Rekursive Funktionen

und ihre programmtechnische Umsetzung

Klaus Kusche, Juli 2012

Inhalt

- Die Idee und ihre Programmierung
- Die Abarbeitung zur Laufzeit
- Die Speicherung der Daten
- Praktisches & Theoretisches
- Anwendungen

Die Idee

"Löse eine Aufgabe,
indem du sie auf ein oder mehrere
<u>gleichartige, aber "kleinere" Aufgaben</u>
zurückführst,
die du wieder <u>genauso löst</u>,

bis die Aufgaben <u>so klein</u> geworden sind,
dass die Lösung <u>ganz einfach</u> ist."

==> Das Lösungsverfahren
"<u>verwendet sich selbst</u>" für Teilaufgaben!

… und ihre Programmierung

- Eine Funktion

ruft sich selbst auf

d.h. der Code der Funktion **recFunc**
enthält wieder einen Aufruf von **recFunc(…)**

==> "*direkte Rekursion*"

- Oder ("*indirekte Rekursion*"):

funcA ruft **funcB** auf und
funcB ruft wieder **funcA** auf.

(auch über 3 oder noch mehr Funktionen)

Beispiel: Die Fakultät

```
int fak(int n)
{
  // Der „ganz einfache“ Fall:
  // „Direkte“ Lösung
  if (n <= 1) return 1;

  // Sonst:
  // Löse das „nächstkleinere“ Problem
  // und berechne daraus die eigene Lösung
  else return n * fak(n - 1);
}
```

Ablauf anschaulich...

“**Die Fakultäts-Brüder**”:
Lauter idente “*Klone*” desselben Rechenmeisters!

Vater fragt ältesten Sohn: *Fakultät von 4?*

Ältester Sohn fragt mittleren Bruder: *Fakultät von 3?*

Mittlerer fragt seinen kleinen Bruder: *Fakultät von 2?*

Kleiner Bruder fragt Baby-Bruder: *Fakultät von 1?*

Baby-Bruder antwortet kleinem Bruder: *1*

Kleiner Bruder rechnet 1*2, antwortet mittlerem: *2*

Mittlerer Bruder rechnet 2*3, antwortet großem Bruder: *6*

Der älteste rechnet 6*4, antwortet seinem Vater: *24*

Vater sagt: Sehr gut!

… und Ablauf technisch

- *Mehrere Aufrufe derselben Funktion* werden
  (mit verschiedenen Parametern)

  der Reihe nach gestartet.

- Zu jedem Zeitpunkt

  rechnet nur der aktuell “innerste” Aufruf,

  alle anderen warten auf die Rückkehr
  des von ihnen gestarteten nächstinneren Aufrufs.

- Die Aufrufe kehren

  in umgekehrter Reihenfolge ihres Aufrufs zurück!

Daten in rekursiven Funktionen

Die Daten in allen Aufrufen sind **unabhängig voneinander**:

- Jeder Aufruf hat seine

  eigenen lokalen Variablen und Parameter!

  Ausnahme:
  `static`-Variablen sind *für alle Aufrufe gemeinsam*!

- Jeder Aufruf merkt sich *getrennt* von den anderen,
  wohin er *zurückkehren* muss.

  (Der äußerste nach `main` oder in eine andere Funktion,
  die inneren in den vorigen Aufruf derselben Funktion.)

==> Es existieren *mehrere Sätze* von lokalen Variablen,
Parametern und Return-Adressen *gleichzeitig!*
(so viele, wie gerade Aufrufe aktiv sind)

Realisierung im Speicher (1)

Mittels **Call Stack** implementiert:

- *Linearer Speicherbereich* im RAM, meist „ganz oben“.

  (Wenn multi-threaded: Ein Stack *pro Thread*.)

- *Wächst und schrumpft an einem Ende* dynamisch, meist „nach unten“:

  - Bei jedem **Call**:

    \- Aufrufer legt die *Parameter* auf den Stack („push“).
    \- Call-Befehl legt die *Return-Adresse* auf den Stack.
    \- Aufgerufene Funktion reserviert *Platz für ihre Variablen*.

  - Bei jedem **Return**:

    Freigabe *genau umgekehrt*…

Realisierung im Speicher (2)

Ein *eigenes CPU-Register* („**Stack-Pointer**“) zeigt stets auf das aktuelle Stack-Ende.

==> Alle Funktionen adressieren
ihre Variablen und Parameter
relativ zum Stack-Pointer:

„Variable n *liegt auf*
Adresse (Stack-Pointer + 16)“

==> Trifft stets die *Daten des aktuell innersten Aufrufs*,
egal wie viele Aufrufe aktiv sind,
und unabhängig von den absoluten Adressen der Daten!

Damit es klappt ... (1)

Ende der Rekursion:

Jede rekursive Funktion **muss**
mindestens *einen nichtrekursiven Zweig haben!*

(d.h. ein `return` **ohne** vorherigen rekursiven Aufruf)

Das ist der “ganz einfache”, *direkt lösbare Fall*.

Sonst:

Endlos-Rekursion (endlos viele Aufrufe ohne `return`)
bis zum Absturz durch *Speicherüberlauf!*

(Stack voll ==› kein Platz für die Variablen
eines weiteren Aufrufs)

Damit es klappt … (2)

- Das Problem muss mit jedem Aufruf **kleiner** werden!
- Man muss mit jedem Aufruf einen Schritt **näher zum "ganz einfachen" Fall** kommen!

==> Jeder rekursive Aufruf
muss von seinen **Parameter-Werten** her

"**näher** *beim Rekursionsende sein*"

als der Aufrufer:

Kleinere Zahl, weniger Array-Elemente,
weniger noch offene Rechenschritte, …

Theoretische Erkenntnisse

- **`if` + Rekursion + unendlicher Stack** reichen als *universeller Rechen-Mechanismus* aus:

  ==> Man braucht *keine Schleifen* und kein `goto` !

  ==> Jede Schleife kann *durch Rekursion ersetzt* werden!

- Man kann jede Rekursion *durch eine Schleife ersetzen*, benötigt aber ein potentiell *unendlich großes Array* zur *Simulation des Stacks*.

(Beide Transformationen sind automatisiert möglich, resultieren aber in ziemlich *unleserlichem* Code.)

Anwendungsbeispiele

- Rekursion statt Schleifen
- Rekursiv definierte Datenstrukturen
- Divide and Conquer
- Backtracking,
  Alpha-Beta-Algorithmus
- (Compilerbau)
- Rekursive „Spielereien“

Rekursion statt Schleifen (1)

Die *klassischen Lehrbuch-Rekursionsbeispiele*
(Fakultät, rek. ggT nach Euklid, rek. binäre Suche, ...)
sind rekursiv *schöner, aber ineffizienter*:

Sie brauchen (im Vergleich zur Lösung mit Schleife):

- *Konstant mehr Rechenzeit*: Overhead von Call & Return
- *n-fach mehr Speicher*: 1 Aufruf pro Schleifenumlauf!

  ==> Stackbedarf wächst linear mit der Problemgröße,
  eine Schleife braucht nur konstant viel Speicher!

==> Rekursion ist in diesen Fällen

praktisch nicht sinnvoll!

Rekursion statt Schleifen (2)

Beispiele mit **zwei** **„um 1 kleineren“** rekursiven Aufrufen

```
int fib(int n) {
  if (n < 2) return 1;
  return fib(n - 1) + fib(n - 2);
}

int binom(int n, int k) {
  if (k > n) return 0;
  if ((k == 0) || (k == n)) return 1;
  return binom(n - 1, k - 1) + binom(n - 1, k);
}
```

... sind *extrem ineffizient* **($\sim 2^n$ rekursive Aufrufe) !!!**

Viel besser:

“*Von unten nach oben*” mit *Array und Schleife* rechnen!

==> Zeit $\sim n$ statt $\sim 2^n$, Speicher konstant statt $\sim n$

Aber …

… fast alle anderen Anwendungen von Rekursion sind

sehr wohl sinnvoll, effizient

und in der Praxis äußerst bedeutend!

Weil (gerade bei 2 oder mehr rekursiven Aufrufen!):

- Die „natürliche“ Lösungsidee ist oft rekursiv,
  eine reine Schleifen-Lösung ist *schwerer zu verstehen*.

  Bei zwei oder mehr rekursiven Aufrufen ist
  der „derekursivierte“ Code meist *völlig unleserlich und viel länger*!

- Es gibt meist *keine Lösungen* ohne Rekursion,
  die *signifikant effizienter* sind.

Rekursive Datenstrukturen

Beispiel: **Binärer Baum** (sinngemäß):

„Ein binärer Baum ist entweder *leer*
oder besteht aus einem Wert, an dem
links und rechts wieder ein binärer Baum hängt."

==> Fast alle Operationen auf Bäumen
machen einen *rekursiven Aufruf*
für den *linken und/oder rechten Unterbaum*:

```
int nodeCount(tree *p)
{
   if (p == NULL) return 0;
   else return
      1 + nodeCount(p->left) + nodeCount(p->right);
}
```

“Divide and Conquer”

= *“Teile und Herrsche”*

Wesentlicher Unterschied zu „primitiver“ Rekursion:

- Teile das Problem in <u>**mehrere**</u> **Teile**:

  Meist <u>*2 möglichst gleichgroße Hälften*</u> (n/2) statt 1 Unterproblem der Größe (n-1) .

- Löse <u>*rekursiv*</u> *jedes Teilproblem einzeln* für sich ==> <u>**mehrere**</u> rekursive Aufrufe!

- Berechne aus den Teillösungen die Gesamtlösung.

==> *Meist schöne & sehr effiziente Lösungen!*

Beispiel: Quicksort

- Such dir ein “*mittelgroßes*” Element.
- Schaufle alle *kleineren* Elemente nach *links*,
  alle *größeren* nach *rechts*.
- *Sortiere den linken und den rechten Teil* getrennt für sich (*rekursiv* wieder mit Quicksort)
  ==> Das gesamte Array ist danach schon fertig sortiert!
- Rekursionsende: Ein Teil der Länge 1 ist schon sortiert...

==> Rekursionstiefe: *~log(n)*, Zeit: *~n*log(n)*

==> Viel besser als „dummes“ Sortieren mit Schleifen:
Zeit: *~n*n*

Backtracking (1)

Wofür?

Such- und Optimierungsprobleme,
deren Lösung aus *Einzelschritten* / Einzelentscheidungen zusammengesetzt ist.

Wie?

Systematisches *Durchprobieren aller Möglichkeiten,*
aber bei “sinnlosen” Teilschritten bzw. Sackgassen
gleich umkehren, restliche Schritte gar nicht probieren

==> *1 Schritt zurück,* deshalb “**Backtracking**”.

==> Viele *sinnlose Kombinationen* werden
gar nicht berechnet!

Backtracking (2)

Grundidee:

- *Jede Ebene der Rekursion / jeder Aufruf probiert* *alle* *Möglichkeiten für* *einen* *Teilschritt*
- … und macht *für jede* *sinnvolle* *Möglichkeit*
  (und nur für diese!)
  einen *rekursiven* *Aufruf*
  zur Lösung der *restlichen* *Teilschritte*.

Weiterentwicklungen:

u.a. *Alpha-Beta-Algorithmus* für 2-Personen-Spiele
(Grundlage aller Schachprogramme)
= Backtracking + Bewertungsstrategie

Backtracking (3)

Funktion *"Mache den i-ten Schritt"*:

```
if Ziel erreicht / letzter schon Schritt gemacht
then Drucke Lösung
else
   for alle Möglichkeiten für den i-ten Schritt
      if Möglichkeit ist zulässig
         Speichere den aktuellen i-ten Schritt in der Lösung
         Mache rekursiv den (i+1)-ten Schritt
         Ev.: Lösche den i-ten Schritt wieder aus der Lösung
      else
         Ignoriere die Möglichkeit
```

Beispiel: 8 Damen

```
void probier(int brett[ANZAHL], int zeile)
{
  if (zeile == ANZAHL) {
    drucke(brett);
  } else {
    // setze die Dame in Zeile "zeile"
    int spalte;
    for (spalte = 0; spalte < ANZAHL; ++spalte) {
      if (ok(brett, zeile, spalte)) {
        brett[zeile] = spalte;
        probier(brett, zeile + 1);
      }
    }
  }
}
```

Rekursive „Spielereien“

- **„Towers of Hanoi“**:

  „Um n Scheiben umzulegen,
  lege zuerst n-1 Scheiben …“

- **Raumfüllende Kurven**
  (Drachenkurve, Hilbert-Kurve, Sierpinski-Kurve, …):

  „Ersetze jede Teilstrecke
  durch eine verkleinerte Kopie der ganzen Kurve!“