Heutiges Thema: Nachlese & -träge

- Performanzsteigerung auf Kosten höheren Speicherverbrauchs

  Nachlese zu...

  - Memoration und
  - dynamischer Programmierung

- Wohldefiniertheits- und Korrektheitsfragen

  Nachträge zum...

  - 3. Vorlesungsteil vom 26.04.2005

Performanzsteigerung auf Kosten höheren Speicherverbrauchs 1(2)

Beobachtung:

- Viele Probleme sind so strukturiert, dass sich die Lösung des Gesamtproblems auf die Lösung vieler, großteils *identischer* Teilprobleme zurückführen lässt.

  Kanonisches Bsp.: Die Berechnung der *Fibonacci*-Zahlen

Probleme des obigen Typs...

- lassen sich oft dramatisch effizienter lösen, indem die Lösung von Teilproblemen gespeichert und wiederverwendet anstatt neu berechnet wird.

Performanzsteigerung auf Kosten höheren Speicherverbrauchs 2(2)

Dieses Prinzip, d.h. ...

- *Wiederverwendung statt Wiederberechnung*

...lässt sich in der Praxis unterschiedlich realisieren!

Memoration vs. Dynamisches Programmieren

Zwei Spielarten sind dabei besonders bedeutungsvoll:

- *"Opportunistische"* Variante: Beginne unmittelbar mit der Lösung des Gesamtproblems und berechne und speichere Lösungen von Teilproblem dann, wenn sie dabei (erstmals) gebraucht werden.

  $\leadsto$ bekannt als *Memoration* (engl. *Memoization*)

- *Systematische* Variante: Berechne und speichere zunächst die Lösungen aller möglichen Teilprobleme und beginne erst dann mit der Lösung des Gesamtproblems.

  $\leadsto$ bekannt als *dynamische Programmierung* (engl. *dynamic Programming*)

Memoration 1(3)

(nach Field/Harrison, S. 447f.)

Die Idee...

- Jede Funktion wird ersetzt durch eine Memo-Variante, die in üblicher Weise den Wert der Funktion berechnet, zusätzlich aber in einer Tabelle, der sog. *Memo-Tabelle*, das entsprechende Argument/Wert-Paar speichert, um so im Bedarfsfall Wiederberechnung durch Wiederverwendung ersetzen zu können.

  Grundlage: Referentielle Transparenz
  $\leadsto$ somit *Seiteneffektfreiheit*!

  Beachte: ...Suggestion zustandsbehafteter imperativer Implementierungsumgebung mit destruktiver Zuweisungsoperation.

Memoration 2(3)

Schematische Umsetzung...

Ersetze

```
f <arg-list> = ...
```

durch

```
memo_f <arg-list> =
      if in_memo_table (<arg-list>)
         then lookup (<arg-list>)
         else insert_into_memo_table (f (<arg-list>))
```

Hinweis: ...beachte die imperative Natur von `insert_into_memo_table`!

Memoration 3(3)

Die Hauptprobleme einer Realisierung...

- Explosiver Speicherverbrauch der Memo-Tabelle

  Dadurch induzierte Teil- und Folgeprobleme...

  - Verlängerte Zugriffszeiten (die den Vorteil der vermiedenen Wiederberechnung (mehr als) aufzehren können).
  - Management der Memo-Tabelle
    - Erkennen nicht mehr benötigter und daher löschbarer Einträge
    - Erkennen am ehesten ersetzbarer Einträge bei volllaufender Tabelle fester Größe

Volle Memoration vs. "Lazy" Memoration 1(4)

Erinnerung:

Wiederverwendung statt Wiederberechnung, falls...

- Funktion angewendet wird auf *gleiches* Argument
  $\leadsto$ *volle Memoration* (engl. (*full memoization*))

- Funktion angewendet wird auf *identisches* Argument
  $\leadsto$ *"lazy" Memoration* (engl. (*lazy memoization*))

Volle Memoration vs. “Lazy” Memoration 2(4)

Lazy Memoration...

- *Ziel*: Memoration auch für Funktionen auf komplexen Datenstrukturen effizient realisieren zu können
- *Trick*: Test auf Gleichheit von Werten komplexer Datenstrukturen auf “Speicherung an gleicher Stelle”, also auf Zeigergleichheit zurückführen
- *Erforderlich*: Zeigergleichheit muss als primitive Operation zur Verfügung stehen, ebenso Primitive für Zugriff und Verwaltung der Memo-Tabelle (Vorschlag somit nicht direkt in Haskell umsetzbar)

Volle Memoration vs. “Lazy” Memoration 3(4)

Verlust der Ausdruckskraft? Nein, denn...

Volle Memoration ist...

- durch lazy Memoration modellierbar
  $\leadsto$ kein Verlust der Ausdruckskraft durch Übergang von voller zu lazy Memoration
- *Implementierungsidee*: Rückführung des Tests auf strukturelle Gleichheit auf Zeigergleichheit durch Übergang zu *eindeutigen* Strukturen

Volle Memoration vs. “Lazy” Memoration 4(4)

Grundlage: Sog. hashing-Konstruktoren.

(Vorschlag geht zurück auf J. M. Spitzen, K. N. Levitt, 1978.)

- *Idee*: Ersetze jeden (Daten-) Konstruktor durch einen zugehörigen Hashing-Konstruktor, der angewendet auf Argumente nur dann eine neue Speicherzelle für den zusammengesetzten Wert zuweist, wenn nicht schon eine für identische Argumente existiert.
- *Vor- und Nachteile*:
  - Volle Memoration mittels lazy Memoration ausdrückbar
  - Aber: Alle praktischen Probleme voller Memoration wieder präsent, insbesondere aufwändiger Eindeutigkeits- (Gleichheits-) Test, schwer kontrollierbares Wachstum der Memo-Tabelle, spezielle Sprachprimitive (etwa zum Zeigervergleich) erforderlich.

Memoration vs. Dynamisches Programmieren – Ein relativer Vergleich 1(2)

Relative Vorteile von...

- Memoration über dynamisches Programmieren
  - Nur die Lösung solcher Teilprobleme wird berechnet und gespeichert, die zur Lösung des Gesamtproblems tatsächlich benötigt wird.

    Beachte: ...oft ergibt sich dadurch ein substantieller Performanzunterschied; oft ist aber auch nichtsdestotrotz die Lösung aller Teilprobleme zu berechnen, wie das Beispiel der Berechnung der Fibonacci-Zahlen zeigt.
  - Keine Notwendigkeit einer systematischen Zerlegung des Gesamtproblems in Teilprobleme und deren Vorwegberechnung.

    $\leadsto$ *...intellektuell weniger anstrengend.*

Memoration vs. Dynamisches Programmieren – Ein relativer Vergleich 2(2)

Relative Vorteile von...

- dynamischem Programmieren über Memoration
  - oft Lösungen mit signifikant geringerem Speicherverbrauch möglich

    Stichwort: ...*limited history recurrences* wie im Beispiel der Berechnung der Fibonacci-Zahlen: `fib n = fib(n-1) + fib(n-2)`

    ⇝ ...nur zwei vorangegangene Werte sind erforderlich, um einen neuen Wert zu berechnen. Die Speicherzellen dieser beiden Werte können somit immer wiederverwendet werden.
  - *"Faustregel"*: Lösungen mittels dynamischen Programmierens i.a. um einen konstanten Faktor effizienter als durch Memoration.

Memoration vs. Dynamisches Programmieren am Beispiel 1(10)

...unterschiedlicher Implementierungsvarianten der Fibonacci-Funktion.

Die klassische, ebenso naheliegende wie ineffiziente Variante:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1)  +  fib (n-2)
```

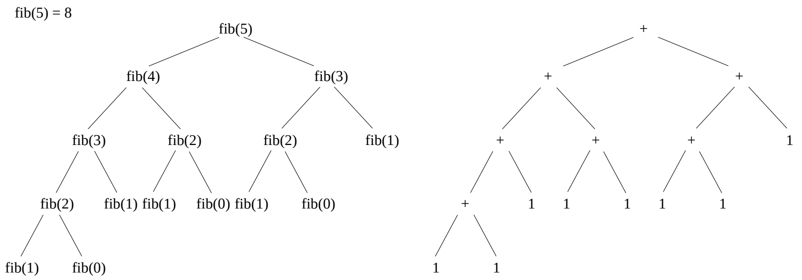

Memoration vs. Dynamisches Programmieren am Beispiel 2(10)

Illustration...

fib 0 = 1	fib 10 = 89	fib 20 = 10.946
fib 1 = 1	fib 11 = 144	fib 21 = 17.711
fib 2 = 2	fib 12 = 233	fib 22 = 28.657
fib 3 = 3	fib 13 = 377	fib 23 = 46.368
fib 4 = 5	fib 14 = 610	fib 24 = 75.025
fib 5 = 8	fib 15 = 987	fib 25 = 121.393
fib 6 = 13	fib 16 = 1.597	fib 26 = 196.418
fib 7 = 21	fib 17 = 2.584	fib 27 = 317.811
fib 8 = 34	fib 18 = 4.181	fib 28 = 514.229
fib 9 = 55	fib 19 = 6.765	fib 29 = 832.040
		fib 30 = 1.346.269

Beachte: ...das Resultat von fib verdoppelt sich jeweils für 2er-Argumentschritte.

Memoration vs. Dynamisches Programmieren am Beispiel 3(10)



Intuitiv:

- fib n ...gleich Anzahl der Blätter im Baum
- (fib n) - 1 ...gleich Anzahl der Additionen
- (2*fib n) -1 ...gleich Anzahl der Knoten gleich Anzahl der (rekursiven) Aufrufe von fib n

Memoration vs. Dynamisches Programmieren am Beispiel 4(10)

Quintissenz

Die Berechnung von fib n, n≥ 2, erfordert...

- (2*fib n) -1 Aufrufe von fib mit Argumenten zwischen 0 und n und somit zur Berechnung von max. n+1 verschiedenen Werten

Am Beispiel des Aufrufs fib 30 heißt das...

- (2*1.346.269-1) Aufrufe von fib mit nur 31 verschiedenen Argumenten und folglich mit max. nur 31 verschiedenen Resultaten
  - ⇝ *...eine Unmenge doppelter Arbeit!*
  - ⇝ *...viel Potential für Memoration und dynamische Programmierung!*

Memoration vs. Dynamisches Programmieren am Beispiel 5(10)

Unter Ausnutzung der Haskell zugrundeliegenden *lazy evaluation*-Auswertungsstrategie...

- mittels Strömen (Variante 1):

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

fib :: Int -> Integer
fib n = fibs!!n

-- Oder:
-- fib :: Int -> Integer
-- fib n = last (take (n+1) fibs)
```

Ein Fall von Memoration, dynamischem Programmieren, oder...?

Memoration vs. Dynamisches Programmieren am Beispiel 6(10)

Unter Ausnutzung der Haskell zugrundeliegenden *lazy evaluation*-Auswertungsstrategie...

- mittels Strömen (Variante 2):

```
flist = [ fib x | x <- [0 ..] ]
fib 0 = 1
fib 1 = 1
fib n = flist !! (n-1)  +  flist !! (n-2)
```

Ein Fall von Memoration, dynamischem Programmieren, oder...?

Memoration vs. Dynamisches Programmieren am Beispiel 7(10)

Unter Ausnutzung, dass stets nur zwei Vorgängerwerte zur Berechnung benötigt werden...

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fibh 1 2 2 n

fibh :: Integer -> Integer -> Integer -> Integer -> Integer
fibh fibPredm fibm m n
   | m == n    = fibm
   | otherwise = fibh fibm (fibPredm+fibm) (m+1) n
```

Ein Fall von Memoration, dynamischem Programmieren, oder...?

Memoration vs. Dynamisches Programmieren am Beispiel 8(10)

Direkte Berechnung der Fibonacci-Zahlen...

Bezeichne F_n, $n \in \mathbb{N}$, die wie folgt definierte n-te F-Zahl:

$$F_0 = 0; \; F_1 = 1; \text{ für jedes } n \geq 2, \; F_n = F_{n-2} + F_{n-1}$$

Mit diesen Bezeichnungen gilt:

Theorem

$$\forall\, n \in \mathbb{N}.\; F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Memoration vs. Dynamisches Programmieren am Beispiel 9(10)

Wegen

$$(F_i)_{i \in \mathbb{N}} = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

$$(fib_i)_{i \in \mathbb{N}} = 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

gilt somit:

Korollar $\forall\, n \in \mathbb{N}.\; fib(n) = F_{n+1}$

Memoration vs. Dynamisches Programmieren am Beispiel 10(10)

In solchen Situationen...

- Memoration und dynamisches Programmieren unnötig

Aber...

- auf die Existenz einer solchen direkten Charakterisierung muss man erst einmal kommen :-)

Deren Beweis, in diesem Falle...

- mittels *verallgemeinerter Induktion*

ist dann wieder einfacher.

Beweisprinzipien: Vollständige, verallgemeinerte, strukturelle Induktion

Zur Erinnerung: Die Prinzipien der...

- *vollständigen Induktion*

  $$(A(1) \wedge (\forall n \in \mathbb{N}.\, A(n) \Rightarrow A(n+1))) \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, A(n)$$

- *verallgemeinerten Induktion*

  $$(\forall n \in \mathbb{N}.\, (\forall m < n.\, A(m)) \Rightarrow A(n)) \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, A(n)$$

- *strukturellen Induktion*

  $$(\forall\, s \in S. \forall\, s' \in Komp(s).\, A(s')) \Rightarrow A(s)) \quad \Rightarrow \quad \forall\, s \in S.\, A(s)$$

Beweis des Theorems 1(5)

Beweis des Theorems ...mithilfe verallgemeinerter Induktion.

Unter der Induktionshypothese, dass für alle $k < n$ mit $n \in \mathbb{N}$ beliebige natürliche Zahl die Beziehung

$$F_k = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k}{\sqrt{5}}$$

gelte, können wir die Prämisse der dem Prinzip der verallgemeinerten Induktion zugrundeliegen Implikation für alle natürlichen Zahlen n durch folgende Fallunterscheidung beweisen.

Beweis des Theorems 2(5)

Fall 1: $n = 0$. In diesem Fall erhalten wir durch Nachrechnen wie gewünscht:

$$F_0 = 0 = \frac{1-1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Beweis des Theorems 3(5)

Fall 2: $n = 1$. Auch in diesem Fall erhalten wir durch Nachrechnen die gewünschte Gleichheit:

$$F_1 = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Beweis des Theorems 4(5)

Fall 3: $n \geq 2$. Unter Anwendung der Induktionshypothese (IH) für $n-2$ und $n-1$ erhalten wir die gewünschte Gleichheit:

$$\begin{array}{rcl}
 & & F_n \\
\text{(Def. von } F_n\text{)} & = & F_{n-2} + F_{n-1} \\
\text{(IH (zweimal))} & = & \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}} \\
 & = & \frac{\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\right] - \left[\left(\frac{1-\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]}{\sqrt{5}} \\
 & = & \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1+\sqrt{5}}{2}\right] - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1-\sqrt{5}}{2}\right]}{\sqrt{5}} \\
(*) & = & \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} \\
 & = & \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}
\end{array}$$

Beweis des Theorems 5(5)

...wobei die mit (∗) gekennzeichnete Gleichheit durch die beiden folgenden Beziehungen gerechtfertigt ist, die sich mithilfe der Binomischen Formeln einfach zeigen lassen:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1+\frac{1+\sqrt{5}}{2}$$

Auf ähnliche Weise rechnet man nach:

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = 1+\frac{1-\sqrt{5}}{2}$$

□

Zum Abschluss zu Mem. & dyn. Prog.

Rein funktionale Implementierung der Memo-Variante von fib...

```
type Arg_type   = Integer
type Res_type   = Integer

type Memo_table = [(Arg_type,Res_type)]

type Manager = Arg_type -> [Arg_type]

look_up :: (Memo_table,Arg_type) -> (Bool,Res_type)
-- liefert "wahr" und das Resultat, falls die Memo-Tabelle
-- fuer das gegebene Argument einen Eintrag enthaelt.
look_up (tab,arg)
  | tab == []             = (False,(-1))
  | fst (head tab) == arg = (True,snd (head tab))
  | otherwise             = look_up ((tail tab),arg)
```


Fkt. Implement. der Memo-Var. von fib

```
delete :: (Memo_table,[Arg_type]) -> Memo_table
-- loescht, so vorhanden, die zu den gegebenen Argumenten
-- gehoerigen Eintraege in der Memo-Tabelle:
delete (tab,l)
  | l == []   = tab
  | otherwise = delete (delete_entry (tab,head l), tail l)

delete_entry :: (Memo_table,Arg_type) -> Memo_table
delete_entry (tab,arg)
  | tab == []             = tab
  | fst (head tab) == arg = tail tab
  | otherwise             = (head tab) : delete_entry (tail tab, arg)
```


Fkt. Implement. der Memo-Var. von fib

```
insert :: (Memo_table,(Arg_type,Res_type),Manager) -> Memo_table
insert (tab,(arg,res),man) = (arg,res) : delete (tab, man(arg))

fib n = let (finalTab, answer) = mfib ([], n) in answer

mfib (tab, n) =
  if n <= 1
    then (tab,1)
    else let (knowlt,res) = look_up(tab,n) in
               if knowlt
                 then (tab,res)
                 else let (tab1,a1) = mfib(tab,n-1) in
                          let (tab2,a2) = mfib(tab1,n-2) in
                            let theAnswer = a1+a2 in
                              (insert(tab2,(n,theAnswer),\y -> [y-2]),
                                 theAnswer)
```

Literaturhinweise zu Memoration und dynamischer Programmierung 1(2)

In funktionalem Kontext...

- *Memoration*
  - Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988, Kapitel 19.
- *Dynamische Programmierung* (*und Memoration*)
  - Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions*. Brooks/Cole Publishing Company, 1999, Kapitel 12.
- *"Hashing"-Konstruktoren*
  - J. M. Spitzen, K. M. Levitt. *An Example of Hierarchical Design and Proof*. Communications of the ACM 21(12), 1064 - 1075, 1978.

Literaturhinweise zu Memoration und dynamischer Programmierung 2(2)

In nichtfunktionalem Kontext...

- *Dynamische Programmierung*
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. 2nd edition, MIT Press, 2001, Kapitel 15.
  - Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag (Telos), 1997, Kapitel 3.

Korrektheit von Programmen/Beweis von Programmeigenschaften

Induktion vs. Rekursion

- *...eine Liste ist leer oder ein Paar bestehend aus einem Element und einer weiteren Liste*
- *...ein Baum ist leer oder besteht aus einem Knoten und einer Menge von weiteren Bäumen*

Beobachtung

- Definition von Datenstrukturen
  ...häufig induktives Definitionsmuster
- Funktionen auf Datenstrukturen
  ...häufig rekursives Definitionsmuster

Induktive Beweise über (endliche) Listen

Beweismuster... Sei P eine Eigenschaft über Listen...

1. *Induktionsanfang*: ...beweise, dass P für die leere Liste gilt, d.h. beweise $P([])$.
2. *Induktionsschluss*: ...beweise unter Annahme der Gültigkeit von $P(xs)$ (*Induktionshypothese*) die Gültigkeit von $P(x : xs)$.

Allgemeiner

- ...nicht nur für Listen
  induktiver Beweis über die Struktur (*strukturelle Induktion*)

Induktion über endl. Listen / Bsp. 1(2)

Behauptung

$$\forall xs, ys.\ length\ (xs \mathbin{++} ys) = length\ xs\ +\ length\ ys$$

Beweis über den induktiven Aufbau von xs

Induktionsanfang

$$\begin{aligned} & length([] \mathbin{++} ys) \\ = \ & length\ ys \\ = \ & 0\ +\ length\ ys \\ = \ & length\ []\ +\ length\ ys \end{aligned}$$

Induktion über endl. Listen / Bsp. 2(2)

Induktionsschluss

$$\begin{aligned} & length((x:xs) \mathbin{++} ys) \\ = \ & length\ (x:(xs \mathbin{++} ys)) \\ = \ & 1\ +\ length\ (xs \mathbin{++} ys) \\ = \ & 1\ +\ (length\ xs\ +\ length\ ys) \quad \text{(Induktionshypothese)} \\ = \ & (1\ +\ length\ xs)\ +\ length\ ys \\ = \ & length\ (x:xs)\ +\ length\ ys \end{aligned}$$

□

Gleichheit von Funktionen 1(2)

```
listSum :: Num a => [a] -> a
listSum []    = 0
listSum (x:xs) = x + listSum xs
```

Behauptung

$$\forall xs.\ listSum\ xs\ =\ foldr\ (+)\ 0\ xs$$

Beweis über den induktiven Aufbau von xs

Induktionsanfang

$$\begin{aligned} & listSum\ [] \\ = \ & 0 \\ = \ & foldr\ (+)\ 0\ [] \end{aligned}$$

Gleichheit von Funktionen 2(2)

Induktionsschluss

$$\begin{aligned} & listSum\ (x:xs) \\ = \ & x\ +\ listSum\ xs \\ = \ & x\ +\ foldr\ (+)\ 0\ xs \quad \text{(Induktionshypothese)} \\ = \ & foldr\ (+)\ 0\ (x:xs) \end{aligned}$$

□

Eigenschaften von map und fold 1(2)

...weitere Beispiele induktiv beweisbarer Eigenschaften:

```
map (\x -> x) = \x -> x
map (f.g) = map f . map g
map f.tail = tail . map f
map f . reverse = reverse . map f
map f . concat = concat . map (map f)
map f (xs++ys) = map f xs ++ map f ys
```

Ist `f` strikt, gilt zusätzlich

```
f . head = head . map f
```


Eigenschaften von map und fold 2(2)

...induktiv lässt sich ebenfalls zeigen:

```
foldr op e xs = foldl op e xs
```

...für assoziativen Operator `op` mit `e 'op' x = x 'op' e` für alle `x` und endliches `xs`

```
foldr op e xs = foldl (flip op) e (reverse xs)
```

...für alle endlichen `xs`

```
foldr op1 e xs = foldl op2 e xs
```

...falls

```
x 'op1' (y 'op2' z) = (x 'op1' y) 'op2' z   und
x 'op1' e = e 'op2' x
```


Eigenschaften der Listenkonkatenation

...für alle `xs`, `ys` und `zs` gilt:

```
(xs++ys) ++ zs = xs ++ (ys++zs)   (Assoziativitaet von ++)
xs++[] = []++xs    ([] neutrales Element von ++)
```


Eigenschaften von `take` und `drop`

...für alle `m`, `n` mit `m,n` $\geq$ 0 und endlichen `xs` gilt:

```
take n xs ++ drop n xs = xs
take m . take n = take (min m n)
drop m . drop n = drop (m+n)
take m . drop n = drop n . take (m+n)
```

...für `n` $\geq$ `m` gilt zusätzlich

```
drop m . take n = take (n-m) . drop m
```

Eigenschaften von `reverse`

...für alle endlichen `xs` gilt:

```
reverse (reverse xs) = xs
head (reverse xs) = last xs
last (reverse xs) = head xs
```


Endliche Listen vs. Ströme

Eigenschaften auf endlichen Listen

- können...
  z.B. `take n xs ++ drop n xs = xs`
- ...müssen sich aber nicht auf Ströme übertragen
  z.B. `reverse (reverse xs)) = xs`

...neue Beweisstrategie erforderlich.

Zur Intuition

Sukzessive Approximation von Listen

- endliche Situation ...[1,2,3,4]

```
bottom
1 : bottom
1 : 2 : bottom
1 : 2 : 3 : bottom
1 : 2 : 3 : 4 : bottom
1 : 2 : 3 : 4 : []
```

- unendliche Situation ...[1,2,3,4,..

```
bottom
1 : bottom
1 : 2 : bottom
1 : 2 : 3 : bottom
1 : 2 : 3 : 4 : bottom
1 : 2 : 3 : 4 : 5 : bottom
...
```


Sprechweisen

- `bottom` ...*total undefinierte Liste*
- `1 : 2 : 3 : 4 : 5 : .. : bottom` ...*partielle Liste*

Anmerkung

...jeder Haskell-Datentyp enthält den speziellen Wert $\perp$.

Polymorph	Konkret
`bot :: a`	`bot :: Integer`
`bot = bot`	

$\perp$ repräsentiert...

- fehlerhafte oder nicht terminierende Berechnungen
- kann als "kleinste" Approximation von (gewöhnlichen) Elementen des zugehörigen Datentyps aufgefasst werden

Induktive Beweise über Ströme

Beweismuster... Sei P eine Eigenschaft über Strömen

1. *Induktionsanfang*: ...beweise, dass P für die am wenigsten definierte Liste gilt, d.h. beweise $P(\perp)$ (anstelle von $P([])$).
2. *Induktionsschluss*: ...beweise unter Annahme der Gültigkeit von $P(xs)$ (*Induktionshypothese*) die Gültigkeit von $P(x:xs)$.

Induktion über Ströme / Bsp. 1(2)

Behauptung

$$\forall \text{Ströme } xs.\ take\ n\ xs\ {+}{+}\ drop\ n\ xs\ =\ xs$$

Beweis über den induktiven Aufbau von xs

Induktionsanfang

$$\begin{aligned} & take\ n\ \perp {+}{+}\ drop\ n\ \perp \\ = \ & \perp\ {+}{+}\ drop\ n\ \perp \\ = \ & \perp \end{aligned}$$

Induktion über Ströme / Bsp. 2(2)

Induktionsschluss

$$\begin{aligned} & take\ n\ (x:xs)\ {+}{+}\ drop\ n\ (x:xs) \\ = \ & x\ :\ (take\ (n-1)\ xs\ {+}{+}\ drop\ (n-1)\ xs \\ = \ & x\ :\ xs \qquad \text{(Induktionshypothese)} \end{aligned}$$

$\square$

Ergänzende und weiterführende Literatur

- L. C. Paulson. *Logic and Computation – Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
- Simon Thompson. *Proof for Functional Programming*. In K. Hammond, G. Michaelson (Hrsg.), *Research Directions in Parallel Functional Programming*, Springer, 1999.

Vorschau auf die nächsten Vorlesungstermine...

- Di, 24.05.2005, 14 - 16 Uhr, CompLang-Bibliothek
- Di, 07.06.2005, 14 - 16 Uhr, CompLang-Bibliothek