Programmierung und Modellierung

Methodisches Programmieren:
Konstruktion effizienter Programme

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

SS 2009

Inhalt

- Kap. 8 Konstruktion effizienter Programme
  1. Effiziente Datenstrukturen und Algorithmen
  2. Effizienz der Rekursion

8. Konstruktion effizienter Programme

- Wie kommt man zu effizienten Programmen?
  - Effiziente Algorithmen
  - Effiziente Datenstrukturen
  - Einbeziehung des Auswerteprozesses,

    z.B. günstige Rekursionsschemata, Vermeidung von “append”.
- Der dritte Punkt lässt sich relativ schematisch umsetzen und wird mehr und mehr durch Fortschritte in der Compilertechnik automatisiert.
- Die ersten beiden Punkte erfordern Kreativität, Fachwissen und Erfahrung.

8.1 Effiziente Datenstrukturen

- Oft liegt der Schlüssel zu einem effizienten Algorithmus in der Wahl der geeigneten Datenstruktur.
- Beispiel: Binäre Suchbäume (BST) aus der letzten Vorlesung.
  - Suchen eines Elementes in beliebigem (ungeordneten) Baum `t` erfordert Zeit O(n), wobei n = Anzahl_Knoten(`t`).
  - Suchen eines Elementes im Binären Suchbaum `t` erfordert Zeit O(Höhe(`t`)), d.h., falls der Suchbaum balanciert ist, O(log(n)).

Effiziente Algorithmen: Beispiel

- **Sortieren durch Einfügen**
  - Eine Liste [x1, ... ,xn] heißt *sortiert*, wenn x1 < x2 < … < xn.

```
fun insertEl(a, nil) = [a]
|   insertEl(a, h :: t) =
        if a <= h then a :: h :: t
        else h :: insertEl(a,t);
fun insertSort nil = nil
|   insertSort (h::l) = insertEl(h, insertSort l);
```

- Ist `l` sortiert, so auch `insertEl(a,l)` und es enthält dieselben Elemente wie `a::l`.
- Für beliebiges `l` ist `insertSort(l)` sortiert und enthält dieselben Elemente wie `l`.
- Worst case: Laufzeit von `insertEl`: O(n)
- Worst case: Laufzeit von `insertSort`: $O(n^2)$

Beispiel für effizienten Algorithmus

- Wir haben gesehen, dass Sortieren durch Einfügen quadratische Komplexität hat.
- Wir betrachten jetzt ein rekursives Verfahren, welches sowohl praktisch, als auch theoretisch (asymptotisch) bessere Laufzeit aufweist.

Sortieren durch Mischen

- Um eine Liste der Länge n zu sortieren:
  - Teile man die Liste in zwei gleichgroße Hälften
  - Sortiere jede Hälfte durch rekursiven Aufruf (bei Länge 1 ist natürlich nichts zu tun)
  - Füge die beiden sortierten Hälften “im Reißverschlussverfahren” zusammen.

In SML

```
fun split nil = (nil,nil)
| split([a]) = ([a],nil)
| split(a::b::l) =
    let val (u,v) = split l in (a::u, b::v) end;
fun merge(nil,l) = l
|   merge(l,nil) = l
|   merge(a::t, b::u) =
            if a <= b then a :: merge(t, b::u)
            else b :: merge(a::t, u);
fun mergeSort nil = nil
| mergeSort([a]) = [a]
| mergeSort l =
    let val (u,v) = split l
        val u1 = mergeSort u
        val v1 = mergeSort v in
    merge(u1, v1) end;
```

Laufzeitkomplexität von mergesort

Seien n, n1, n2 die Längen der Listen `l, l1, l2.`

- Worst Case Laufzeit von `split l`: O(n)
- Worst Case Laufzeit von `merge(l1,l2)`: O(n1+n2)
- Worst Case Laufzeit von `mergeSort(l)`: O(n*log(n))
  - Informelle Begründung
    - Bei jedem Aufruf von `mergeSort` werden `split` und `merge` ausgeführt mit Laufzeit O(2*n) = O(n).
    - Wegen der Halbierung der Liste bei jedem Aufruf von `mergeSort` benötigt man log n Runden (mit jeweils parallelen Aufrufen von `mergeSort` über disjunkten Teilen der Liste bis der Abbruchfall erreicht ist.

8.2 Effizienz der Rekursion

- Wir untersuchen exemplarisch die Effizienz rekursiver Funktionsdefinitionen:
  - Eine bestimmte Art der Rekursion (“endständige Rekursion”) ist besonders platzeffizient.
  - Es ist manchmal möglich, rekursive Funktionen in endständig rekursive Form zu bringen.
  - Es gibt Formen der Rekursion, für die solch eine Transformation nur unter Aufbietung zusätzlichen Hilfsspeichers in derselben Größe wie die resultierende Ersparnis möglich ist.

Rekursion im Rechner

- Die **Auswertung rekursiver Funktionen** erfolgt im Rechner näherungsweise wie im **Substitutionsmodell**, d.h. durch die sukzessive Ersetzung von Funktionen durch ihre Definition.
- **Beispiel: Paritätsfunktion**
  - `fun gerade(n) =`
    `if n = 0 then true else not(gerade(n-1));`
  - Auswertung

    `gerade(4) =`
    `not``(gerade(3)) =`
    `not``(`**`not`**`(gerade(2))) =`
    `not``(`**`not`**`(`**`not`**`(gerade(1)))) =`
    `not``(`**`not`**`(`**`not`**`(`**`not`**`(gerade(0))))) = `**`true`**
  - Platzverbrauch von `gerade(n)` ist O(n), aber **nicht** O(1).

Endständig rekursive Einbettung der Parität

- ```
  fun gerade2(n) = gerade2_aux(n, true);
  fun gerade2_aux(n, a) =
          if n = 0 then a
          else gerade2_aux(n-1, not(a));
  ```
- **Auswertung**

  ```
  gerade2(4) = gerade2_aux(4, true) =
  gerade2_ aux(3, not(true)) =
  gerade2_aux(3, false) =
  gerade2_aux(2, true) =
  gerade2_aux(1, false) =
  gerade2_aux(0, true) = true
  ```
- **Platzverbrauch** von `gerade2(n)` = O(1).
  ```

Zusammenhang der beiden Funktionen

- Für alle natürlichen Zahlen n und für alle Wahrheitswerte a gilt:

```
gerade2_aux(n, a) =
    if a then gerade(n) else not(gerade(n))
```

- **Beweis** durch vollständige Induktion über n.

Beispiel: Fakultät

- ```
  fun fak(n) =
    if n=0 then 1
    else n * fak(n-1);
  ```
- Auswertúng

  ```
  fak(4) =
  4 * fak(3)) =
  4 * (3 * fak(2)) =
  4 * (3 * (2 * fak(1))) =
  4 * (3 * (2 * (1 * fak(0)))) =
  4 * (3 * (2 * (1 * 1))) = 24
  ```
- Der “Computer” kann nicht wissen, dass man die Multiplikationen auch umklammern und vereinfachen kann: die gesamte Multiplikationsaufgabe bleibt stehen, bis endlich `fak(0)` ausgewertet wird.
- Platzverbrauch von `fak(n)` ist O(n), aber nicht O(1).
  ```

Endständig rekursive Einbettung der Fakultät

- ```
  fun fak2(n) = fak2_aux(n, 1);
  fun fak2_aux(n, res) =
      if n = 0 then res
      else fak2_aux(n-1, res*n);
  ```
- Auswertung

  ```
  fak2(4) = fak2_aux(4, 1) =
  fak2_aux(3, 1*4) =
  fak2_aux(3, 4) =
  fak2_aux(2, 12) =
  fak2_aux(1, 24) =
  Fak2_aux(0, 24) = 24
  ```
- Platzverbrauch hier O(1).
  ```

Zusammenhang der beiden Funktionen

- Für alle natürlichen Zahlen n und `res` gilt:

```
fak2_aux(n, res) = res * fak(n)
```

- **Beweis** durch vollständige Induktion.

Lineare und endständige Rekursion: Wiederholung

- Eine rekursive Funktionsdeklaration `f(x)= E(f,x)` heißt **linear rekursiv**, wenn
  - in jedem Zweig einer Fallunterscheidung des Rumpfes `E(f,x)` höchstens ein rekursiver Aufruf `f(y)` von `f` vorkommt.
- Sie heißt **endständig rekursiv (tail recursive, auch repetitiv rekursiv)**, wenn
  - `f` linear rekursiv ist und
  - jede Fallunterscheidung mit rekursivem Aufruf die Form `f(G)` hat; d.h. dass f das äußerste Funktionszeichen der Fallunterscheidung ist.
- **Merke:**
  - Bei der Auswertung endständig rekursiver Funktionen entsteht kein zusätzlicher Platzbedarf für die Verwaltung der Rekursion.
  - Sehr häufig lassen sich **linear rekursive** Definitionen **durch Einbettung in endständig rekursive Form** bringen. Daraus resultiert dann eine beträchtliche Platz- und Zeitersparnis.

Endständige Rekursion und Iteration

- Eine endständig rekursive Funktionsdefinition der Form

  `f(x:typ1):typ2 = if p(x) then a(x) else f(b(x))`

  (p beliebiges Prädikat, a, b beliebige Funktionen) kann man auch "imperativ" wie folgt auswerten:

  1. Schreibe die Eingabe in die Speicherstelle `x`
  2. Solange `p` auf den Inhalt von `x` **nicht** zutrifft, wiederhole folgendes:
     1. Bestimme `y := b(Inhalt von x)`
     2. Ersetze den Inhalt von `x` durch `y`
  3. Gib als Ergebnis `a(Inhalt von x)` zurück.

- In imperativem Programmierstil inJava:

```
static typ2 f(typ1 x) {
   while(!p(x)) {
          x = b(x);
   }
   return a(x);
}
```

Vor- und Nachteile des imperativen Stils

- Imperativer Stil ist zunächst verständlicher.

- Funktionale Definition lassen sich durch Gleichungen spezifizieren und verifizieren.
- Im imperativen Stil muss man über Zustand der Variablen zu bestimmten Zeitpunkten sprechen, was komplizierter ist.
- In der Effizienz unterscheidet sich die imperative Version **nicht** von der endständig rekursiven Version.

Nichtlineare Rekursion

- Sei `fun f(x) = E(f, x)` eine rekursive Definition mit Abstiegsfunktion m. Finden im Rumpf bis zu a rekursive Aufrufe statt, so erzeugt die vollständige Auswertung von `f(x)` bis zu $a^{m(x)}$ rekursive Aufrufe.
- **Beispiel**
  - Fibonacci Fkt. `fib`, `mergeSort`, Ackermann Fkt. `ack`: `a = 2`
  - Fibonacci: `m(x) = x` => bis zu 2^x rekursive Aufrufe
  - mergeSort: `m(x) = O(log(length x))` => bis zu $2^{O(\log(\text{length } x))} = O(\text{length } x)$ rekursive Aufrufe
- **Fazit**: Bei nichtlinearer Rekursion Laufzeit im Auge behalten!

Fibonacci als endständige Rekursion

- Manchmal lassen sich auch nichtlineare Rekursionen in endständige Rekursionen umschreiben:

```
fun fib(n) =
    if n = 0 orelse n = 1
       then 1
       else fib(n-1) + fib(n-2);
```

- **Definiere**

```
fun fib_aux(n, a, b) =
    if n = 0 then a
    else fib_aux(n-1, b, a + b);
```

- **Es gilt:**

```
    fib_aux(n,fib(k),fib(k+1)) = fib(n+k).
```

- **Also**

```
fun fib(n) = fib_aux(n,1,1);
```

- Dies ist ein Spezialfall der **dynamischen Programmierung**, ein allgemeines Optimierungsschema für nichtlineare Rekursionen! (Siehe Vorlesung “Effiziente Algorithmen”).

Türme von Hanoi

- Es gibt drei senkrechte Stäbe. Auf dem ersten liegen n gelochte Scheiben von nach oben hin abnehmender Größe.
- Man soll den ganzen Stapel auf den dritten Stab transferieren, darf aber immer nur jeweils eine Scheibe entweder nach ganz unten oder auf eine größere legen.
- Angeblich sind in Hanoi ein paar Mönche seit Urzeiten mit dem Fall n = 64 befasst.

- Idee: 1883 Edouard Lucas, 1842- 1891, franz. Math.
- Basierend auf Legende der Türme von Brahma:

  In einem Tempel in der indischen Stadt Benares liegen 64 kostbare Scheiben aus Diamant zu einem Turm aufgeschichtet. Wenn der Turm an einer Stelle abgebaut und an andere Stelle wieder ganz aufgebaut wurde, wird der Tempel und mit ihm die ganze Welt zu Staub zerfallen.
- *[History of Math. Archive, St. Andrews;*
- *W.W.R. Ball, Mathematical and Recreational Essays - The Macmillan Co., NY 1939]*

Lösung

- Für n = 1 kein Problem.
- Falls man schon weiß, wie es für n - 1 geht, dann schafft man mit diesem Rezept die obersten n - 1 Scheiben auf den zweiten Stab (die unterste Scheibe fasst man dabei als “Boden” auf.).
- Dann legt man die größte nunmehr freie Scheibe auf den dritten Stapel und verschafft unter abermaliger Verwendung der Vorschrift für n-1 die restlichen Scheiben vom mittleren auf den dritten Stapel.

Lösung in SML

```
fun hanoi_aux(0, _, _, _) = []
|   hanoi_aux(n, from, to, using) =
      hanoi_aux(n-1, from, using, to) @
      [(from, to)] @
      hanoi_aux(n-1, using, to, from);
fun hanoi(n) = hanoi_aux(n, 1, 3, 2);
```

Echte nichtlineare Rekursion

- Bei vielen nichtlinearen Rekursionen wir bei Ackermann oder `mergeSort` lässt sich die nichtlineare Rekursion nur “künstlich” vermeiden, indem man Hilfsdatenstrukturen einführt, z.B. Stapel, die genauso viel Platz verbrauchen, wie die Verwaltung der ursprünglichen Rekursion.
- Hier ist also keine echte Verbesserung möglich. In diesen Fällen ist die nichtlineare Rekursion eine vernünftige Lösung.
- Grundsätzlich ist im Frühstadium der Softwareentwicklung eine klare rekursive Lösung vorzuziehen. Bei modularer Planung kann diese später ggf. durch eine effizientere Lösung ersetzt werden.

Erinnerung

- Eine rekursive Definition kann auch deshalb ineffizient sein, weil jeder einzelne Verarbeitungsschritt aufwendig ist, nicht weil die Zahl der rekursiven Aufrufe selbst zu groß wäre.
- **Beispiele**: Sortieren durch Einfügen oder eine ineffiziente Version des Umdrehens einer Liste mit “@”
  - ```
    fun rev nil = nil
      | rev (x::l) = (rev l) @ [x];
    ```
- Hier haben wir jeweils O(n) rekursive Aufrufe, deren jeder aber Aufwand O(n) verursacht, also Gesamtaufwand $O(n^2)$.
- Während bei der Einbettung (wg O(`::`) = konstant)
  - ```
    fun rev_aux nil res = res
      |   rev_aux(x::l) res = rev_aux l (x::res);
    ```
  - ```
    fun rev l = rev_aux l nil;
    ```

  nur Gesamtzeitaufwand O(n) und Platzbedarf O(1) erforderlich ist.
    ```

Zusammenfassung

- Effiziente Programme erhält man
  - durch die Wahl effizienter Algorithmen und Datenstrukturen und
  - durch Einbeziehung des Auswerteprozesses in die Programmkonstruktion.
- In vielen Fällen lassen sich Platzbedarf und Laufzeit rekursiver Definitionen durch Einbettung in endständige Rekursion verbessern.