Kapitel 4

Entwurfsmethoden, spezielle Datenstrukturen

4.1 Teile und Herrsche (Divide and Conquer)

Diese Entwurfsmethode für Algorithmen kommt in vielen Bereichen vor und läßt sich folgendermaßen beschreiben:

- Teile das Problem in kleinere Unterprobleme (Divide)
- Löse rekursiv die entstehenden Unterprobleme (Conquer)
- Setze die Lösungen zusammen.

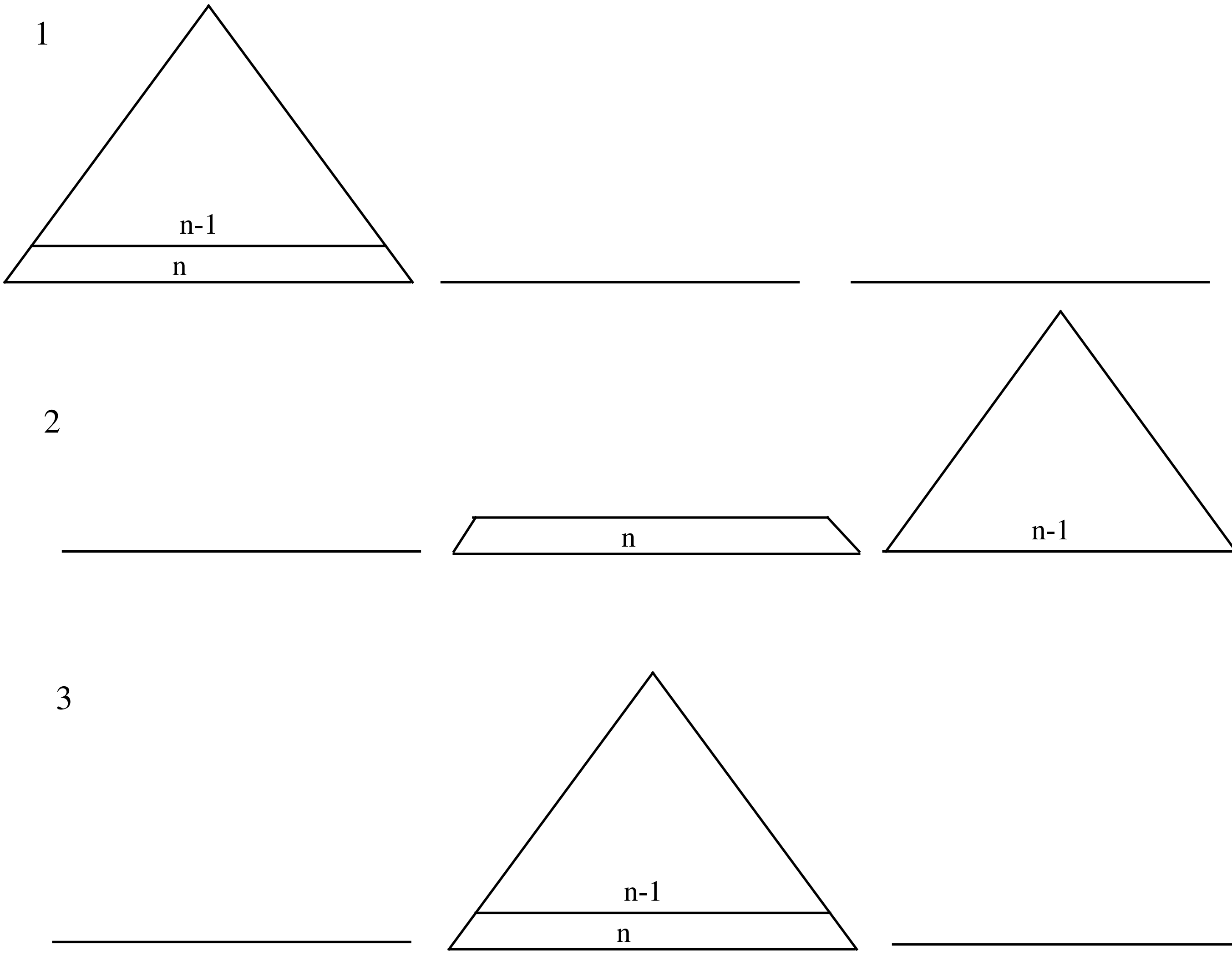
Einige Instanzen haben wir schon kennengelernt:

- Mergesort: Die Liste wurde in zwei gleichlange Unterlisten zerlegt.
- Quicksort. Die Liste wurde in zwei Unterlisten, der kleineren und größeren Elemente zerlegt.
- Intervallhalbierung: Das Intervall wurde in zwei Hälften zerlegt, aber das Zusammensetzen der Lösung war nicht nötig.
- Die schnelle Berechnung ganzzahliger Potenzen.

Kennzeichnend für die Methode Teile-und-Herrsche ist, daß man hiermit oft einen linearen Anteil Laufzeit in einen logarithmischen verbessern kann. Dabei darf der Aufwand des Teilens und Zusammenfügens nicht zu groß sein, da sonst keine Verbesserung eintritt. Ebenso sollte die Gesamtsumme der Größen der Teilprobleme höchstens die Größe des Problems selbst haben. Zum Beispiel ist der Einfügesort $O(n^2)$, während der Mischsort Laufzeit $O(n * \mathtt{log}(n)$ hat.

Beispiel 1. Wir betrachten das Beispiel der "Türme von Hanoi". Gegeben ist ein Stapel von verschieden großen Goldscheiben. Die Aufgabe ist, diesen Stapel auf einen zweiten umzustapeln, wobei man einen weiteren Hilfsstapel verwenden darf, aber nur kleiner auf größeren Scheiben liegen dürfen. Nach der Sage war dieser Stapel 64 Scheiben hoch, und wenn die Aufgabe gelöst ist, dann ist das Ende der Zeit gekommen.

Die Berechnung der notwendigen Umstapelungen läßt sich leicht mit der Teile-und-Herrsche Methode zerlegen:



Man erhält: Die Bewegungen für n, die notwendig sind:

1. Bewegungen, die notwendig sind, um einen $n-1$ Stapel von 1 nach 3 umzustapeln, wobei 2 der Hilfsstapel ist.

2. Bewege die Scheibe n von Stapel 1 nach Stapel 2

3. Bewegungen, die notwendig sind, um den $n-1$ Stapel von 3 nach 2 umzustapeln, wobei 1 der Hilfsstapel ist.

Wenn man die Nr. der Stapel als Variable mitführt, so erhält man einen recht einfachen Algorithmus dafür, wenn man bei der rekursiven Lösung der Teilprobleme die unteren Scheiben ignoriert. Das Ergebnis ist eine Liste der Länge $2^n - 1$, wenn n die Anzahl der Scheiben ist.

```
--  hanoi gibt Zugfolge aus, die zum Ziel f\"uhrt:
--   Scheiben-groesse,  von-Stapel,  zu-Stapel


--  hanoi:  Stapel, Stapelnr, Zielstapelnr Hilfstapelnr
hanoi  xs a b c  = hanoiw  (reverse xs) a b c
hanoiw  [] _ _ _  = []

hanoiw  xa a b c  =
             (hanoiw  (tail xa) a c b)
                                ++
              ((head xa ,(a,b)): (hanoiw   (tail xa) c b a))

--   Testaufruf und Ergebnis:
--  hanoi  [1,2,3] 1 2 3
--  [(1,(1,2)), (2,(1,3)), (1,(2,3)), (3,(1,2)), (1,(3,1)), (2,(3,2)), (1,(1,2))]
```

Übungsaufgabe 1. Wir nehmen an, daß Mengen als Listen implementiert sind. Berechne die Liste aller Teilmengen einer Menge (d.h. die Potenzmenge) mit dem divide-und conquer-Verfahren. Was ist der Zeitbedarf des Algorithmus? Gibt es einen logarithmischen Faktor?

4.2 Suche

4.2.1 Normale Suche mit backtracking

Suchverfahren werden benutzt, wenn man für ein Problem anhand gegebener Parameter eine Lösung finden oder eine (möglichst gute) Lösung konstruieren will. Ein bekanntes Beispiel ist das sogenannte Handlungsreisenden-Problem (travelling salesman problem), das, gegeben eine Landkarte mit zu besuchenden Städten und den Wegen und den Entfernungen, nach einem optimalen Weg fragt, der durch alle Städte geht.

Eine solche Suchaufgabe kann man als Suche nach Wegen in einem ungerichteten Graphen mit durch Zahlen bewerteten Kanten betrachten. Falls Einbahnstraßen zu berücksichtigen sind, kann man auch eine gerichteten Graphen betrachten. Die Suche verfolgt normalerweise eine Strategie, den Suchbaum zu durchlaufen: Tiefensuche (depth-first), Breitensuche (breadth-first) oder auch andere. Die Tiefensuche erfordert normalerweise mit einem Hoch- und wiederabsteigen im Baum. Diesen Vorgang nennt man Zurücksetzen (backtracking).

Die naive Strategie würde alle Wege konstruieren, diese bewerten und dann den besten nehmen. Allerdings hat dieses Verfahren dann exponentielle Laufzeit in Abhängigkeit von der Anzahl der Knoten und Wege. Dies würde somit schon für relative kleine Graphen versagen.

Es gibt viele Probleme, die ähnlich strukturiert sind, z.B. die Suche nach einem optimalen Zug in Spielen, wie Dame, Schach, Tictactoe, Go, oder die

Probleme im Zusammenhang mit logischen Aussagen, wie z.B. die Frage nach einer erfüllenden Belegung für eine aussagenlogische Formel.

Als Beispiel betrachten wir das Problem, aus einem gegebenen Satz von Gewichten ein gegebenes Gewicht zu konstruieren. Vereinfachtes Modell ist eine Multimenge (oder eine Liste) von positiven ganzen Zahlen und es soll eine Untermultimenge gefunden werden, deren Summe genau das gewünschte Gewicht ergibt.

1. Erste Methode: Man konstruiere die Liste aller Unterlisten, berechne deren Summe und vergleiche dann mit dem gewünschten Gewicht:

```
data Maybe a = Nothing | Just a
gewichtx xs ziel =
   let ulisten = unterlisten xs
       wertuliste = zip (map sum ulisten) ulisten
       okliste = filter (\(w,xs) -> w == ziel) wertuliste
    in head okliste

unterlisten [x] = [[x]]
unterlisten (x:xs) =
      let unterl = unterlisten xs
       in ([x]: (map (\ul -> x:ul) unterl)) ++ unterl
```

   Diese Methode ist normalerweise ineffizient, da die Konstruktion aller Unterlisten einen exponentiellen Platzbedarf hat. Durch die Vorgehensweise der lazy Auswertung in Hugs wird allerdings nur das notwendigste ausgewertet, d.h. es werden solange Unterlisten konstruiert, bis die gefilterte Liste ein Element enthält. Danach wird dieses Element ausgegeben und die weitere Konstruktion gestoppt.

   Dies ist bereits eine Suche mit backtracking, allerdings wird eine blinde Suche ohne Rücksicht auf die Werte durchgeführt, d.h., wenn die Unterliste bereits zu viele Gewichte enthält, wird trotzdem weiter versucht, noch welche dazuzunehmen, natürlich ohne Erfolgsaussicht.

2. Eine Suche mit backtracking geht von links nach rechts in der List vor, probiert alle Möglichkeiten aus, und macht backtracking, wenn die Summe der Gewichte zu groß ist.

   Das folgende Programm nutzt diese Einsicht aus, allerdings kann man diese Auswertung so nicht leicht verfolgen, deshalb ist in den Programmen eine Version angegeben, die den Weg ebenfalls verfolgt.

```
--   Suche mit backtracking:
```

```
gewichtxbt xs ziel =
    case gewichtxbtw xs ziel []
       of Nothing -> Nothing
          Just res -> Just (sum res, res)

gewichtxbtw xs 0 res = Just res
gewichtxbtw [] ziel res = Nothing
gewichtxbtw (x:xs) ziel res =
     if ziel < 0 then Nothing
     else
       let versuch1 = gewichtxbtw  xs (ziel-x) (x:res)
           versuch2 = gewichtxbtw  xs ziel res
        in case versuch1 of
             Just x -> Just x
             Nothing -> versuch2
```

Testeingabe = $[5, 5, 3, 3, 3, 3]$, gewünschtes Ergebnis ist 12.

```
(Just (12,[3, 3, 3, 3]),
[[5], [5, 5], [5, 5, 3], [5, 5, 3], [5, 5, 3], [5, 5, 3],
[5, 3], [5, 3, 3], [5, 3, 3, 3], [5, 3, 3, 3], [5, 3, 3],
[5, 3, 3, 3], [5, 3, 3], [5, 3], [5, 3, 3], [5, 3, 3, 3],
[5, 3, 3], [5, 3], [5, 3, 3], [5, 3], [5], [5, 3], [5, 3, 3],
[5, 3, 3, 3], [5, 3, 3, 3], [5, 3, 3], [5, 3, 3, 3],
[5, 3, 3], [5, 3], [5, 3, 3], [5, 3, 3, 3], [5, 3, 3],
[5, 3], [5, 3, 3], [5, 3],
[3], [3, 3], [3, 3, 3], [3, 3, 3, 3]])
```

Man sieht allerdings, daß es auch hier noch Verbesserungsmöglichkeiten gibt, wenn man Symmetrien ausnutzt und bei gleichen Gewichten nicht dasselbe mehrmals probiert.

3. Eine allgemeine Suchfunktion, die Tiefensuche verwendet und erfolglose Suche abschneidet ist:

```
suchbtdf :: a -> (a->[a]) -> (a-> Bool) -> (a-> Bool) -> Maybe a
--  anf:  aktuelles Objekt (Zustand)
--  toechter: Nachfolgezustaende
--  ziel:  Praedikat, das testet ob der aktuelle Zustand ein Zielzustand ist.
--  cut:  Wenn erfuellt, dann werden keine Nachfolgezustaende untersucht
suchbtdf anf toechter ziel cut =
   if ziel anf then Just anf
   else if cut anf
     then Nothing
     else let nexts = toechter anf
```

```
          in suchbtdfl nexts toechter ziel cut
suchbtdfl [] toechter ziel cut = Nothing
suchbtdfl (x:xs) toechter ziel cut =
   let  result1 = suchbtdf x toechter ziel cut
        result2 =  suchbtdfl xs toechter ziel cut
    in case result1 of
          Nothing -> result2
          Just x -> Just x

testgewicht_allgem = suchbtdftr ([],[5,5,3,3,3,3])
                     (\(xs,ys) -> if ys == [] then []
                                  else [(head ys:xs,tail ys),(xs,tail ys)] )
                     (\(xs,_) -> (sum xs) == 12)
                     (\(xs,_) -> (sum xs) > 12)

---   75 Schritte


testgewicht_allgemmv =
    suchbtdftr ([],[(5,2),(3,4)])
       (\(xs,ys) -> if ys == [] then []
         else let (yh1,yh2):yr = ys
              in if yh2 == 1
                 then [(yh1:xs,yr),(xs,yr)]
                 else [(yh1:xs,(yh1,yh2-1):yr),(xs,yr)])
       (\(xs,_) -> (sum xs) == 12)
       (\(xs,_) -> (sum xs) > 12)

---   17 Schritte
```

4.2.2 Greedy Algorithmen, lokal optimale Schritte

Die Strategie "greedy" (gefräßig) ist die Implementierung der Idee, bei der Suche jeweils den lokal optimalsten Schritt zu machen und danach, aufbauend auf der Teillösung, weiter zu suchen. Dies kann einen Verzicht auf eine optimale Lösung bedeuten, allerdings ist der Algorithmus schnell und man erhält oft eine fast optimale Lösung. Bei exakten Lösungen wie dem Gewichtsproblem oben ist die Strategie eher von geringem Nutzen.

Wir betrachten eine Abwandlung des Gewichtsbeispiels und nennen es das Paketproblem: Gegeben einige Teile, die im Paket verpackt werden sollen und eine Obergrenze des Gesamtgewichts, ab der das Paket zu teuer wird, soll eine Zusammenstellung der Teile angegeben werden, die das maximale Gewicht möglichst optimal ausnutzt.

Wir vereinfachen wieder und betrachten nur Listen von Zahlen.

- Zum Vergleich die Methode, die den optimalen Wert findet, indem alle Unterlisten erzeugt werden und danach die optimalen herausgefiltert werden.

```
postmin1 xs max =
   let ulisten = unterlisten xs
       wertuliste = zip (map sum ulisten) ulisten
       okliste = filter (\(w,xs) -> w <= max) wertuliste
       sortliste = mischsortg okliste (\(x,_) (y,_) -> x > y)
    in head sortliste
```

- Dies ist eine Implementierung der greedy-Methode, die immer den lokal optimalen Wert nimmt, auch wenn sie sich daran verschluckt. Man muß sagen, was lokal optimal bedeutet: hier nehmen wir das jeweils größte Gewicht, das noch in das Paket hineinpasst. Dieses Verfahren ist in diesem speziellen Fall nicht optimal, wie die Beispiele zeigen. Es gibt allerdings auch Beispiele, in denen das greedy-Verfahren immer den optimalen Wert findet.

```
postgreedy xs max = postgreedyw xs max []
postgreedyw [] max res = (sum res,res)
postgreedyw _ 0 res = (sum res,res)
postgreedyw xs max res =
   let  ls = filter (<= max) xs
        m = maximum ls
     in case ls of
        [] -> (sum res,res)
        _ -> postgreedyw (delete m xs) (max - m) (m:res)

--  Beispiele:
--- postgreedy [1,2,3,4,5] 12  --->  (12,[3, 4, 5])

--  nicht optimal, aber nahe dran:
--  postgreedy [5,5,3,3,3,3] 12  --->    (10,[5, 5])
--  postmin1 [5,5,3,3,3,3] 12    --->    (12,[3, 3, 3, 3])
--  postgreedy [5,5,3,3,3,3] 14  --->    (13,[3, 5, 5])
--  postmin1 [5,5,3,3,3,3] 14    --->    (14,[5, 3, 3, 3])
```

- Huffman-Kodierung zur Codierung von Nachrichten (bzw. Kompression von Dateien.)

  Die Problemstellung ist analog zum Morsecode: gegeben eine lange Nachricht (Datei), über einem Alphabet A, von dem die relativen Häufigkeiten der einzelnen Zeichen bekannt ist. Zur Übertragung der Nachricht (zur Kompression der Datei) steht ein anderes (hier ein kleineres) Alphabet B zur Verfügung. Finde eine Kodierungsfunktion c die alle Zeichen von A in Worte von B überführt, so daß der Übermittlungsaufwand der kodierten Nachricht möglichst klein ist. D.h. die Anzahl der benutzten Zeichen des Alphabets B in der Nachricht minimal wird. Dies ist asymptotisch gleich der mittleren Länge des Kodes $C := \{c(a) \mid a \in A\}$, gewichtet mit der relativen Häufigkeit der Zeichen in A. Wir nehmen $B = \{0, 1\}$.

Bei der Huffman-Kodierung wird eine Kodierung c gewählt, die die Originalzeichen durch Worte ersetzt, so daß die Menge der Worte C $:= \{c(a) \mid a \in A\}$ präfixfrei ist, d.h. kein Wort in C ist Präfix eines anderen in C. Die Worte in C können verschiedene Längen haben. Insgesamt nennt man das eine Kodierung.

Zu einer solchen präfixfreien Kodierung kann ein binärer Baum aufgebaut werden, der die Dekodierung sehr einfach macht. Die Kodeworte sind gerade die Adressen der Blätter (0: nach links, 1: nach rechts), die Markierungen der Blätter sind die gesuchten Kode-Buchstaben. D.h. man kann eindeutig und auch schnell dekodieren.

Die Codierung eines Wortes $b_1b_2 \ldots b_m$ ist $c(b_1)c(b_2)\ldots c(b_m)$. Wenn der Anfang bekannt ist, dann kann man beim Dekodieren jeweils den Baum hinunterlaufen. Wenn man am Blatt angekommen ist, hat man das kodierte Zeichen und dekodiert dann das nächste Wort.

Vergleicht man jetzt mit dem Morse-Kode, so erkennt man, daß dieser kein Huffman-code ist:

a	·–	ä	·–·–	b	–···	c	–·–·
ch	––––	d	–··	e	·	f	··–·
g	––·	h	····	i	··	j	·–––
…							

Denn es gibt Verletzungen der Präfixfreiheit. Die Benutzung des Morsekodes war tatsächlich so, daß es eigentlich noch ein extra Zeichen gab, nämlich eine Pause zwischen den Zeichen, so daß diese eindeutig trennbar waren. Die Länge der Kodierungen der Buchstaben wurde auch beim Morsekode nach der Auftrittswahrscheinlichkeit der Buchstaben in englischen Texten bestimmt.

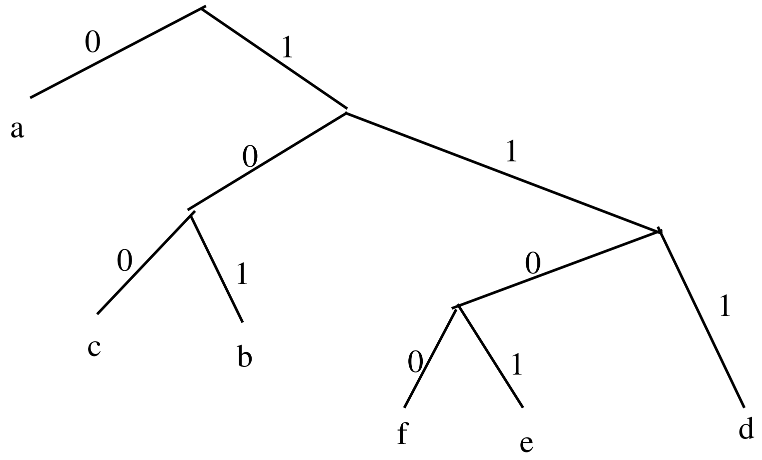
Seien $a_1, \ldots, a_n$ die zu kodierenden Zeichen und $p_1, \ldots, p_n$ die relativen Häufigkeiten. Beachte $\sum_{i=1}^{n} p_i = 1$. Der zugehörige Code soll aus Worten über $\{0,1\}$ bestehen, d.h. $c(a_i) \in \{0,1\}^*$, und es soll $\sum_{i=1}^{n} |c(a_i)| * p_i$, die mittlere Kodewortlänge, minimal werden.

Ein greedy-Algorithmus zur Berechnung eines optimalen Kodes baut einen Code-Baum auf, wobei man mit einer Liste von Bäumen startet, die nur aus einem Blatt bestehen und von denen die Summe der relativen Häufigkeit der Blätter bekannt ist. Es werden nun sukzessive immer die beiden Bäume mit der geringsten Häufigkeit genommen und als 0-1 kodiert, d.h. es wird ein neuer Baum erzeugt, der als neue Teilbäume gerade diese beiden Bäume hat. Danach hat man einen Baum weniger in der Liste und iteriert dieses Verfahren bis nur noch ein Baum übrig ist. Dieser ist dann der gesuchte Code-Baum.

Dieser greedy-Algorithmus zum Aufbau des Kode-Baumes erzeugt tatsächlich immer einen optimalen Huffman-Kode, und nicht nur einen suboptimalen, was nicht zu schwer zu beweisen ist (aber nicht Gegenstand dieses Skripts) .

Man hat offenbar jedesmal die Wahl zwischen 0 und 1, so daß diese Kodes durch die Optimalitätsforderung nicht eindeutig bestimmt sind.

```
--   Huffman encoding.

data Hufftree a = Hleaf a Float
                | Hnode (Hufftree a) (Hufftree a) Float

huffle :: Hufftree a -> Hufftree a -> Bool
huffle htr1 htr2 = (huffrh htr1) <= (huffrh htr2)

huffrh (Hleaf _ x) = x
huffrh (Hnode _ _ x) = x

huffgreedy xs = huffgreedyw (map (\(x,p)->(Hleaf x p)) xs)

huffgreedyw [x] = x
huffgreedyw xs@(_:_) =
    let   y1:(y2:yr)  = mischsortg xs huffle
      in huffgreedyw ((Hnode y1 y2 ((huffrh y1) + (huffrh y2))):yr)

huffextractcode ht =
   mischsortg  (map (\(x,y) -> (x,reverse y)) (huffxcw "" ht))
               (\(x1,_) (x2,_)  -> x1 <= x2)
huffxcw prefix (Hleaf x _) = [(x,prefix)]
huffxcw prefix (Hnode tl tr _) =
     (huffxcw ('0':prefix) tl) ++  (huffxcw ('1':prefix) tr)


testhuffgreedy =
   let   probs =
           [('a',0.45),('b', 0.13),('c',0.12),('d',0.16),('e',0.09),('f',0.05)]
         tr = huffgreedy probs
     in (mittlere_codewortlaenge tr, huffextractcode tr)

mittlere_codewortlaenge tr = mitt_cwl tr 0.0
mitt_cwl (Hleaf _ x) tiefe = x*tiefe
mitt_cwl (Hnode tl tr x) tiefe =
        (mitt_cwl tl (tiefe + 1.0)) + (mitt_cwl tr (tiefe + 1.0))


> testhuffgreedy
> (2.24,[('a',"0"), ('b',"101"), ('c',"100"),
         ('d',"111"), ('e',"1101"), ('f',"1100")])
```



Der Vollständigkeit halber das Kodieren und Dekodieren unter Benutzung der Huffman-Kodierung. Schneller ist die Kodierung, wenn man zum Kodieren ein Feld benutzt und mittels ord(.) (analog zu Hash-Funktionen) den Index im Feld berechnet.

```
huffkodiere xs tr = huffcode xs (huffextractcode tr)
huffcode [] tr     = []
huffcode (x:xs) tr = (kodiere x tr) ++ huffcode xs tr
kodiere x ((a,ca):xs) = if x == a then ca else kodiere x  xs

huffdekodiere xs tr =  huffdecode xs tr tr
huffdecode [] (Hleaf a _) _ = [a]
huffdecode xs (Hleaf a _) tr = a : (huffdecode xs tr tr)
huffdecode ('0':xs) (Hnode trl trr _) tr = huffdecode xs  trl tr
huffdecode ('1':xs) (Hnode trl trr _) tr = huffdecode xs  trr tr

testkodiere =
  let wort = "badfadcade"
      tr = huffgreedy hufftestprobs
      kodiertes_wort = huffkodiere wort tr
      dekodiertes_wort =  huffdekodiere  kodiertes_wort tr
   in (kodiertes_wort, dekodiertes_wort,wort, dekodiertes_wort == wort)

>  testkodiere
>  ("1010111110001111000111110 1","badfadcade","badfadcade",True)
```

Praktisch verwendete Kompressionsverfahren arbeiten i.a. anders. Z.B. werden bei der normalen Kompression ganze Zeichenketten kodiert, so daß bei der Kompression eines Textes / Programms häufig vorkommende Worte jeweils gleich und kurz kodiert werden.

4.3 Scan: lineare Algorithmen

Um Algorithmen zu konstruieren, die bestimmte Probleme in linearer Zeit lösen, ist es i.a. ratsam, eine Vorgehensweise zu wählen, die von dem divide and conquer Verfahren wie im Mischsort verschieden ist. Problemstellungen für Listen von Zahlen, die in Linearzeit gelöst werden sollen, und die fast alle Elemente anfassen müssen, sind am einfachsten zu konstruieren, wenn man die Liste von vorne nach hinten durchläuft (scanned) und bei jedem Schritt nur einen konstanten Zeitbedarf hat. Der beste Fall ist, daß man nur einen Durchlauf machen muß. Die Vorstellung ist, daß man eine große sequentielle Datei (einen File) zu bearbeiten hat, (z.B. ein File, der auf einem Band gespeichert ist) zu verarbeiten, bei dem man nur einen begrenzten lokalen Speicher hat und bei dem das Programm pro Arbeitseinheit (Zeichen, Satz, o.ä.) nur konstanten Zeitbedarf und auch Platzbedarf hat. Ein zweiter Durchlauf würde ein neues Durchlesen des Files bedeuten.

Mehrere Durchläufe sind erlaubt, wenn die Anzahl der Durchläufe eine feste obere Schranke hat. Der asymptotische Zeitbedarf kann dann linear bleiben, allerdings erhöht sich der konstante Faktor des Zeitbedarfs.

In verzögert auswertenden funktionalen Programmiersprachen (in Hugs) ist es einer Funktion nicht so ohne weiteres anzusehen, ob diese einen Durchlauf über eine Liste macht, oder mehrere: Z.B. `filter p (map q xs))` ist nur ein Durchlauf, da `map` jeweils ein Element verarbeitet, es dann an `filter` weiterreicht, bevor ein weiteres Element verarbeitet wird. In ungünstigen Fällen kann es passieren, daß aufgrund der Anforderungen die ganze Liste gleichzeitig im Hauptspeicher gehalten werden muß, z.B. bei `map q (reverse xs)`.

Beispiel 2. Die Berechnung des Maximums einer Liste von Zahlen kann man durchführen, indem man zunächst sortiert, dann das erste Element auswählt. Dies ergibt einen Algorithmus mit Zeitbedarf $O(n * \log(n))$, wenn n die Länge der Liste ist.

Dies wird zu einem linearen Algorithmus, wenn man die Liste von vorne nach hinten absucht:

```
minim (x:xs) = minimr x xs
minimu x [] = x
minimr x (y:ys) = if x <= y then minimr x ys
                else minimr y ys
```

Berechnet man das Minimum mit dem Ausdruck

```
minim_sort xs = head (mischsort xs)
```

dann ergibt die naive Analyse, daß man einen Zeitbedarf von $O(n * log(n)$ hat, allerdings wird eine genauere Analyse ergeben, daß man in Wahrheit doch nur einen $O(n)$-Algorithmus hat, denn es wird nur soviel sortiert, wie zur Bestimmung des Minimums notwendig ist.

Beispiel 3. In linearer Zeit kann man auch die Suche eines Wortes in einem langen String (Textdatei) durchführen, wenn man als Eingabe das Wort und den String mißt.

Beispiel 4. Die Erzeugung eines optimalen Huffman-Kodes indem man zunächst die Datei durchläuft, die Häufigkeiten der Zeichen ermittelt, und die nachfolgende Kodierung erforden insgesamt zwei Durchläufe. Es gibt Kodierungsalgorithmen, die nur einen Durchlauf machen.

Übungsaufgabe 2. Gegeben eine Liste von n Zahlen. Beschreibe einen Algorithmus, der herausfindet, ob es eine Zahl gibt, die öfter als $n/2$ mal in der Liste vorkommt. Gibt es einen linearen Algorithmus?.

4.4 Beispiele

4.4.1 Median einer Liste

Der (ein) Median einer Liste von Zahlen ist ein Element der Liste, das von der Größe her in der Mitte liegt. D.h. sowohl die Anzahl der kleineren ($\leq$) als auch die Anzahl der größeren ($\geq$) ist mindestens die Hälfte der Anzahl der Elemente in der Liste. Z.B. der Median von $[1, 2, 3, 1000, 2000]$ ist 3, ein Median von $[1, 2, 1000, 2000]$ ist 2 bzw. 1000. Ein Median wird z.T: für statistische Zwecke verwendet.

- Die einfachste Methode, einen Median zu finden, ist die Vorgehensweise, zunächst die Liste zu sortieren, und dann das mittlere Element zu nehmen. Wie wir wissen, erfordert diese Vorgehensweise $O(n * log(n)$ Zeitbedarf, obwohl man doch eine einfachere Aufgabe gestellt hat als das Sortieren.

  Verwendet man Z.B.

  ```
  take (n/2) (mischsort xs)}
  ```

  und erhofft sich eine Verbesserung von der lazy Auswertung gegenüber Mischsort,, dann trifft das zu, aber die Anzahl Schritte ist vermutlich nicht $O(n)$.

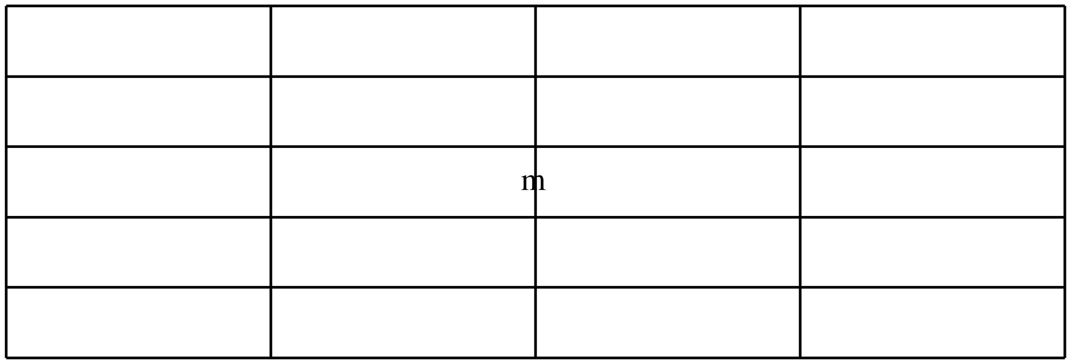
- Teile-und-Herrsche wie beim Merge-Sort

  1. Zerlege die Liste in zwei gleiche Teile xs_1, xs_2.
  2. bestimme die Mediane m_1, m_2 von xs_1, xs_2.
  3. Sei $m_1 \leq m_2$. Dann ist offenbar $m_1 \leq m \leq m_2$. Zerlege xs_1 in $xs_{11} \leq m_1 \leq xs_{12}$ und xs_2 in $xs_{21} \leq m_2 \leq xs_{22}$. Danach wende den Median auf $xs_{12}{+}{+}xs_{21}$ an.

  Das Problem dieser Methode ist, daß die Summe der Größen der Teilprobleme das 1.5-fache des Originalproblems ist. D.h. im schlechtesten Fall hat dieser Algorithmus asymptotische Laufzeit größer als $O(n * log(n))$.

- Teile-und-Herrsche.
  Verallgemeinere das Median- Problem zu: finde das k-te Elemente in der Sortierreihenfolge einer Liste. Der Median kann dann als das Element an der Stelle $n/2$ ermittelt werden.

1. Zerlege die Liste in kurze Listen der Länge 5. Bestimme deren Mediane
2. Sei M die Liste der Mediane Bestimme deren Mediane m.
3. Finde die Listen $S_{<m}, S_{=m}, S_{>m}$.
4. Wenn $\#(S_{<m}) \geq k$, dann finde k-tes Element in $S_{<m}$
5. Wenn $\#(S_{<m}) \leq k$ und $\#(S_{\leq m}) \geq k$, dann ist m als der gesuchte Wert ermittelt.
6. Wenn $\#S_{\leq m} < k$, dann sei $k' = k - \#S_{\leq m}$. Ermittle das k'-te Element in $\#S_{>m}$

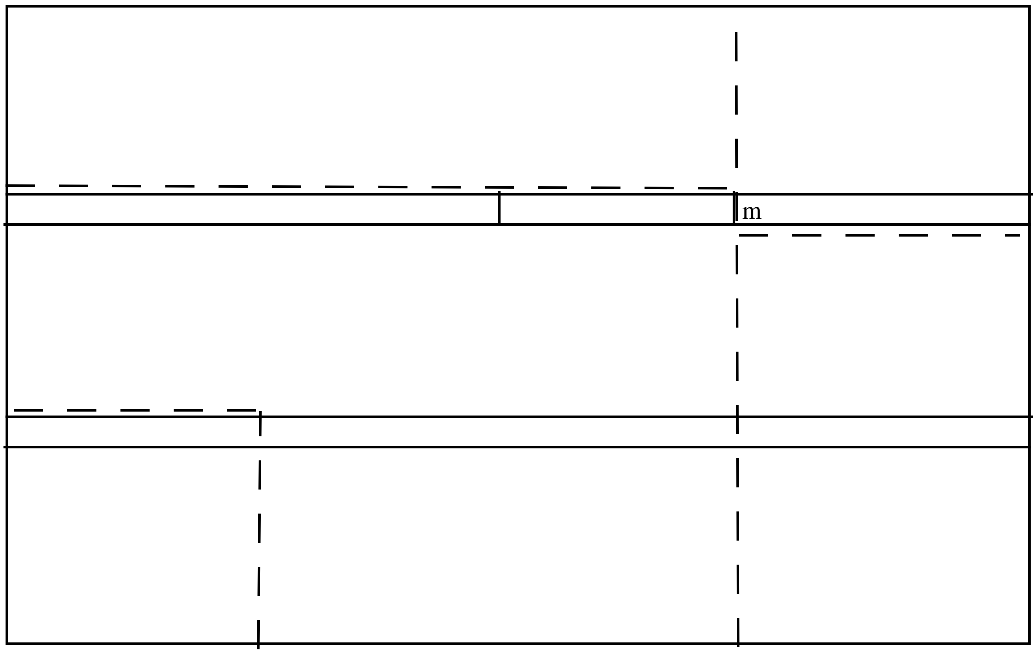


Wir bestimmen die Summe der Größen der Teilprobleme: Die rekursive Medianbestimmung von M trägt $1/5$ bei, wobei wir die Mediane der kurzen Listen der Länge 5 insgesamt in linearer Zeit finden können. Man sieht in der Zeichnung: mindestens $3/10$ der Elemente sind größer und mindestens $3/10$ sind kleiner als m. D.h. im nächsten Rekursionschritt ist die Liste maximal $7/10$ groß. Die Summe ergibt $9/10 < 1$.

Die Rekursionsgleichung zeigt dann, daß diese Art der Medianbestimmung Zeitbedarf $O(n)$ hat, allerdings hat sie einen relativ hohen konstanten Faktor, so daß der asymptotische Vorteil dieses Algorithmus erst bei sehr großen Listen bemerkbar wird.

Man kann einen linearen divide and conquer Algorithmus zur Medianbestimmung ohne Verallgemeinerung finden: Man gibt sich k fest vor, teilt die Liste der Länge n in Stücke der Länge k, sortiert diese und ruft dann die Medianbestimmung für die Liste auf, die aus den Elementen besteht, die an der Stelle $2k/3$ in den sortierten k-Listen stehen. Diese wendet man nochmal an für die Liste der Element der oberen Hälfte. Damit erhält man ein Element m_1, das größer als der Median der Eingabe Liste ist, denn es gibt $(2n/3) * (3/4) = n/2$ kleinere Elemente. Man entfernt somit die größeren, das sind: $n/3 * (1/4) = n/12$. Das gleich wendet man symmetrisch für die kleineren Element an: D.h. Median der Liste der $n/3$-Elemente, dann Median der unteren Hälfte, danach kleinere weglassen. Am Ende ruft man Median für die Liste der übriggebliebenen Element auf.

Die Gesamtrechnung ergibt als Größe der Teilprobleme: $3n/k$ für die Medianbestimmungen der 1/3 und 2/3-Liste; $5n/6$ für den rekursiven Aufruf. Das sind $3n/k + 5n/6$. Damit dies kleiner als 1 ist, muß $3/k + 5/6 < 1$ sein. Dies erfordert $k > 18$.



Mit großer Wahrscheinlichkeit gibt es keinen Scan-Algorithmus für dieses Problem.

4.4.2 Summe von Teilfolgen

Wir betrachten die Aufgabe, die Summe der längsten zusammenhängenden Teilfolge einer endlichen Folge von ganzen Zahlen zu finden, bei der alle Elemente nichtnegativ sind (mcv: maximal contiguous subvector).

Dazu betrachten wir verschiedenen Methoden:

- direktes Verfahren: Bestimme alle Teilfolgen (ohne Lücken), wähle die aus, die nur nichtnegative Elemente haben, bilde die Summe und bestimme das Maximum.

  Dieses Verfahren ist kubisch $O(n^3)$: i) Es gibt quadratisch viele dieser Teillisten, ii) Selektion läuft über jede dieser Listen, d.h. Zeitbedarf $O(n * n^2)$, iii) Summenbildung ist linear in einer Teilliste, also auch $O(n*n^2)$ und Maximumsbestimmung sind linear (d.h. quadratisch). Zusammen $O(n^3)$.

  ```
  startsublists [] = []
  startsublists [x] = [[x]]
  startsublists (x:t) =  [x] : (map (\y->(x:y)) (startsublists t))
  ```

```
nesublists [] = []
nesublists (x:t) = (startsublists (x:t)) ++ (nesublists t)

mcv_quadrat [] = 0
mcv_quadrat (x:xs) = maximum (map sum (nesublists (x:xs)))
```

- divide and conquer: Halbiere die Liste in der Mitte, bestimme die mcv-Zahlen m_l, m_r rekursiv, berechne zusätzlich die rechte mcv-Zahl m_{lr} der linken Teilliste und linke mcv-Zahl m_{rl} der rechten Teilliste. Bilde das Maximum der Zahlen $m_l, m_r, m_{rl} + m_{lr}$.

```
mcv_dq [x] = if x <= 0 then 0 else x
mcv_dq (x:xs) = let len =length (x:xs)
                    xa = take (len `div` 2) (x:xs)
                    xb = drop (len `div` 2) (x:xs)
                    mca = mcv_dq xa
                    mcb = mcv_dq xb
                    mca_r = sum (takeWhile (>= 0) (reverse xa))
                    mcb_l = sum (takeWhile (>= 0) xb)
                  in maximum [mca,mcb,mca_r+mcb_l]
```

- Scan-Verfahren: Das Verfahren benötigt zum Durchlauf der Liste eine Umgebung: Das bisher gefundene Maximum, die Summe der aktuellen Teilliste und die Restliste. Dieser Algorithmus ist linear in der Länge der Liste, da er pro Element nur einen konstanten Zeitbedarf hat.

```
mcv_scan x = fst (mcv_scan_h x)
mcv_scan_h []   = (0,0)
mcv_scan_h [x] = let maxsum = if x <= 0 then 0 else x
                        in (x,x)

mcv_scan_h (x:t) = let  (maxsum, maxsuml) = mcv_scan_h t
                        maxsum_n = max maxsum (maxsuml + x)
                        maxsuml_n = if x >= 0 then max maxsuml (maxsuml + x)
                                     else 0
                      in (maxsum_n,maxsuml_n)
```

4.5 Parallele Algorithmen und deren Ressourcenbedarf

Parallele Berechnung (oder konkurrierende, oder verteilte) bedeutet i.a. die Durchführung einer Berechnung (eines Prozesses) durch Zusammenwirken verschiedener Computer oder Prozessoreinheiten. Es gibt grundverschiedene Modellvorstellungen dieser Berechnungen:

Parallel Mehrere Prozessoren haben einen gemeinsamen Hauptspeicher und können unabhängig lesen und schreiben. Bei Untersuchungen unterscheidet man danach, ob Lese- und/oder Schreibzugriffe exklusiv oder konkurrierend sind (EREW, CRCW, CREW).

Verteilt, lose Kopplung Mehrere unabhängige Rechner kommunizieren über ein Netzwerk und arbeiten gemeinsam an einer Berechnung. Hier unterscheidet man zum Teil nach der Art der Kommunikation (synchronisierte / nicht synchronisiert) und danach, ob die Berechnung hierarchisch gegliedert ist (Client / Server).

massiv parallel, enge Kopplung Sehr viele unabhängige Rechner sind über ein schnelles Netzwerk gekoppelt und arbeiten gemeinsam an einer Berechnung. Beispiele: Hyperwürfel und ähnliche Netzwerke, Hardware für künstliche neuronale Netze.

Vektorrechner, Feldrechner Hier gibt es ein Programm, das gleichzeitig auf viele Daten angewendet wird. Sinnvoller Einsatz: Wettersimulationen, numerische Berechnungen . Diesen Typ bezeichnet man auch als SIMD (single instruction, multiple data)

Pipelining Hier ist meist die parallele Ausführung von Maschinenkode auf der Hardware eines Prozessor gemeint. Das Pipelining beschleunigt diese Ausführung zunächst dadurch, daß die Befehlsbearbeitung in kleinere Einheiten zerlegt wird, die dann nacheinander abgearbeitet werden, so daß mehrere Maschinenbefehle quasi parallel abgearbeitet werden. Durch Mehrfachauslegung von internen Einheiten kann man das noch weiter beschleunigen.

Hier betrachten wir parallele Verarbeitung auf der Auswertungsebene, d.h. wir können uns eine Hardware vorstellen, bei der mehrere Prozessoren auf den gleichen Speicher lesend und schreibend zugreifen (shared memory architecture).

Um für einen gegebenen Algorithmus (ein gegebenes Programm) Aussagen machen zu können über maximale Beschleunigung, den Bedarf einer Berechnung an Anzahl Prozessoren u. ä. , verwenden wir als vereinfachtes Modell die bisherige Auswertungsmethode für Haskell, wobei wir erlauben, daß unabhängige Transformationsschritte parallel durchgeführt werden dürfen. Um dies formal korrekt durchzuführen, stellen wir uns vor, daß Haskell-Ausdrücke als markierte gerichtete Graphen vorliegen, so daß Transformationen als Veränderungen in diesem Graphen angesehen werden können.

Hierbei müssen wir festlegen, was ein Transformationseinzelschritt ist: Wir vereinbaren: dies soll eine β-Reduktion, eine case-Reduktion oder die Reduktion einer eingebauten arithmetischen Funktion $(+, -, *, > . <, \geq, \leq, \ldots)$ sein. Dies schließt Listenkomprehensionen aus. Wir gehen also davon aus, daß Listenkomprehensionen in expandierter Form, d.h. als Ausdrücke unter Benutzung von `map, filter, concat` vorliegen. Bei den eingebauten Funktionen, die direkt auswerten, ziehen wir eine Grenze, so daß eine arithmetische Auswertungen (von $+$) ein Schritt ist.

Für die folgenden Betrachtungen muß man natürlich annehmen, daß die Prozessoren nicht unterscheidbar sind, d.h. alle die gleiche Arbeit leisten können.

- Die gleichzeitige Durchführung einer Menge von unabhängigen Auswertungseinzelschritten ist dann *ein paralleler Auswertungsschritt*.

  Die Anzahl dieser Schritte bis zu einem Ergebnis ist dann die Anzahl der parallelen Reduktionsschritte.

- Die maximale Anzahl gleichzeitiger Transformationen in einem Schritt entspricht der Anzahl der parallelen Prozessoren

Definition 3.

- Die Parallelisierung ist *konservativ*, wenn nur Reduktionen durchgeführt werden, die für das Erreichen des Resultats notwendig sind.

- Die Parallelisierung ist *spekulativ*, wenn auch Reduktionen durchgeführt werden können, die nicht für das Berechnen des Resultats notwendig sind.

Beispiel 5. Die normale Ordnung der Auswertung in Haskell macht nur notwendige Reduktionen, und berechnet einen Wert, wenn es eine Reduktion zu einem Wert gibt.

Die parallele Reduktion von s, t in einem Ausdruck

```
if exp then s else t
```

führt i.a. überflüssige Reduktion durch. Allerdings weiß man dies erst nachdem `exp` zu einem Wert reduziert worden ist.

Sei ein Algorithmus gegeben. n die Größe der jeweiligen Eingabe, p die Anzahl der Prozessoren. Gegeben sei ein Algorithmus in Haskell. Dann sei $\tau(n,p)$ die minimale Anzahl der parallelen Reduktionsschritte bis zum Ergebnis, wenn man p parallele Reduktionsschritte pro Schritt machen darf. $\tau(n,1)$ ist dann der sequentielle Reduktionsbedarf (sequentielle Zeit).

- Den Faktor $\dfrac{\tau(n,1)}{\tau(n,p)}$ nennt man (relative) *parallele Beschleunigung.*

  Die parallele Beschleunigung ist eine Zahl zwischen 1 und p. und gibt den Faktor an, um den eine Berechnung beschleunigt wird. Diese Zahl kann nicht kleiner als 1 sein in unserem Modell, da stets eine Reduktion möglich ist. Sie kann aber auch nicht größer als p sein, denn eine parallele Reduktion zu einem Ergebnis kann man sequentiell nachvollziehen.

- Den Faktor $\dfrac{\tau(n,1}{p * \tau(n,p)}$ nennt man *parallele Effizienz*. Dies ist der Anteil der für den Algorithmus nutzbaren Prozessorleistung.

	$\tau(100,1)$	$\tau(100,3)$	$\tau(100,4)$
Zeit	1000	500	400
Beschleunigung	1	2	2,5
Effizienz	1	66,6%	62,5%

Die parallele Effizienz ist eine Zahl zwischen 1 und $1/p$. 1 ist das Optimum und zeigt an, daß alle Prozessoren zur Berechnung beitragen. $1/p$ bedeutet, daß die Berechnung im wesentlichen sequentiell ist und nur einen Prozessor auslasten kann.

> Prinzip: Ist für eine Anzahl p von Prozessoren die parallele Effizienz =1, dann gilt das im wesentlichen auch für kleinere Anzahlen von Prozessoren.

Denn man kann die parallelen Reduktionen ja auf einer kleineren Anzahl von Prozessoren ebenfalls ausführen, nur etwas sequentialisierter.

- Den Faktor $\dfrac{\tau(n,1}{\tau(n,\infty)}$ nennt man *maximale parallele Beschleunigung.*

  Die maximale parallele Beschleunigung ist der Wert, den man erhält, wenn man eine unbeschränkte Anzahl von Prozessoren hat. Diesen Wert kann man als obere Schranke einer praktisch erreichbaren parallelen Beschleunigung ansehen. Wenn die maximale Beschleunigung q ist mit $1 \leq q \leq p$, dann kann man den sequentiellen Anteil der zur Berechnung benötigten Zeit ausdrücken als: $1/q$.

  Hierzu paßt Amdahls Gesetz der parallelen Beschleunigung: Von einer Berechnung kennen wir die Zusammensetzung der Gesamtzeit aus parallelisierbarem Anteil und sequentiellem Anteil: $T = T_{par} + T_{seq}$. Dann kann man die Beschleunigung durch p Prozessoren beschreiben als $\dfrac{T_{par} + T_{seq}}{(1/p) * T_{par} + T_{seq}}$. Man sieht dann, daß der sequentielle Anteil eine untere Grenze der Beschleunigung darstellt.

- Die Anzahl $w(n,p)$ (die verrichtete Reduktionsarbeit), sei die minimale Anzahl von Reduktionsschritten aller parallelen Reduktionen, die mit $\tau(n,p)$ Reduktionsschritten ein Ergebnis berechnen.

- Ein Maß für die mittlere Anzahl beschäftigter Prozessoren läßt sich berechnen als $\dfrac{w(n,p)}{\tau(n,p)}$. Die Zahl $\dfrac{w(n,p)}{p * \tau(n,p)}$, könnte man die *mittlere Auslastung* nennen.

Die insgesamt verrichtete Arbeit $w(n,p)$ ist immer größer als oder gleich$\tau(n,1)$, die sequentielle Zeit. I.a. ist für einen optimal parallel beschleunigten Algorithmus die insgesamt verrichtete Arbeit höher als für den besten sequentiellen Algorithmus.

Nimmt man Arbeit = Stromverbrauch, parallele Reduktion = Zeit, so kann man nicht immer gleichzeitig den Stromverbrauch und die Zeit minimieren.

Analog zum asymptotischen Ressourcenbedarf geben wir auch hier Abschätzungen im Sinne der O-Notation an.

Beispiel 6.

- `quadratsumme` 2 2 $\longrightarrow$ (2*2)+(2*2) $\longrightarrow$ 4+4 $\longrightarrow$ 8.

- `fak` 3
  $\longrightarrow$ `if` 3 == 1 `then` 1 `else` 3*(`fak` (3-1))
  $\longrightarrow$ `if` False `then` 1 `else` 3*(`if` 2 == 1 `then` 1 `else` 2*(`fak` (2-1))
  $\longrightarrow$ 3*(`if` False `then` 1 `else` 2*(`if` 1 == 1 `then` 1 `else` 1*(`fak` (1-1)))
  $\longrightarrow$ 3*(2*(`if` True `then` 1 `else` 1*(`if` 0 == 1 `then` 1 `else` 1*(`fak`0)))
  $\longrightarrow$ 3*(2*(1))
  $\longrightarrow$ 3*(2)
  $\longrightarrow$ 6
  Dies sind 7 parallele Auswertungsschritte. Man sieht (Übungsaufgabe), daß diese Anzahl linear in n ist. D.h. selbst durch massiven Einsatz von vielen Prozessoren läßt sich dieser Algorithmus nicht wesentlich beschleunigen.

Übungsaufgabe 4. Schreibe einen Algorithmus, der bei paralleler Auswertung die Fakultät in logarithmischer Zeit berechnet, wenn das Argument eine Zahl (und kein Ausdruck) ist.

Es kann vorkommen, daß in diesem parallelen Modell unnötige Transformationen gemacht werden. Wenn wir uns für den geringstmöglichen Bedarf (bzw. für eine untere Grenze) der parallelen Reduktionsschritten (Zeit) interessieren, dann ist unser Modell zur einfachen Berechnung dieser Zahl geeignet. Wenn wir allerdings die Anzahl der parallel möglichen Transformationen beschränken wollen, da z.B. nur eine feste Anzahl an Prozessoren zur Verfügung steht, dann ist das eine schwerere Aufgabe, da wir über viele mögliche parallele Reduktionsfolgen minimieren müssen. Wir betrachten hier nur die maximal mögliche Parallelität und die schnellstmögliche Verarbeitung. Hierzu haben wir im Modell auch eine Vereinfachung gemacht: die Vernachlässigung der Kommunikation und der Handhabung der Ausdrücke, d.h. wir vernachlässigen die Zeit zur Suche nach den Reduktionsmöglichkeiten, die Zeit zur Ersetzung der Ausdrücke und die Zeit für Reorganisation des gerichteten Graphen.

Beispiel 7. Betrachte den Ausdruck `map quadrat [1..n]`. Dieser Ausdruck evaluiert nach folgendem Schema:
`map quadrat [1..n]`
1: `map quadrat [2..n]`
1: 4: `(map quadrat [3..n])`

Man sieht, daß dies auch bei beliebiger unabhängiger Transformation n parallele Schritte benötigt. D.h. dieser Algorithmus läßt sich analog zum normalen Algorithmus für Fakultät nicht durch parallele Auswertung beschleunigen.

Man sieht, daß man bei Optimierung bzgl. paralleler Auswertung beachten muß, daß die Anzahl der sequentiellen Abhängigkeiten gering bleibt und daß man andere Algorithmen schreiben muß, die besser für parallele Auswertung geeignet sind.

Beispiel 8. Wir betrachten verschiedene Algorithmen zur Summation von n Zahlen.

- Wenn die n Zahlen in einer Liste gegeben sind, dann kann man den Algorithmus `sum` nehmen:

  ```
  sum [] = 0
  sum (x:xs) = x+ (sum xs)
  ```

  `sum`$[n, n-1, ..., 1]$ ergibt n β-Schritte, einen Schritt `sum[]`, und n Summationen. Dies sind $2 * n + 1$ Schritte, d.h. $O(n)$ Reduktionsschritte.

- Wir nehmen an, daß die Zahlen in einem balancierten binären Baum gegeben sind. Dann nehmen wir folgenden Algorithmus:

  ```
  sumbt (Bblatt x) = x
  sumbt (Bknoten bl  br) = (sumbt bl) + (sumbt br)
  ```

  Das ergibt für einen binären Baum der Tiefe h, wenn ein Blatt die Tiefe 0 hat: $2 * (h + 1)$ Schritte, d.h., da der Baum balanciert ist, weniger als $\log_2(n) + 1$ Schritte, d.h. dieser parallele Algorithmus benötigt eine logarithmische Anzahl von parallelen Transformationen. Dieser Algorithmus ist damit viel besser für parallele Verarbeitung geeignet als der erste.

- Schnelles Fold über Bäume: Man erkennt jetzt die Problematik: Wenn man nur eine sequentielle Auswertungsmaschine hat, dann ist das `foldbt` die schnellste Methode. Wenn allerdings parallele Auswertung möglich ist, dann ist das `fold` besser, das die Struktur des Baumes berücksichtigt.

Beispiel 9. Paralleles Sortieren von (verschiedenen) Zahlen. Wir wollen hier nur ein Beispielproblem vorführen und zeigen, daß man parallel wesentlich schneller sortieren kann als sequentiell. Nehme an, daß die Zahlen als ein balancierter binärer Baum eingegeben werden.

Die Idee dieses Algorithmus ist die parallele Ermittlung der Rangfolge einer Zahl in der Folge der sortierten Zahlen.

```
csnrtop bb = csnr bb bb
csnr (Bblatt x) bb = Bblatt (x,sumbt (snr x bb))
csnr (Bknoten bl br) bb = Bknoten (csnr bl bb) (csnr br bb)

--    zaehlt die kleineren
snr x (Bblatt y) = Bblatt (if y <= x then 1 else 0)
snr x (Bknoten bl br) = Bknoten (snr x bl) (snr x br)
```

Wir versuchen für diesen Algorithmus die parallele Laufzeit abzuschätzen. Sei h die Tiefe des binären Baumes.

1. `csnr` benötigt von oben bis zu einem Blatt h parallele ReduktionsSchritte.

2. Danach braucht die Auswertung von `snr` innerhalb des `sumbt`-Ausdrucks h Schritte.

3. `sumbt` benötigt ebenfalls h parallele Reduktionsschritte.

In der Summe ergibt dies $O(h)$ Schritte. In Abhängigkeit von n, der Anzahl der Zahlen am Anfang, ergibt dies eine Größenordnung von $O(\mathtt{log}(n)$. Wollte man als Ergebnis eine Liste erzeugen, dann benötigt man zum sequentiellen Erzeugen der Liste n Schritte. Man kann aber in paralleler Zeit $O(\mathtt{log}(n))$ einen binären Baum erzeugen, der von links nach rechts sortiert ist.

Eine parallele Erzeugung der Liste sortierter Zahlen aus einem links-rechts geordneten binären Baum funktioniert mit folgendem Algorithmus:

```
--   Umbau in einen sortierten Baum
gen_sbaum bb =
   case gen_sbaumw bb 1 (csanzahl bb)
       of Just x -> x

--    Dieses Funktion erzeugt einen Suchbaum
--    mittels divide und conquer

gen_sbaumw (Bblatt (x,nr)) von bis =
        if von <= nr && nr <= bis
        then Just (Bblatt x)
        else Nothing

gen_sbaumw topbaum@(Bknoten bl br) von bis =
    if von == bis then
        case gen_sbaumw br von bis of
            Just baum -> Just baum
            Nothing -> gen_sbaumw bl von bis
    else
      let mitte = (von+bis) `div` 2
      in if von +1 == bis
         then gen_sbaumw_comb
                (gen_sbaumw topbaum von von)
                (gen_sbaumw topbaum bis bis)
         else gen_sbaumw_comb
                (gen_sbaumw topbaum von mitte)
                (gen_sbaumw topbaum (mitte+1) bis)

--   Kombination der Ergebnisse
gen_sbaumw_comb Nothing Nothing   = Nothing
gen_sbaumw_comb (Just x) Nothing  = Just x
gen_sbaumw_comb Nothing (Just x)  = Just x
gen_sbaumw_comb (Just bl) (Just br)  = Just (Bknoten bl br)


csanzahl (Bblatt _) = 1
csanzahl (Bknoten bl br) = (csanzahl bl) + (csanzahl br)
```

```
--   parallel (logarithmische) Erzeugung einer Liste

slgen (Bblatt x) = [x]
slgen (Bknoten bl br) = slinksgen bl (slgen br)

slinksgen (Bblatt x) tail = x: tail
slinksgen (Bknoten ubl ubr) tail = slinksgen ubl (slinksgen ubr tail)

-- test_sortbaum = (Bknoten
                    (Bknoten (Bblatt 8) (Bblatt 7))
                    (Bknoten (Bblatt 6) (Bblatt 5)))
--  test_sort2 = slgen (gen_sbaum (csnrtop test_sortbaum))
```

Dieser Algorithmus aus `slgen` und `slinksgen` erzeugt bei paralleler Auswertung die sortierte Liste in Zeit $O(\mathtt{log}(n))$:

Die Argumentation: `slgen` kann unabhängig von `rest` bis zur Tiefe des Baumes auswerten. `slinksgen` erzeugt zwei unabhängige zu reduzierende Ausdrücke, die ebenfalls die Tiefe um eins reduzieren. Mit Induktion sieht man: Nur die Tiefe des Baumes bestimmt die Anzahl der parallele Reduktionsschritte.

Damit haben wir als parallelen Zeitbedarf des Sortierens mit dem oben angegebenen Algorithmus: $O(log(n))$, wenn n Zahlen als Blätter in einem balancierten Baum gegeben sind. Dies ergibt dann auch eine obere Schranke des parallelen Zeitbedarf des Sortierens.

Im allgemeinen ist man aber mit der Erzeugung eines Suchbaumes besser bedient als mit der Erzeugung einer sortierten Liste.

Wir ermitteln dazu noch die anderen Zahlen zum Ressourcenbedarf:

- Die parallele Beschleunigung ist, wenn wir etwas lax rechnen: $O(n * log(n)/log(n))$. D,h, wir erhalten eine lineare Beschleunigung. (Eigentlich muß man diese Rechnung etwas genauer machen)

- Um die parallele Effizienz zu ermitteln, benötigt man die Anzahl der Prozessoren, die maximal gleichzeitig beschäftigt sind, bzw. die maximale Anzahl gleichzeitiger Reduktionsschritte. Diese maximale Anzahl wird in der Funktion `csnr` erreicht. Es wird für jedes Blatt des Baumes über eine Kopie des Baumes addiert. Dies erfordert im schlechtesten Fall $n * n$ Prozessoren.

  Die parallele Effizienz ergibt sich dann zu c/n. D.h. die Nutzung der Prozessoren ist ziemlich schlecht.

- Den exakten Gesamtaufwand an Reduktionen zu berechnen sei dem Leser überlassen. Hier soll eine Schätzung versucht werden: Die Summation ergibt $c_1 * n * n$ für das Summieren; $c_2 * n * log(n) * log(n)$ für den Umbau des Baumes, und $c_3 * n * log(n)$ für das Erzeugen der Liste. Der quadratische Term überwiegt asymptotisch. Also ergibt sich $O(n^2)$.

Wenn man nur n Prozessoren zur Verfügung hat, besteht bereits die Gefahr, daß der parallele Algorithmus schlechter wird als ein guter sequentieller

Sortieralgorithmus. Das Beste was man dann noch erwarten kann, ist ein linearer Algorithmus, da die Gesamtanzahl an Reduktion $O(n * n)$ ist.

Man kann aus diesem Algorithmus lernen, daß Sortieren durch Parallelisierung wesentlich beschleunigt werden kann. Allerdings wird ein praktikabler Algorithmus anders aussehen.

Beispiel 10. Parallelisierung des Bubble-Sort. Wir stellen uns dies als ein Array der Länge n vor. Die Prozessoren sollen so operieren, daß benachbarte Feldinhalte vertauscht werden, wenn die Zahlen in falscher Reihenfolge sind. Dafür genügen $n-1$ Prozessoren. Wenn man noch vereinbart, daß die Prozessoren immer abwechselnd operieren, d.h. im ersten Schritt 1,3,5,7,..., im zweiten Schritt 2,4,6,.... Dann ergibt sich, daß nach n Schritten das Feld sortiert ist (Dies ist nicht ganz so einfach zu verifizieren, aber es stimmt: siehe in der Literatur unter "odd-even transposition sort").

Betrachtet man die Maßzahlen dann erhält man:

- parallele Beschleunigung: $\sim log(n)$).
- parallele Effizienz: $\sim log(n)/n$
- Gesamtanzahl an Operationen: $\sim n * n$

D.h. man kann mit n Prozessoren tatsächlich mit dieser Methode einen einfachen Sortieralgorithmus angeben.

Beispiel 11. Die Suche nach einem Element in einem Suchbaum läßt sich durch Parallelisierung nicht beschleunigen. Denn der Baum muß von oben nach unten durchlaufen werden. D.h. die Verarbeitung braucht mindestens soviele Schritte, wie die Höhe des Baumes.

Gedankenexperiment: Wir versuchen, die Suche nach einem Element in konstanter Zeit durchzuführen. Die Idee: Wir verteilen (ordnen zu) die n Zahlen jeweils auf einen Prozessor. Wenn eine bestimmtes Element gesucht wird, dann schauen alle Prozessoren gleichzeitig nach, und vergleichen. Der Prozessor, der die Antwort findet, ruft: "ja".

Die beschriebenen Verarbeitungsschritte sind tatsächlich in konstanter Zeit machbar, allerdings zeigt eine genauere Analyse, das die Modellvorstellung die Wirklichkeit nicht ganz trifft:

1. Wie kommen die n Zahlen in ihren jeweiligen Prozessor?
2. Wie erhalten die Prozessoren den gesuchten Wert?
3. Wie wird die Antwort ermittelt?

Dazu Modell der verteilten Rechner und eines zentralen Rechners, der die Anfrage hat und das Ergebnis benötigt.

1. Man benötigt ein Senden eines Vektors an adressierte Rechner: Wenn alle Rechner mit allen verbunden sind, so kann der Aufbau der Nachrichten schon ein lineares Problem darstellen.

2. Zum versenden der Zahl benötigt man ein Senden an alle (sogenanntes broadcast). Wenn alle Rechner mit allen verbunden sind, so ist es technisch denkbar, daß dies in konstanter Zeit abläuft. Wenn dies nicht der Fall ist, so geht die Anzahl der Vermittlungen ein. Nimmt man eine obere Grenze für die Anzahl der direkt verbundenen Rechner, so geht dieser Vermittlungsaufwand logarithmisch ein.

3. Die Antworten müssen gesammelt und ausgewertet werden. Nimmt man an, daß nur die positive Antwort versendet wird, dann ist das zwar maximal eine Nachricht, aber im Falle daß keine Nachricht kommt, ist unklar, welche Zeit man abwarten muß, bis das "nein" wirklich klar ist. Sendet und empfängt man man alle Antworten, dann ist der Empfangsaufwand hoch.

1. Nimmt man das Modell des gemeinsamen Speichers, dann müssten die Prozessoren (in irgendeinem Parameter) unterschiedlich sein: sie müßten adressierbar sein, da sich sonst die korrekte Verteilung der verschiedenen Zahlen nicht durchführen läßt.

   Dies könnte man in konstanter Zeit durchführen, indem ein Aktivierungsfeld ein für alle Prozessoren sichtbares Kommando enthält: Hole deinen Wert im Feld ... ab:

2. Der gesuchte Wert könnte genauso verteilt werden.

3. Die Antwort kann man sammeln, indem nur die positive Antwort an einer bestimmten vorher vereinbarten Stelle im Speicher abgelegt wird. Wenn die Prozessoren im snychronisierten Takt arbeiten, dann kann auch im Falle nur negativer Antworten dies festgestellt werden

Übungsaufgabe 5. Angenommen, wir wollten den parallelen Zeitbedarf für die Ermittlung eines besten Zuges im Schachspiel bestimmen. Welche Größenordnung ist zu erwarten in der Anzahl n der Züge, die man vorausschauend betrachtet?.

Übungsaufgabe 6. Schreibe einen Algorithmus in Haskell, der den Schnitt von zwei Mengen berechnet, die als balancierte Suchbäume gegeben sind, so daß die parallele Anzahl der Reduktionen von der Größenordnung $O(\texttt{log}(n))$ ist.

Falls der Algorithmus keine balancierten Suchbäume erzeugt: Ändere den Algorithmus so ab, daß das Resultat ein balancierter Suchbaum ist.

Übungsaufgabe 7. Beschreibe einen Algorithmus zur parallelen Addition von binären Zahlen, die als beliebig lange Listen von Elementen aus $\{0, 1\}$ gegeben sind. (Hinweis: Divide und Conquer; starte Berechnung mit und ohne initialen Übertrag)

Bemerkung. Teile-und Herrsche ist offenbar ein Verfahren, bei dem man recht gute parallele Algorithmen erhalten kann. Die Algorithmen, die zeit-linear sind und eine Liste mehrfach durchlaufen, erweisen sich oft als ungeeignet zur Parallelisierung.

Bisher haben wir keine Rücksicht darauf genommen, ob die Transformationen auch wirklich notwendig waren, d.h. wir haben ohne Rücksicht auf Verluste alles parallel ausgewertet. Diese Methode heißt auch *spekulative Parallelisierung.* Diese Methode erweist sich leider in praktischen Versuchen als hoffnungslos, da i.a. die Anzahl der nutzlosen Transformationen nach kurzer Zeit bei weitem die der nützlichen überwiegt. Außerdem steigt der Platzbedarf oft sehr stark. D.h. die Ausdrücke blähen sich an unerwarteten Stellen stark auf. Operational ist dann meist nicht festzustellen, wo die relevanten Ausdrücke sind. Versucht man, nur solche Unterausdrücke zu transformieren, die für das Endergebnis notwendig sind, so ergibt sich das Dilemma, daß man diese Unterausdrücke nicht definitiv ermitteln kann. Man kann eine (operationale) Methode angeben, die "konservativ" parallel reduziert, indem man folgende Markierungsstrategie verwendet:

- die oberste Ebene des Ausdruck ist als transformierbar markiert
- Wenn `op s t` als zu reduzieren markiert ist, dann auch s, t, wenn `op` ein arithmetischer (eingebauter) Operator ist.
- Analog kann man verfahren mit Funktionen, von denen man zur Kompilierzeit ermitteln kann, daß die parallele Auswertung der Argumente garantiert konservativ ist
- Wenn der Ausdruck `if` b `then` s_1 `else` s_2 als reduzierbar markiert ist, und b nicht schon `True` oder `False` ist, dann markiere auch b.

Diese Markierungsstrategie kann zur Kompilierzeit vorbereitet werden und bestimmt, welche Auswertungen zur Laufzeit tatsächlich gestartet werden.

Dies ergibt eine (parallele) Transformationsstrategie, die genausoviel Reduktionen ausführt wie die normale Reihenfolge der Auswertung, aber möglicherweise nicht immer die parallele Zeit optimiert.

Dieses Verfahren ist schon nahe an einer guten Parallelisierung. Allerdings mußman für eine reale parallele Maschine beachten, daß diese meist eine feste Anzahl an Prozessoren hat. Das große praktische Problem ist die gute Verteilung der Arbeitseinheiten auf diese Prozessoren. Ein weiteres Problem sind die möglicherweise zu kleinen Arbeitseinheiten. Wenn die angeforderte Anzahl an Auswertungen die der Prozessoren übersteigt, dann ist es schwer, automatisch die Prioritäten so zu setzen, daß die wichtigen Reduktionen zuerst kommen. Es kann sein, daß zu viele Auswertungsanforderungen die Speicherverwaltung überlasten, weil zu weit vorgegriffen wurde.

Übungsaufgabe 8. Als Beispiel betrachte man die Aufgabe, in einem großen sequentiellen Text-File die Anzahl der Vorkommen des Zeichens "e" zu ermitteln. Angenommen, der File paßt als Baum oder Feld in den Hauptspeicher. Parallel ergibt dies soviele Reduktionen, wie der File Zeichen enthält. Da erst die (parallele) Addition alles wieder aufsammelt und den Platz freigibt, kann es im schlimmsten Fall dazu kommen, daß die Auswertung stecken bleibt, weil kein Platz mehr vorhanden ist.

Mischt man parallele und sequentielle Reduktion, dann hat man, praktisch gesehen, eher eine Chance mit weniger Platzanforderung die Rechnung zu Ende zu bringen und die mögliche Parallelität zu nutzen.

Betrachtet man andere Programmiersprachen, so ist diese Parallelisierung, die wir betrachtet haben, eine implizite Parallelisierung, d.h analog zu einer vom Compiler berechneten parallelisierten Ausführung. Da hier ein praktisch befriedigendes Verhalten nur selten erreicht wird, erlaubt man i.a. *explizite Parallelisierung*, d.h es gibt explizit parallele Befehle. Z.B.: Führe eine Operation auf allen Elementen eines Feldes aus (paralleles `map`). Das parallele Skalarprodukt: $\sum_{i=1}^{n} a_i * b_i$ ist oft eine parallelisierte Basisfunktion.

Bemerkung. Das praktische Problem der parallelen Algorithmen ist, daß im Moment die technische Entwicklung der Beschleunigung der sequentiellen Rechner nach einer gewissen Zeit die parallelen Architekturen wieder einholt.

Da die Architekturen für parallele Rechner sehr unterschiedlich sind, ist es i.a. sehr aufwendig, Anwendungsprogramme zu entwickeln, die den Zeitvorteil nutzen können.

D.h. Parallelisierung und/oder verteilte Berechnung eines einzelnen Problems lohnt sich nach heutigem Stand der Technik nur in wenigen Fällen:

- Die Aufgabe ist von Hand in viele gleichartige größere Stücke zu zerlegen und kann auf andere Rechner verteilt werden: Z.B. Faktorzerlegung einer Primzahl. (grobkörniger Parallelismus)
- Die Parallelität ist im Programm unsichtbar oder sehr kontrolliert einsetzbar und wird von der Maschine / dem Kompiler hinzugefügt.
- Eine wichtige Anwendung mit vielen gleichartigen kleinen Berechnungen lohnt den Aufwand, massiv parallele Systeme einzusetzen. (Simulationen, Wetter, ...) (sogenannter feinkörniger Parallelismus)

Diese Situation könnte sich in Zukunft wieder ändern.

4.6 Programmierparadigmen

In diesem Kapitel sollen einige wichtige Programmierparadigmen und zugehörige Programmiersprachen bzw. typische Konstrukte und Modellvorstellungen besprochen und gegenübergestellt werden.

Wir werden das im folgenden anhand von typischen beschreibenden Stichworten machen. Diese beschreibenden Eigenschaften sind nicht immer disjunkt, sondern setzen oft nur andere Schwerpunkte.

4.6.1 Imperative P., Prozedurale P.

Hierunter versteht man Programmiersprachen bzw. einen Programmierstil, der sich im wesentlichen an einer festen Ausführungsreihenfolge (Auswertungsreihenfolge) orientiert. Die Vorstellung, die dahinter steckt, ist die eines Hauptspeichers, der schrittweise manipuliert wird. D.h. das Programm besteht aus einer modularen Zusammensetzung von kleinsten Zustandsübergängen, den Befehlen. D.h. im wesentlichen die von-Neumann-Architektur:

Der Gesamtzustand ist zusammengesetzt aus Speicherinhalt und aktuellem Befehl. Die einzelnen Befehle eines Programm können sein: Im Hauptspeicher nachschauen (Variable referenzieren), im Hauptspeicher etwas ändern (Zuweisung), kleinere lokale Operationen durchführen (addieren, Ausdrücke auswerten), aufgrund von Bedingungen den nächsten auszuführenden Befehl bestimmen (nächster Befehl, oder bedingte Anweisung: `if-then-else, branch`, oder unbedingter Sprungbefehl: `go-to, jump`, Schleifenkonstrukte: `while, for`). Auch Prozeduraufrufe und Unterprogrammaufrufe sind i.a. Standard, wobei der Hauptspeicher dann nicht mehr als flache Struktur, sondern hierarchisch (durch die Aufrufe) strukturiert ist.

Modernere Versionen haben ein Modulsystem, ein Typsystem, z.t. objektorientiert. Programmiersprachen, die man hier einordnen kann: Pascal, C, C^{++}, Fortran, Algol, Python, ...

4.6.2 funktionale

Die funktionalen Programmiersprachen benutzen als wesentliches Basiskonstrukt die Funktionsanwendung (analog zum Prozeduraufruf). Es gibt verschiedene Ausprägungen dieser Sprachen:

Lisp-Familie : ist eine Kombination von imperativ und funktional. Es gibt Prozeduren höherer Ordnung. Auswertung von Ausdrücken und Rekursion sind die vorherrschenden Strukturierungsmethoden bei Algorithmen. Es ist aber auch erlaubt, Variablen direkt zu verändern. Dies kann nur funktionieren, wenn die Auswertungsreihenfolge der Ausdrücke beim Programmieren bekannt ist. Z.B. ist es hier möglich, in Unterausdrücken s, t des Ausdrucks $(s+t)$ Zuweisungen zu haben, die genau zum Zeitpunkt der Ausführung wirksam werden. Wenn nicht klar festgelegt ist, daß zuerst s, dann t, dann die Addition ausgeführt wird, dann wären die Ergebnisse nicht mehr eindeutig definiert.

Programmiersprachen: Lisp, Scheme, ML, Caml,

rein funktionale Diese haben wir in dieser Vorlesung zur Erklärung von Algorithmen und zum Programmieren verwendet. Auswertung von Ausdrücken und Rekursion sind die Strukturierungsmethoden. Es wird auf explizite Speicherveränderungen (Zuweisungen) verzichtet. Dadurch hat man eine freie Wahl bei der Reihenfolge der Auswertung von Ausdrücken, was einerseits die Verifikation erleichtert, die Korrektheit vieler Programmtransformationen offensichtlich macht, und andererseits auch korrekte Parallelisierung leicht erklärbar macht. In einer sequentiellen Implementierung ist die Reihenfolge auf sogenannte verzögerte Reduktion festgelegt

Modernere funktionale Programmiersprachen sind mit einem komplexen Typsystem ausgestattet.

Programmiersprachen: Haskell, Miranda, Lazy-ML

Eine wichtige Programmiermethodik ist das *monadische Programmieren*, das im Zusammenhang mit Ein/Ausgabe benötigt wird, bzw. bei Zustandsübergängen und auch bei der Simulation imperativer Programmiermethodik.

Die syntaktische Konvention dafür ist die do-Notation:

$$\text{do } \{x_1 \leftarrow e_1; \ldots; x_n \leftarrow e_n; \text{return } e\}$$

Dies kann man so deuten, daß zunächst e_1 berechnet wird und der Wert der Variablen x_1 mittels eines `let` zugewiesen wird. Danach dasselbe für e_2 und x_2. Hierbei stehen die Werte der weiter links definierten Variablen jeweils rechts zur Verfügung. Am Ende wird mit `return` ein Wert bereitgestellt. Diese Berechnung ermöglicht eine sequentielle Verarbeitung, die auch für Ein/Ausgabe benutzt wird. Sie erlaubt auch eine Optimierung des Speicherplatzes z.B. bei Arrayverarbeitung.

logische P. Die logischen Programmiersprachen beruhen auf der Prädikatenlogik. Benutzt wird eine eingeschränkte Klasse von Formeln, sogenannte Horn-formeln bzw. Horn-klauseln. In der Notation verzichtet man auf Quantoren, denn man benötigt nur All-quantoren auf oberster Ebene der Formel.

Horn-Klauseln sind im wesentlichen Wenn-dann- Aussagen, wobei man als Fakten auch Aussagen ohne Bedingungen haben kann. Als Dann-Aussagen sind keine Oder-Aussagen zugelassen. D.h. eine Hornklausel ist von der Form:

$\forall x_1 \ldots x_n . \neg P_1(t_{11}, \ldots, t_{1,n_1}) \vee \ldots \vee \neg P_m(t_{m1}, \ldots, t_{m,n_m}) \vee P_{m+1}(t_{(m+1)1}, \ldots, t_{m+1,n_{m+1}})$

oder umgeschrieben als: $\forall x_1 \ldots x_n . (P_1(t_{11}, \ldots, t_{1,n_1}) \wedge \ldots \wedge P_m(t_{m1}, \ldots, t_{m,n_m})) \Rightarrow P_{m+1}(t_{(m+1)1}, \ldots, t_{m+1,n_{m+1}})$

nichtdegenerierte Hornklauseln nennt man auch *Regeln.* Es ist erlaubt, Klauseln ohne Bedingung zu haben: Das sind sogenannte *Fakten.* Klauseln ohne Konklusion, d.h. nur mit negierten Literalen, sind sogenannte *Anfragen.* Das liegt daran, daß man nach Umformulierung eine Existenzaussage erhält:

$\neg \exists x_1 \ldots x_n . P_1(t_{11}, \ldots, t_{1,n_1}) \wedge \ldots \wedge P_m(t_{m1}, \ldots, t_{m,n_m})$.

Das Negationszeichen stört etwas. Dies ist aber logisch korrekt, denn normalerweise stellt man die Anfrage ohne Negationszeichen:

$\exists x_1 \ldots x_n . P_1(t_{11}, \ldots, t_{1,n_1}) \wedge \ldots \wedge P_m(t_{m1}, \ldots, t_{m,n_m})$.

Die Auswertung eines logischen Programms besteht darin, diese Antworten zu konstruieren, d.h. Werte für diese existierenden $x_1, \ldots, x_n$ zu finden. Operational ist dies am einfachsten, wenn man die Negation der Anfrage dem Programm hinzufügt und dann versucht, einen Widerspruch zu finden.

Die Berechnung erfolgt durch mehrfaches Anwenden von Implikationen, wobei diese Berechnung durch eine Anfrage an das Programm gesteuert wird.

Beispiel 12. Als Beispiel die Berechnung der Vorfahr-Relation aus der Vater und Mutter-Relation: Das Programm kann so aussehen:

```
Vater(heinrich, paula)
Mutter(paula, sabine)
Vater(x,y) --->  Vorfahr(x,y)
Mutter(x,y) ---> Vorfahr(x,y)
Vorfahr(x,y), Vorfahr(y,z)  ---> Vorfahr(x,z)
```

mögliche Anfragen:

```
Vorfahr(heinrich,sabine)
>  ja
```

weitere Anfrage:

```
Vorfahr(X,Y)
> X = heinrich, Y = paula
> X = paula, Y = sabine
> X = heinrich, Y = sabine
```

Die bekannteste Programmiersprache dazu ist Prolog in verschiedenen Varianten. Allerdings ist der reine Ansatz durch verschiedene Konstrukte verwässert worden, so daß die üblichen Implementierungen sich wie eine imperative Programmiersprache verwenden lassen. So kann man z.B. das Programm verändern, indem man Klauseln während des Ablaufs hinzufügt oder löscht. Oder man kann die Strategie beeinflussen, die die Implikationen berechnet. Die Semantik des Programms weicht dann immer mehr von der eigentlichen "logischen Semantik" ab.

Als weiteres Beispiel die Bearbeitung von Listen in Prolog. Listen stellen wir dar mit cons(.,.) und nil.

Die append-Funktion, die zwei Listen zusammenhaengt hat folgendes Programm, wobei append(X,Y,Z) ungefähr bedeuten soll: bei Eingabe zweier Listen in X, Y soll in Z die Liste $X ++Y$ ausgegeben werden.

append(nil, X,X).
append(cons(U,V),Y,Z) $\Rightarrow$ append(V,Y,cons U Z)

Eine zugehörige Anfrage ist: append(cons(1, cons(2,nil)), cons(3, cons(4,nil)), Z) ?

Als Antwort würde man erhalten: Z = cons(1, cons (2, cons(3, cons(4,nil)))).

Interessanterweise kann man diese Programme auch rückwärts verwenden:

append(X,Y, cons(1, cons (2, cons(3, cons(4,nil)))))

Ergibt alle Möglichkeiten für Listen X,Y die zusammengehängt die Liste $[1, 2, 3, 4]$ ergeben. Übrigens sind Listen i.a. in Prolog syntaktisch schon vorgesehen.

Es gibt Erweiterungen wie Constraint-logische Programmiersprachen, die das Auflösen von verschiedenartigen Bedingungen und Gleichungen integriert haben.

Eine Einschränkung der logischen Programmierung, die in deduktiven Datenbanken Verwendung findet, ist *Datalog*. Die Einschränkung ist, daß als Terme in den Prädikaten nur Variable erlaubt sind. Das obige Vorfahr-Beispiel ist ein Datalog-Beispiel. Damit kann man einfache relationale und deduktive Datenbanken beschreiben, und hat auch gleich einen Abfragemechanismus. Datalog ist allerdings keine vollständige Programmiersprache: alle Programme terminieren, bzw. haben nur eine endliche Menge von Antworten, d.h. man kann sie so auswerten lassen, daß Terminierung garantiert ist.

maschinennahe P. Diese Programmiersprachen sind dazu gedacht, maschinennah zu programmieren, d.h. sind an den Möglichkeiten der Architektur und des Prozessors orientiert. Dazu zählen Speicherveränderungen, Adressrechnung, Änderungen des Befehlsablaufs, Unterbrechungen, usw. Zu diesen Programmiersprachen zählt Assembler in allen Varianten, aber auch C auf einer etwas höheren Abstraktionsstufe. Diese Programmiersprachen erlauben im Prinzip Algorithmen sehr effizient zu programmieren. Allerdings kommt man immer mehr davon ab, direkt in diesen Programmiersprachen zu programmieren (außer für sehr spezielle Anwendungen): Die Gründe sind einleuchtend einfach: Man muß sehr viel und sehr exakt programmieren, die Programme sind fehleranfällig, und zudem oft prozessorspezifisch. Die Prozessoren ändern sich in relativ kurzen Zeitabständen, so daß Programme altern. Der Wartungsaufwand ist hoch. Außerdem sind Übersetzungsprogramme (Kompiler) verfügbar, die Programme einer etwas höheren Abstraktionsstufe jeweils in effizienten Maschinenkode übersetzen können (Z.T. in effizienteren als von Hand geschriebenen). Ein Kompiler an eine neuen Prozessor anzupassen und danach Programme zu kompilieren ist viel einfacher als alle Programme von Hand zu ändern.

regelbasierte P. Programme bestehen im wesentlichen aus Regeln der Form:

`Wenn` Bedingung `Dann` Aktion

Damit kann man Problemlösungen modellieren, die auf solchen Regeln basieren. Diagnostische Expertensystem wurden z.T. implementiert aufgrund unter Benutzung regelbasierter P.

Welche Ausdruckskraft eine solche regelbasierte P. hat, hängt von der Form der Bedingungen und der erlaubten Aktionen ab. Die einfachste

Form ist: es gibt nur Bezeichner für einfache Datentypen: Zahlen, Strings, Wahrheitswerte. Diese sind auf einer Tafel (blackboard) notiert. Aktionen können neue Werte in die Tafel eintragen. Typische Regel:

Wenn Körpertemparatur $> 37°$ Dann Fieber

Wenn Fieber und Husten Dann Erkältung

Eine einfache Form von regelbasierten Programmen kann man in Prolog simulieren, wenn man als einziges nichtpures Konstrukt erlaubt, daß Werte auf der Tafel (als Fakten) geändert werden können.

objektorientierte P. Diese beinhalten ein Reihe von Konzepten, die auch in anderen Programmiersprachen vorkommen. Die Besonderheit ist, alle Daten als Objekte aufzufassen. Dies entspricht im wesentlichen Datentypen, deren Implementierung versteckt ist (bzw. versteckt werden kann). Jedes Objekt gehört zu einem Typ, bzw. einer Klasse, da Typen zu Klassen erweitert sind, wobei für eine Klasse i.a. Merkmale, d.h. sowohl Attribute (Werte) als auch Methoden (Prozeduren), definiert werden. Typisch ist die Klassenhierarchie, die die Klassen in einer Hierarchie einordnet. Die Kommunikation zwischen den Objekten erfolgt durch das Senden von Nachrichten, wobei dies auch als Prozeduraufruf bzw. Funktionsaufruf gesehen werden kann.

Ein Strukturierungshilfsmittel ist die *Vererbung*, die es erlaubt, Zugriffsmethoden auf allgemeinen Objekten nur einmal zu definieren, und diese dann für Objekte von Unterklassen zu verwenden. Klasseninformationen sind i.a. zur Laufzeit noch vorhanden. Die vorherrschenden objektorientierten P. haben i.a. als Ausführungsmodell das von-Neumann-Modell: es gibt einen Speicher aller Objekte, der Inhalt von Objekten kann sich ändern, und es gibt (lokal) eine feste Sequenz der Verarbeitung.

Als Beispiel betrachte man die Strukturierung einer Datenbank

Oberklasse: any	
Attribute:	angelegt am, geaendert am, von-wem
Methoden:	erzeuge , loesche
Unterklasse Kunde	
Attribute:	Name, Adresse, Kundennr
Methoden:	lese Namen, Adresse, lese Kundennr
	ändere Kundennr, ...

Beispiele sind: Eiffel, Modula, C++, Python, Java

deklarative P. Hier liegt die Betonung darauf, daß man sagt, *WAS* man haben will im Gegensatz dazu, *WIE* es berechnet werden soll. Es gibt allerdings unterschiedliche Auffassungen darüber, was als deklarativ gelten soll, was nicht:

Rein funktionale Bechreibungen eines Problems kann man als deklarativ ansehen, auch wenn, wie wir gesehen haben, auch diese Beschreibung gewisse Abhängigkeiten der Berechnungen implizieren, und verschiedene Beschreibungen auch eine andere Ausführung erzwingen.

Eine Definition eines Problems in einer logischen Programmiersprache kann man ebenfalls als deklarativ ansehen, solange das Programm in purem Prolog geschrieben ist (d.h. als Hornklauseln vorliegt).

Datenbankabfragesprachen Werden oft als eigenständige Programmiersprachen betrachtet, auch wenn diese i.a. nicht das volle Repertoire an Konstrukten haben. Diese sind i.a. auch nicht äquivalent zu Programmiersprachen, da Anfragen terminieren. Standardbeispiel ist SQL und Varianten davon. Das zugrundeliegende Modell ist eine Menge von vorgegebenen Relationen, die die Datenbank darstellen. Die Anfragesprache dient dazu, eine weitere Relation zu konstruieren aufgrund der Anfrage und der Datenbank:

Eine mögliche Anfrage ist:
"Wähle alle Studenten aus, deren Studienzeit größer ist als 20 Semester"

SQL selbst ist noch erweitert um Funktionen (Summationsfunktionen,...), die innerhalb von Anfragen verwendet werden dürfen. Man kann den Anfrageteil von SQL weitgehend als logisches Programm sehen. Allerdings gibt es Spezialitäten (Verwendung von gleichen Variablennamen in SQL-Anfragen), die diese Analogie unübersichtlich machen.

Außerdem gibt es einen Teil von SQL, der die Datenbankmanipulation betrifft.

4.6.3 Bemerkungen zur Äquivalenz von Programmiersprachen

Zwei Programmiersprachen L_1 und L_2 sind *berechnungsäquivalent*, wenn sie (prinzipiell) die gleichen Funktionen berechnen können. Genauer:

Die Programmiersprachen L_1 ist zu L_2 berechnungsäquivalent, wenn es eine Übersetzungsfunktion ϕ gibt, die Programme der Programmiersprache L_1 in solche der Programmiersprache L_2 übersetzt, so daß zu jedem Programm P_1 und jeder Eingabe e:
P_1 berechnet bei Eingabe e den Wert a genau dann wenn $\phi(P_1)$ berechnet bei Eingabe $\phi(e)$ den Wert $\phi(a)$.

Dies beschreibt die Programme aufgrund ihrer Eigenschaft, Funktionen zu berechnen. Die folgenden Programmiersprachen sind alle äquivalent in diesem Sinne: Pascal, Haskell, Assembler, Lambda-Kalkül, Turingmaschine.

Daß dies der allgemeinste Begriff für Berechenbarkeit ist, ist die *Churchsche These*.

Die Probleme, die man mit Programmen lösen kann, nennt man auch (Turing-) *berechenbare Probleme*

Es gibt eingeschränkte Programmiersprachen, die nicht äquivalent zur Turingmaschine sind:

Sei L eine Programmiersprache und $P(L)$ seien die zugehörigen Programme. Dann ist L nicht berechnungsäquivalent zur Turingmaschine, wenn eine der folgenden Bedingungen erfüllt ist:

- Alle Programme in $P(L)$ terminieren bei allen Eingaben .
- Es gibt eine Funktion f, so daßfür jedes Programm $Q \in P(L)$ der Speicherplatz beschränkt ist durch $f(Q)$.

Z.B. ist dann die Menge der Cobol-Programme, die kein Unterprogramm aufrufen dürfen, nicht äquivalent zur Turingmaschine.

Die Menge der regelbasierten Programme ist nicht äquivalent zur Turingmaschine, wenn die Größe der Zahlen und Strings beschränkt ist. Wenn allerdings die Zahlen unbeschränkt groß sind und arithmetische Operationen erlaubt sind, dann sind die zugehörigen Programme äquivalent zur Turingmaschine,

Die Probleme, zu denen es (für alle Eingaben) terminierende Programme gibt, nennt man *entscheidbare Probleme*.

Aber: Es gibt nicht entscheidbare Probleme (unentscheidbare)

Das *Halteproblem*: Gegeben ein Programm und eine Eingabe, bestimme ob die Berechnung anhält. Hierzu gibt es kein Programm, das für alle Eingaben terminiert und dann ja oder nein sagt.

Aber, die Frage: "Gegeben e: Hält P an¿' ist berechenbar: Wenn die Antwort ja ist, dann kann man diese Antwort auch finden

Ein nichtberechenbares Problem ist die Frage:

Gegeben P und e: Hält die Berechnung nicht an?

Die Praktische Äquivalenz von Programmiersprachen betrifft die Frage: Kann man Probleme gleich gut in der Programmiersprache L_1 und L_2 programmieren:

Dies ist i.a. falsch. Einerseits gibt es theoretische Unterschiede im Ressourcenverbrauch: Z.B. hat die Turingmaschine keinen adressierbaren Speicher, und hat damit lineare statt logarithmischen Zugriffszeit (in der Größe des verwendeten Speicherplatzes) auf Daten. Deshalb sind bestimmte Probleme als Turingmaschinenprogramme prinzipiell langsamer als z.B. in Modula oder Haskell, die einen direkt adressierbaren bzw. einen baumförmig organisierten Speicher haben.

Pragmatisch gesehen, hat jede Programmiersprache ihre guten Seiten (Spezialitäten) und ihre schlechten Seiten. I.a. gilt: abhängig von den organisatorischen und anderen Rahmenbedingungen und von der Problemklasse kann und sollte man die Wahl der Programmiersprache abhängig machen.

4.7 Typisierung in Programmiersprachen

In diesem extra Kapitel sind die Typisierungsmethoden in Haskell beschrieben wobei auch auf Methoden anderer Programmiersprachen eingegangen wird.

In Haskell gilt, daß jeder gültige Ausdruck einen Typ haben muß. Außerdem sorgt der Typechecker dafür, daß keine Typfehler zur Laufzeit auftreten. Dies nennt man: *strenge, statische Typisierung*

Das Gegenteil ist *dynamische Typisierung*, die eine Typüberprüfung zur Laufzeit macht. D.h. es gibt ein Typsystem, alle Datenobjekte haben einen Typ (meist durch den implementierten Aufbau der Daten gegeben). Bei falschem Aufruf wird ein Laufzeit Typfehler gemeldet.

Die *schwache, statische Typisierung* hat ein Typcheck zur Kompilierzeit, aber es sind Löcher gelassen für den Programmierer, so daß in bestimmten Fällen doch ein Typfehler zur Laufzeit auftritt.

Monomorphe Typisierung erzwingt, daß alle Funktionen einen Typ haben, d.h. Typvariablen sind verboten bzw. es gibt keine Allquantoren in den Typen. In Haskell würde das bedeutet, daß man für Listen von Zahlen und Strings (Listen von Zeichen) verschiedene Längenfunktionen braucht.

Polymorphe Typisierung erlaubt es Funktionen zu schreiben, die auf mehreren Typ arbeiten können, wobei man schematische Typen meint. In Haskell sind die Typvariablen verantwortlich für die Schematisierung. Man nennt dies auch *parametrischen Polymorphismus.*

Die weitere Strukturierung der Typen in Typklassen ignorieren wir zunächst mal. Wir geben die Syntax von Typen in Haskell an:

Definition 9.

$$
\begin{array}{lll}
\langle\text{Typ}\rangle & ::= & \langle\text{Basistyp}\rangle \mid (\langle\text{Typ}\rangle) \mid langle\text{Typvariable}\rangle \\
 & & \mid \langle\text{Typkonstruktor}_n\rangle\{\langle\text{Typ}\rangle\}^n \\
 & & \text{(n ist die Stelligkeit)} \\
 & & \mid \langle\text{Typ}\rangle \to \langle\text{Typ}\rangle \\
\langle\text{Basisyp}\rangle & ::= & \text{Int} \mid \text{Integer} \mid \text{Float} \mid \text{Rational} \mid \text{Char}
\end{array}
$$

Typen mit $\to$ sind sogenannte Funktionstypen. Die Konvention ist, daß $a \to (b \to c)$ dasselbe meint wie $a \to b \to c$

Beispiele sind:

- $\texttt{length} :: [\alpha] \to \texttt{Int}$
- $(:) :: \alpha \to [\alpha] \to [\alpha]$

Will man das als Formel ausdrücken, so könnte man schreiben: $\forall[\alpha].\texttt{length} :: [\alpha] \to \texttt{Int}$

Da das eine Aussage über Argumente und Resultate ist, sollte folgendes gelten: $\forall x.x :: [\alpha] \Rightarrow \texttt{lengthx} :: \texttt{Int}$.

Eine der Typregeln, die stets einzuhalten ist, betrifft die Anwendung von Funktionen:

$$s :: \sigma \to \tau, t :: \sigma \Rightarrow (s\ t) :: \tau$$

Da die Haskelltypen aber Typvariablen enthalten, formulieren wir diese Regel etwas allgemeiner, benötigen dazu aber den Begriff der Instanz eines Typs. Wenn λ eine Funktion ist, die Typen für Typvariablen einsetzt, dann ist λ eine Typsubstitution.

Z.B. kann man mit $\lambda = \{\alpha \mapsto \texttt{Char}, \beta \mapsto \mathit{Float}\}$ die Typinstanz $\lambda([\alpha] \to \texttt{Int}) = [\texttt{Char}] \to \texttt{Int})$ bilden.

Die Regel für die Anwendung lautet dann:

$$s : \sigma \to \tau, t : .\rho \text{ und } \lambda(\sigma) = \lambda(\rho) \Rightarrow (s\ t) :: \lambda(\tau)$$

Beispiel 13. Der Typ von [1] kann folgendermaßen ermittelt werden: $[1] = 1 : []$
$1 :: \texttt{Int}$ und $[] :: [\beta]$ folgt aus den Typen als Konstanten.
$(:) :: \alpha \to [\alpha] \to [\alpha]$
Anwendung der Regel mit $\lambda = \{\alpha = \texttt{Int}\}$ ergibt:
$(1 :) :: [\texttt{Int}] \to [\texttt{Int}]$
Nochmalige Anwendung der Regel mit $\lambda = \{\beta = \texttt{Int}\}$ ergibt:
$(1 : []) :: [\texttt{Int}]$

Beispiel 14. Wir weisen nach, daß es keinen Typ von $[1,'a']$ gibt: Der Ausdruck ist $1 : ('c' : [])$
$1 :: \texttt{Int}$, $[] :: [\beta]$ und $'a' :: \texttt{Char}$ folgt aus den Typen als Konstanten.
Wie oben ermittelt man: $(1 :) :: [\texttt{Int}] \to [\texttt{Int}]$

Wenn wir jetzt die Typregel anwenden wollen, dann stellen wir fest: es gibt kein λ, das `Int` und `Char` gleichmacht. D.h. die Regel ist nicht anwendbar.

Da das auch der allgemeinste Versuch war, einen Typ für diese Liste zu finden, haben wir nachgewiesen, daß die Liste $[1,'a']$ keinen Typ hat.

Etwas komplexer ist die Typisierung von definierten Funktionen, insbesondere von rekursiv definierten. Man benötigt auch weitere Regeln für Lambdaausdrücke, `let`-Ausdrücke und List Komprehensionen.

In Haskell, ML wird im wesentlichen der sogenannte Typpr´ufalgorithmus von Robin Milner verwendet. Dieser ist i.a. schnell, aber hat eine sehr schlechte Komplexität: in seltenen Fällen hat er exponentiellen Platzbedarf. Dieser Algorithmus liefert auch nicht immer die allgemeinsten Typen.

4.7.1 Andere Programmiersprachen

In den Programmiersprachen Lisp und Scheme gibt es in den Standardvarianten kein statisches Typsystem. Der Grund ist, daß von Anfang keine statische Typdisziplin eingehalten wurde: Z.B. wurde es erlaubt und auch genutzt, daß man alle Werte als Boolesche Werte verwendet durfte: Die Konvention war: Nil = False, alles andere gilt als True. Z.B. ist der Wert von $1 = 0$ in Lisp die Konstante Nil, während $1 = 1$ die Konstante 1 ergibt. D.h. um einen vernünftigen Typisierungsalgorithmus für diese Sprachen zu konzipieren, müßte man zu viel ändern.

Programmiersprachen, die keine Funktionen oder Prozeduren höherer Ordnung haben, begnügen sich meist mit einem monomorphen Typcheck.

Bei Verwendung der arithmetischen Operatoren z.B. des Additionszeichens (+) will man oft erreichen, daß dies für alle Zahlentypen funktioniert. Hier wird teilweise eine Typkonversion vom Kompiler eingefügt, wenn unverträgliche Operanden benutzt werden. In Haskell muß man diese von Hand einfügen, so daß nicht die Gefahr besteht, daß ungewollte Typkonversionen eingefügt werden.