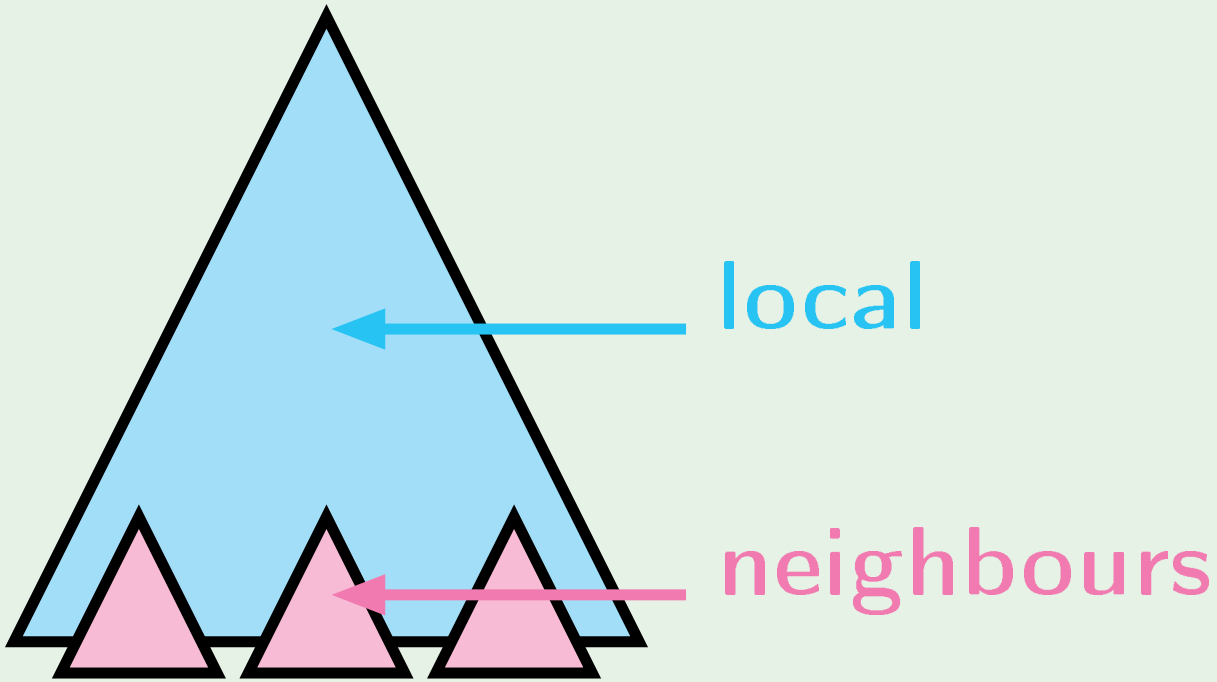
Intuition: Graphstruktur

$$\begin{aligned} \mathit{local} &:: (\mathit{Signature}\ \sigma) \Rightarrow \sigma\ \alpha \rightarrow \sigma\ () \\ \mathit{neighbours} &:: (\mathit{Signature}\ \sigma) \Rightarrow \sigma\ \alpha \rightarrow [\alpha] \end{aligned}$$

$$x \simeq (\mathit{local}\ x, \mathit{neighbours}\ x)$$

Signaturen
Definition

Visualisierung



Signaturen
Definition

Definition

$$\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ \sigma) \Rightarrow \mathit{Signature}\ \sigma\ \textbf{where}\\
&\quad (==^*) \quad :: (\mathit{Eq}\ \alpha) \Rightarrow \sigma\ \alpha \to \sigma\ \alpha \to \mathit{Bool}\\
&\quad \mathit{fmapM} \quad :: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \mu\ \beta) \to \sigma\ \alpha \to \mu\ (\sigma\ \beta)\\
&\quad \mathit{ffold} \quad :: (\alpha \to \beta \to \alpha) \to \alpha \to \sigma\ \beta \to \alpha\\
&\quad \mathit{fzipWithM_} :: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \beta \to \mu\ ()) \to \sigma\ \alpha \to \sigma\ \beta \to \mu\ ()
\end{aligned}$$

Abgeleitete Operationen

$$\begin{aligned}
\mathit{local} &= \mathit{fmap}\ (\mathit{const}\ ())\\
\mathit{neighbours} &= \mathit{ffold}\ (\lambda\ \mathit{xs}\ y \to \mathit{xs} +\!\!+ [y])\ []
\end{aligned}$$

Signaturen
Definition

Definition

$$\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ \sigma) \Rightarrow \mathit{Signature}\ \sigma\ \textbf{where}\\
&\quad (==^{*}) \quad :: (\mathit{Eq}\ \alpha) \Rightarrow \sigma\ \alpha \to \sigma\ \alpha \to \mathit{Bool}\\
&\quad \mathit{fmapM} \quad :: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \mu\ \beta) \to \sigma\ \alpha \to \mu\ (\sigma\ \beta)\\
&\quad \mathit{ffold} \quad :: (\alpha \to \beta \to \alpha) \to \alpha \to \sigma\ \beta \to \alpha\\
&\quad \mathit{fzipWithM_} :: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \beta \to \mu\ ()) \to \sigma\ \alpha \to \sigma\ \beta \to \mu\ ()
\end{aligned}$$

Abgeleitete Operationen

$$\begin{aligned}
\mathit{local} &= \mathit{fmap}\ (\mathit{const}\ ())\\
\mathit{neighbours} &= \mathit{ffold}\ (\lambda\ \mathit{xs}\ y \to \mathit{xs} +\!\!+ [y])\ []
\end{aligned}$$

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**

$$
\begin{aligned}
&\mathit{Identity}\ x ==^{*} \mathit{Identity}\ y && = x == y \\
&\mathit{fmapM}\ f\ (\mathit{Identity}\ x) && = \mathit{liftM}\ \mathit{Identity}\ (f\ x) \\
&\mathit{ffold}\ f\ e\ (\mathit{Identity}\ x) && = f\ e\ x \\
&\mathit{fzipWithM}\_\ f\ (\mathit{Identity}\ x)\ (\mathit{Identity}\ y) && = f\ x\ y
\end{aligned}
$$

Signaturen
Konstruktoren

Konstruktor: Konstanten

$$
\begin{aligned}
&\textbf{newtype } \mathit{Const}\ \alpha\ \beta = \mathit{Const}\ \alpha \\
&\textbf{instance } \mathit{Functor}\ (\mathit{Const}\ \alpha)\ \textbf{where} \\
&\quad \mathit{fmap}\ f = \mathit{id} \\
&\textbf{instance } (\mathit{Eq}\ \alpha) \Rightarrow \mathit{Signature}\ (\mathit{Const}\ \alpha)\ \textbf{where} \\
&\quad \mathit{Const}\ c ==^{*} \mathit{Const}\ d = c == d \\
&\quad \mathit{fmapM}\ f = \mathit{return} \\
&\quad \mathit{ffold}\ f = \mathit{const} \\
&\quad \mathit{fzipWithM}\ \_\ f\ (\mathit{Const}\ c)\ (\mathit{Const}\ d) \mid c == d = \mathit{return}\ ()
\end{aligned}
$$

Signaturen
Konstruktoren

Konstruktor: Punktweises Produkt

$$
\begin{aligned}
&\textbf{data}\ (\otimes)\ \sigma\ \tau\ \alpha = (\otimes)\ (\sigma\ \alpha)\ (\tau\ \alpha)\\
&\textbf{instance}\ (\mathit{Functor}\ \sigma, \mathit{Functor}\ \tau) \Rightarrow \mathit{Functor}\ (\sigma \otimes \tau)\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (x \otimes y) = \mathit{fmap}\ f\ x \otimes \mathit{fmap}\ f\ y\\
&\textbf{instance}\ (\mathit{Signature}\ \sigma, \mathit{Signature}\ \tau) \Rightarrow \mathit{Signature}\ (\sigma \otimes \tau)\ \textbf{where}\\
&\quad (w \otimes x) ==^{*} (y \otimes z) = w ==^{*} y\ \&\&\ x ==^{*} z\\
&\quad \mathit{fmapM}\ f\ (x \otimes y) =\\
&\qquad \mathit{liftM2}\ (\otimes)\ (\mathit{fmapM}\ f\ x)\ (\mathit{fmapM}\ f\ y)\\
&\quad \mathit{ffold}\ f\ e\ (x \otimes y) = \mathit{ffold}\ f\ (\mathit{ffold}\ f\ e\ x)\ y\\
&\quad \mathit{fzipWithM\_}\ f\ (w \otimes x)\ (y \otimes z) =\\
&\qquad \mathit{fzipWithM\_}\ f\ w\ y \gg \mathit{fzipWithM\_}\ f\ x\ z
\end{aligned}
$$

Signaturen
Konstruktoren

Konstruktor: Punktweises Coprodukt

$$\textbf{data}\ (\oplus)\ \sigma\ \tau\ \alpha = Lt\ (\sigma\ \alpha) \mid Rt\ (\tau\ \alpha)$$
$$\textbf{instance}\ (Functor\ \sigma, Functor\ \tau) \Rightarrow Functor\ (\sigma \oplus \tau)\ \textbf{where}$$
$$\quad fmap\ f\ (Lt\ x) = Lt\ (fmap\ f\ x)$$
$$\quad fmap\ f\ (Rt\ y) = Rt\ (fmap\ f\ y)$$
$$\textbf{instance}\ (Signature\ \sigma, Signature\ \tau) \Rightarrow Signature\ (\sigma \oplus \tau)\ \textbf{where}$$
$$\quad \vdots$$
$$\quad fzipWithM\_\ f\ (Lt\ x)\ (Lt\ y) = fzipWithM\_\ f\ x\ y$$
$$\quad fzipWithM\_\ f\ (Rt\ x)\ (Rt\ y) = fzipWithM\_\ f\ x\ y$$

Signaturen
Konstruktoren

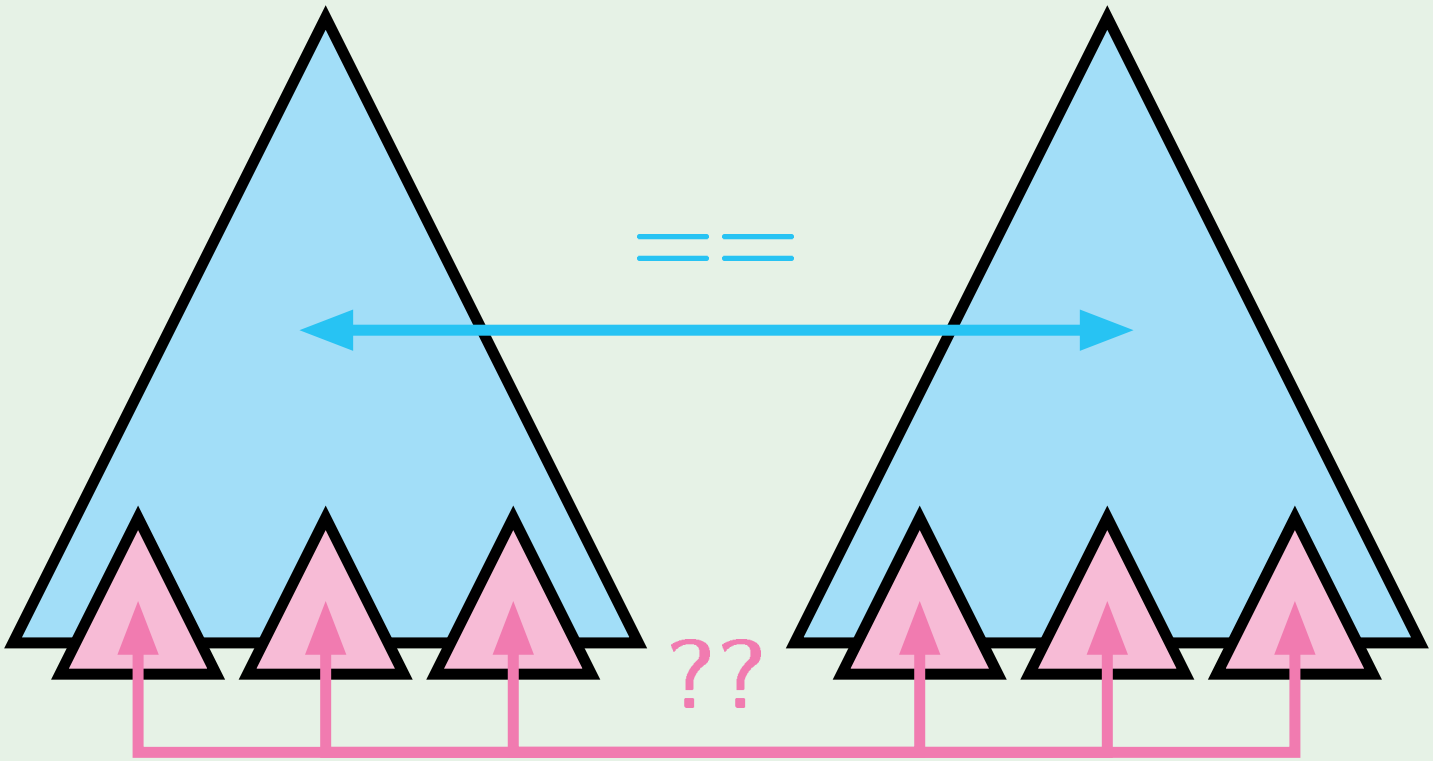
Beispiel: Listen (Wiederholung)

$$\textbf{type } \mathit{Cell}\ \alpha = \mathit{Const}\ () \oplus (\mathit{Const}\ \alpha \otimes \mathit{Identity})$$
$$\textbf{type } \mathit{List}\ \alpha = \mathit{Fix}\ (\mathit{Cell}\ \alpha)$$

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

Signaturen
Relationen auf Daten

Individualtransformationen



Daten

Individualtransformationen

$$\textbf{type}\ \mathit{ITrans}\ \alpha\ \beta = [(\alpha, \beta)]$$

$$\begin{aligned}
&\mathit{trans} :: (\mathit{Signature}\ \sigma) \Rightarrow \sigma\ \alpha \to \sigma\ \beta \to \mathit{ITrans}\ \alpha\ \beta \\
&\mathit{trans}\ x\ y = \mathit{execState}\ (\mathit{fzipWithM_}\ \mathit{upd}\ x\ y)\ [] \\
&\quad \textbf{where} \\
&\qquad \mathit{upd}\ v\ w = \mathit{modify}\ (+\!\!+ [(v, w)]) \\
&\mathit{cover} :: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \beta \to \mu\ ()) \to \mathit{ITrans}\ \alpha\ \beta \to \mu\ () \\
&\mathit{cover} = \mathit{mapM_} \circ \mathit{uncurry}
\end{aligned}$$

Daten

Individualtransformationen

$$\textbf{type}\ \mathit{ITrans}\ \alpha\ \beta = [(\alpha, \beta)]$$

$$\begin{aligned}
&\mathit{trans} &&:: (\mathit{Signature}\ \sigma) \Rightarrow \sigma\ \alpha \to \sigma\ \beta \to \mathit{ITrans}\ \alpha\ \beta \\
&\mathit{trans}\ x\ y &&= \mathit{execState}\ (\mathit{fzipWithM_}\ \mathit{upd}\ x\ y)\ [] \\
&\quad \textbf{where} \\
&\qquad \mathit{upd}\ v\ w &&= \mathit{modify}\ (+\!\!+\ [(v, w)]) \\
&\mathit{cover} &&:: (\mathit{Monad}\ \mu) \Rightarrow (\alpha \to \beta \to \mu\ ()) \to \mathit{ITrans}\ \alpha\ \beta \to \mu\ () \\
&\mathit{cover} &&= \mathit{mapM_} \circ \mathit{uncurry}
\end{aligned}$$

Übersicht

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

$$
\begin{aligned}
&\textbf{newtype } \mathit{Cell} = \mathit{Cell}\ \mathit{Int}\\
&\textbf{newtype } \mathit{Field} = \mathit{Cell}\ \mathit{Int}\\
&\textbf{data } (\mathit{Signature}\ \sigma) \Rightarrow \mathit{Heap}\ \sigma =\\
&\quad \mathit{Heap}\ \{\mathit{nextCell} \;\; :: \mathit{Cell},\\
&\qquad\quad \mathit{nextField} :: \mathit{Field},\\
&\qquad\quad \mathit{cells} \;\;\;\; :: \mathit{Map}\ \mathit{Cell}\ (\sigma\ \mathit{Field}),\\
&\qquad\quad \mathit{fields} \;\;\;\; :: \mathit{Map}\ \mathit{Field}\ \mathit{Cell}\}
\end{aligned}
$$

Speicherzustand
Operationen

Elementarer Speicherzugriff als monadische Operationen

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

Speicherzustand
Operationen

Zentrale Operation 1: Erzeugen von Daten

- Kontextfrei

$$
\begin{aligned}
&\mathit{newCell} \quad :: (\mathit{Signature}\ \sigma) \Rightarrow \sigma\ \alpha \rightarrow \mathit{HeapOp}\ \sigma\ (\mathit{Cell}, \mathit{ITrans}\ \mathit{Field}\ \alpha)\\
&\mathit{newCell}\ s = \mathbf{do}\\
&\qquad\qquad x \leftarrow \mathit{freshCell}\\
&\qquad\qquad t \leftarrow \mathit{fmapM}\ (\mathit{const}\ \mathit{freshField})\ s\\
&\qquad\qquad \mathit{setCell}\ x\ t\\
&\qquad\qquad \mathit{return}\ (x, \mathit{trans}\ t\ s)
\end{aligned}
$$

Speicherzustand
Operationen

Zentrale Operation 2: Speichern von Daten

- Im Kontext einer Ausgabevariable (Aufrufkonvention)

$$
\begin{aligned}
&\mathit{gen} \quad :: (\mathit{Signature}\ \sigma) \Rightarrow \mathit{Field} \rightarrow \sigma\ \alpha \rightarrow \mathit{HeapOp}\ \sigma\ (\mathit{ITrans}\ \mathit{Field}\ \alpha)\\
&\mathit{gen}\ v\ s = \mathbf{do}\\
&\qquad\qquad (y, fs) \leftarrow \mathit{newCell}\ s\\
&\qquad\qquad \mathit{setField}\ v\ y\\
&\qquad\qquad \mathit{return}\ fs
\end{aligned}
$$

Übersicht

Gerüst einer Funktion
Interpretation

Interpretation

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

$$
\begin{aligned}
map_0\ (f, [])\ &= [] \\
map_0\ (f, x : ys)\ &= f\ x : \underbrace{(f, ys)}_{\hookrightarrow} \\
map\ &= [\![(map_0)]\!]
\end{aligned}
$$

- Monadisch über Speicherzustand, Aufrufkonvention

$$
\begin{aligned}
&\textbf{type}\ Interp\ \sigma\ \tau\ &&= \tau\ Cell \to HeapOp\ \sigma\ (\sigma\ (\tau\ Cell)) \\
&\textbf{type}\ Function\ \sigma\ \tau &&= Field \to \tau\ Cell \to HeapOp\ \sigma\ ()
\end{aligned}
$$

Funktionsberechnung
Strategie

Funktionsberechnung

```
eval :: (Signature σ, Signature τ) ⇒ Interp σ τ → Function σ τ
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, _) → getField u >>= setField v
         Nothing     → i x >>= gen v >>= cover recur'
```

Funktionsberechnung
Strategie

Funktionsberechnung mit Stack

$$
\begin{aligned}
&\mathit{eval} \;\; :: (\mathit{Signature}\,\sigma, \mathit{Signature}\,\tau) \Rightarrow \mathit{Interp}\,\sigma\,\tau \rightarrow \mathit{Function}\,\sigma\,\tau \\
&\mathit{eval}\; i = \mathit{recur}\; [] \\
&\quad \textbf{where} \\
&\qquad \mathit{recur}\; q\; v\; x = \\
&\qquad \textbf{let}\; \mathit{recur}' \qquad = \mathit{recur}\; ((v, x) : q) \\
&\qquad\quad \mathit{cycle}\; (\_, y) = x ==^{*} y \\
&\qquad \textbf{in}\; \text{case}\; \mathit{find}\; \mathit{cycle}\; q\; \text{of} \\
&\qquad\quad \mathit{Just}\; (u, \_) \rightarrow \mathit{getField}\; u \gg\!= \mathit{setField}\; v \\
&\qquad\quad \mathit{Nothing} \quad \rightarrow \; i\; x \gg\!= \mathit{gen}\; v \gg\!= \mathit{cover}\; \mathit{recur}'
\end{aligned}
$$

Funktionsberechnung
Strategie

Funktionsberechnung mit Zyklenerkennung

$$
\begin{aligned}
&\mathit{eval} \;\; :: (\mathit{Signature}\,\sigma, \mathit{Signature}\,\tau) \Rightarrow \mathit{Interp}\,\sigma\,\tau \rightarrow \mathit{Function}\,\sigma\,\tau \\
&\mathit{eval}\; i = \mathit{recur}\; [] \\
&\quad \textbf{where} \\
&\qquad \mathit{recur}\; q\; v\; x = \\
&\qquad \textbf{let}\; \mathit{recur}' \qquad\quad = \mathit{recur}\; ((v, x) : q) \\
&\qquad\quad \mathit{cycle}\; (\_, y) = x ==^{*} y \\
&\qquad \textbf{in case}\; \mathit{find}\; \mathit{cycle}\; q\; \textbf{of} \\
&\qquad\quad \mathit{Just}\; (u, \_) \rightarrow \mathit{getField}\; u \gg\!= \mathit{setField}\; v \\
&\qquad\quad \mathit{Nothing} \quad\; \rightarrow\; i\; x \gg\!= \mathit{gen}\; v \gg\!= \mathit{cover}\; \mathit{recur}'
\end{aligned}
$$

Funktionsberechnung
Strategie

Funktionsberechnung mit Zyklenerkennung & -behandlung

$$
\begin{aligned}
&\mathit{eval} \;\; :: (\mathit{Signature}\,\sigma, \mathit{Signature}\,\tau) \Rightarrow \mathit{Interp}\,\sigma\,\tau \rightarrow \mathit{Function}\,\sigma\,\tau \\
&\mathit{eval}\; i = \mathit{recur}\; [] \\
&\quad \textbf{where} \\
&\qquad \mathit{recur}\; q\; v\; x = \\
&\qquad \textbf{let}\; \mathit{recur}' \qquad = \mathit{recur}\; ((v, x) : q) \\
&\qquad\quad \mathit{cycle}\; (\_, y) = x ==^{*} y \\
&\qquad \textbf{in case}\; \mathit{find}\; \mathit{cycle}\; q\; \textbf{of} \\
&\qquad\quad \mathit{Just}\; (u, \_) \rightarrow \mathit{getField}\; u \gg\!= \mathit{setField}\; v \\
&\qquad\quad \mathit{Nothing} \quad \rightarrow\; i\; x \gg\!= \mathit{gen}\; v \gg\!= \mathit{cover}\; \mathit{recur}'
\end{aligned}
$$

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 $[\![i]\!](x)$ (Anamorphismus)
- Beweis durch Graphfärbung & Coinduktion

Funktionsberechnung
Korrektheit

Beispiel: Listen (Demonstration)

$$
\begin{aligned}
test = \ & \mathbf{do} \\
& v \leftarrow freshField \\
& x \leftarrow createWaltz \\
& eval\ (map_0\ song)\ v\ (Identity\ x) \\
& getField\ v
\end{aligned}
$$

Traversierung
Überblick

Traversierung, Suchprobleme

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

Gerüst einer Traversierung
Kalkül

Kalkül

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

$$\textbf{type } \mathit{Calc}\ \alpha = \alpha \to [[\alpha]]$$
$$\mathit{dnf} :: (\alpha \to \mathit{Bool}) \to [[\alpha]] \to \mathit{Bool}$$
$$\mathit{dnf} = \mathit{any} \circ \mathit{all}$$

Gerüst einer Traversierung
Kalkül

Beispiel: Listen (Endlichkeit)

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

Kleinster Fixpunkt für *fin*, größter für *inf*

$$\begin{aligned} k\,(Fin\,[]) &= [[]] & e\,(Fin\,\_) &= False \\ k\,(Fin\,(x : ys)) &= [[Fin\;ys]] & & \\ k\,(Inf\;[]) &= [] & e\,(Inf\;\_) &= True \\ k\,(Inf\;(x : ys)) &= [[Inf\;ys]] & & \end{aligned}$$

Traversierung

```
search      :: (Eq α) ⇒ Calc α → (α → Bool) → α → Bool
search k e = recur []
  where
    recur q x =
    let recur' = recur (x : q)
    in case any (== x) q of
         True  → e x
         False → dnf recur' $ k x
```

Traversierung mit Stack

```
search      :: (Eq α) ⇒ Calc α → (α → Bool) → α → Bool
search k e = recur []
  where
    recur q x =
    let recur' = recur (x : q)
    in case any (== x) q of
         True  → e x
         False → dnf recur' $ k x
```

Traversierung mit Zyklenerkennung

```
search      :: (Eq α) ⇒ Calc α → (α → Bool) → α → Bool
search k e = recur []
  where
    recur q x =
    let recur' = recur (x : q)
    in case any (== x) q of
         True  → e x
         False → dnf recur' $ k x
```

Traversierung mit Zyklenerkennung & -behandlung

```
search      :: (Eq α) ⇒ Calc α → (α → Bool) → α → Bool
search k e = recur []
  where
    recur q x =
    let recur' = recur (x : q)
    in case any (== x) q of
         True  → e x
         False → dnf recur' $ k x
```

Übersicht

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

$$\begin{aligned}\textbf{data}\ \mathit{Univ}\ \alpha = {} & \mathit{Tuple}\ [\alpha] \\ \mid {} & \mathit{Tag}\ \mathit{String}\ \alpha \\ \mid {} & \mathit{Closure}\ \mathit{Function}\ [\alpha]\end{aligned}$$

- Erweiterung des Speichers um Stack und Konstantenpool

$$\textbf{data}\ \mathit{Memory} = \mathit{Memory}\ (\mathit{Heap}\ \mathit{Univ})\ \mathit{Stack}\ \mathit{CPool}$$

Virtuelle Maschine

Modellierung (1)

- Universelle Datensignatur

  data *Univ* α = *Tuple* [α]
  | *Tag String* α
  | *Closure Function* [α]

- 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* | ...

- 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* $| \ldots$

- Ausführungszustand als Monade

  data *Progress* $\alpha =$ *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* → *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

FIXME