Programmierung und Modellierung

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

2. Rekursive Funktionen und Induktion

1. Dateien laden, Kommentare und Fallunterscheidungen
2. Rekursive Funktionen
   - Lineare, endständige, mehrfache, verschachtelte Rekursion, mögliche Nichtterminierung
3. Programmiermethodik für rekursive Funktionen
   - Technik der Einbettung
   - Beweis durch Induktion
   - Beweis der Terminierung
     - Abstiegsfunktion
     - Wohlfundierte Relation

2.1 Dateien laden, Kommentare und Fallunterscheidungen

Dateien laden (einlesen)

- Eine Datei, die z.B. meine `datei.sml` heißt, kann wie folgt geladen werden:

```
- use("meine_datei.sml");
val it = () : unit
```

  Dabei ist () (gesprochen "unity") der **einzige Wert** eines besonderen Datentyps namens `unit` .

- Der Datentyp `unit` wird für Funktionen verwendet, die eigentlich keinen Wert berechnen, sondern einen Seiteneffekt bewirken (wie das Laden von Funktionsdeklarationen aus einer Datei). `unit` entspricht dem Typ `void` in Java.

Kommentare

- In SML sind Kommentare beliebige Texte, die
  mit `(*` anfangen und mit `*)` enden.
- Geschachtelte Kommentare sind erlaubt.
- Beispiel:

```
(*
fun Vorzeichen(x : int) =
if x > 0 then 1
else if x < 0 then ~1
     else (* x = 0 *) 0;
*)
```

- Klare und präzise Kommentare sind in jedem Programm unabdingbar.
- Es ist naiv anzunehmen, dass ein Programm selbsterklärend sei.

Fallunterscheidung

- SML bietet zwei Möglichkeiten für Fallunterscheidungen:
  - **if-then-else** (für 2 Fälle)
  - **Pattern Matching** (für beliebig viele Fälle)
- If-then-else
  - Eine Funktion `vorzeichen` kann z.B. wie folgt definiert werden:

```
fun vorzeichen(x : int) =
  if x > 0 then 1
  else if x < 0 then ~1
       else (* x=0 *) 0;
```

  - Das Konstrukt `if Test then E1 else E2` stellt die Anwendung einer wie folgt definierten Funktion auf `Test` dar:

```
(fn true => E1 | false => E2)
```

  - [der Ausdruck `if Test then E1 else E2` entspricht somit

```
(fn true => E1 | false => E2)(Test) ]
```

Bemerkungen: If-Then-Else

- Der `else`-Teil von `if-then-else`-Ausdrücken muss immer angegeben werden (im Gegensatz zu vielen (imperativen) Programmiersprachen)!
  - Z.B. Ohne `else`-Teil hat `if B then A` keinen Wert, wenn die Bedingung `B` den Wert `false` hat.
  - ➢ in der funktionalen Programmierung unmöglich!

- In einem SML-Ausdruck

  ```
  if B then A1 else A2
  ```

  müssen `A1` und `A2` **denselben** Typ besitzen.

Pattern Matching (Musterabgleich)

- In der Definition der obigen anonymen Funktion sind zwei Aspekte bemerkenswert:
  - „|“ drückt eine Alternative aus.
  - Die Ausdrücke `true` und `false` stellen **Muster (patterns)** dar.
- "**Matcht**“ der Wert des aktuellen Parameters das erste Muster, so wird der Wert des Ausdrucks E1 geliefert. Ansonsten wird getestet, ob der Wert des aktuellen Parameters mit dem zweiten Muster "matcht“.
- Es können mehr als zwei Fälle in der Deklaration vorkommen:
- Die Muster werden sequenziell in der Reihenfolge der Definition probiert, bis eines den Wert des aktuellen Parameter „matcht“.
- Das Muster _ („**wildcard**“) stellt einen Fangfall dar, d.h. matcht jeden möglichen Wert des aktuellen Parameters.
- Das Wildcard-Symbol wird nicht im Rumpf eines Falles (also hinter „=>“) verwendet.

Pattern Matching (Musterabgleich)

- Das folgende Prädikat liefert `true`, wenn es auf eine ganze Zahl angewandt wird, die eine (nicht-negierte) Ziffer ist:

```
val Ziffer =
fn 0 => true
| 1 => true
| 2 => true
| 3 => true
| 4 => true
| 5 => true
| 6 => true
| 7 => true
| 8 => true
| 9 => true
| _ => false;
```

- **Vorsicht:**

  Pattern sind keine Tests wie etwa (x > 0), sondern mögliche Werte des Parameters!

2.2 Rekursive Funktionen

- Lineare Rekursion
  - Beispiele: Summe der *n* ersten natürlichen Zahlen, Fakultät
- EndständigeRekursion
  - Beispiel: Gerade
- Mehrfache Rekursion
  - Beispiel: Fibonacci
- Verschachtelte Rekursion
  - Beispiel: Ackermann
- Mögliche Nichtterminierung
  - Rödelheim

Beispiel für Rekursion
Rekursive Berechnung der Summe der *n* ersten natürlichen Zahlen

- Die Funktion summe, die zu jeder natürlichen Zahl n die Summe aller natürlichen Zahlen von 0 bis einschließlich n liefert, kann z.B. wie folgt definiert weren:

$$\text{summe}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ n + \text{summe}(n - 1) & \text{falls } n > 0 \end{cases}$$

- In SML:

```
fun summe(n) =
     if n = 0 then 0
     else n + summe(n-1);
```

- oder

```
val rec summe =
     fn 0 => 0
     |  n => n + summe(n-1);
```

- Man beachte das Symbol `rec`!
  Bei rekursiven Funktionen muss `rec` nach `val` angegeben werden.

Lineare Rekursion – Beispiel Fakultätsfunktion

- Die Fakultätsfunktion fak(n) = 1*2*…*n
  läßt sich folgendermaßen rekursiv definieren

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

- Die Fakultätsfunktion wächst sehr schnell:
  - `- fak 10;`
    `val it = 3628800`

    - fak(50);
    uncaught exception Overflow [overflow]
    raised at: <file C:\sml\fak.sml>
  - `fak (~1);`
    Das System terminiert nicht (bzw. erst nach Überlauf des Speichers mit Fehler).
- **Lineare Rekursion**
  - Eine rekursive Fktsdeklaration `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.
  - Beispiele für linear rek. Fktsdeklarationen sind `fak` und `summe`

Endständige Rekursion

- Eine rekursive Fktsdeklaration `f(x)= E(f,x)` heißt **endständig rekursiv (tail recursive)**, 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.
- **Beispiel: Test auf gerade Zahl**

```
fun gerade(n) =
    if n = 0 then true
    else if n = 1 then false
         else gerade(n - 2);
```

  - `gerade` ist linear rekursiv, wobei der rekursive Aufruf das äußerste Fkt.symbol im letzten else-Zweig ist.
  - Es gilt

```
gerade(x) = (x mod 2=0) .
```

Mehrfache Rekursion: Die Fibonacci-Zahlen

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

- **Interpretation:**

  `fib(n)` =
  - Hasenpopulation nach n Monaten unter der Annahme, dass Hasenpaare jeden Monat ein Paar von Nachkommen haben;
  - dies jeweils aber erst ab dem zweiten Lebensmonat.
- **Mehrfache Rekursion**
  - Eine rekursive Fktsdeklaration `f(x)= E(f,x)` heißt **mehrfach rekursiv**, wenn in mindestens einem Zweig einer Fallunterscheidung des Rumpfes `E(f,x)` 2 oder mehr rek. Aufrufe von `f` vorkommen.
  - Beispiel: Fibonaccifunktion `fib`

Leonardi Fibonacci
Ca. 1180-1240, Pisa
Liber Abbaci, 1227
(Buch der Rechenkunst, Multiplikation, Dreisatz, …
[Wikipedia]

Fibonacci-Zahlen und Goldener Schnitt

- **Goldener Schnitt**
  - $x : 1 = (1-x) : x$, d.h. $x^2 + x - 1 = 0$;
  - (Negative) Lösung $\phi \sim 1.618$
  - `fib(n)/fib(n-1)` konvergiert gegen ϕ:

    1/1 = 1, 2/1 = 2, 3/2 = 1.5, 5/3 = 1.666..., 8/5 = 1.6,
    13/8 = 1.625, 21/13 = 1.61538...

Verschachtelte Rekursion: Ackermann

- Noch schneller als die Fakultätsfunktion wächst die Ackermann-Funktion (1926); vereinfachte Version von Rosza Peter (1955):

```
fun ack(n, m) =
  if n = 0 then m+1
  else if m = 0 then ack(n-1, 1)
       else ack(n-1, ack(n, m-1));
```

- Es gilt
  - `ack(0, m) = m+1, ack(1, m) = m+2,`
  - `ack(2, m) = 2m+3, ack(3, m) ~ 2^m,`
  - `ack(4, 2) ist Zahl mit 19729 Stellen`

    [http://kosara.net/thoughts/ackermann42.html]
- **Verschachtelte Rekursion**
  - Eine rekursive Fktsdeklaration `f(x)= E(f,x)` heißt **verschachtelt rekursiv**, wenn im Rumpf `E(f,x)` mindestens ein rekursiver verschachtelter rek. Aufruf der Form `f(H(f))` von `f` vorkommt, wobei H einen rek. Aufruf von f enthält.
  - Beispiel: Ackermannfunktion

Wilhelm Ackermann
1896-1962,
Diss 1924 bei Hilbert
Mathelehrer in Lüdenscheid
Prof. h.c. Münster
[Wikipedia]

Rósza Péter
"Mutter der Theorie der rek. Fkt.
1905-1977, Diss 1935
1951 Buch "Recursive Functions
[http://www.sdsc.edu/ScienceWomen]

Nichtterminierung: Rödelheim

- Beachte: Rekursion kann eine Quelle von Undefiniertheit sein:

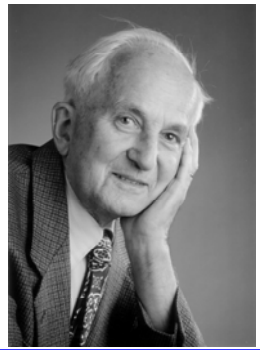
```
fun roedelheim(n) =
    if n = 0 then 1 else roedelheim(n + 1);
```

- Es gilt
  - `roedelheim(0) = 1`
  - `roedelheim(n)` = undefiniert

    für alle n > 0; d.h. Definitionsbereich D(`roedelsheim`) = {`0`}

Weitere Beispiele: Collatz-Vermutung (auch Ulam-Funktion)

- ```
  fun col n =
      if n = 1 then 1
      else if n mod 2 = 0
              then col(n div 2)
              else col(3*n + 1);
  ```
- **Beispiele**
  - `col(3) = col(10) = col(5) = col(16) = col(8) = col(4) = col(2) = col(1)`
  - `col(4) = col(2) = col(1)`
  - `col(5) = col(16) = col(8) = col(4) = col(2) = col(1)`
- `col(n)` terminiert für alle bisher untersuchten natürlichen Zahlen
- Allgemeiner Beweis der Terminierung ist bis jetzt unbekannt
  - Preise für Lösung sind ausgesetzt von : Thwaites (1996) 1000 $, Coxeter (1970) 50 bzw. 100 $



Lothar Collatz
1910-1990, Prof.
Uni Hamburg
Funktionalanalysis,
Num. Mathe.
[Wikipedia, Foto-Studio Volksdorf]
  ```

2.3 Programmiermethodik für rekursive Funktionen

- Beweis durch vollständige Induktion
- Technik der Einbettung
- Beweis der Terminierung
  - Abstiegsfunktion
  - Wohlfundierte Relation

Effiziente Berechnung der Summe der n ersten natürlichen Zahlen

- Die Summe der ersten *n* nat. Zahlen kann auch explizit durch einen geschlossenen Ausdruck definiert werden:

```
fun summe1(n) = (n * (n + 1) ) div 2;
```

➢ wesentlich effizientere Berechnung ohne Verwendung von Rekursion, bei der für jedes *n* nur drei Grundoperationen benötigt werden.

- Warum gilt die Eigenschaft

  summe(n) = n(n+1) div 2 ?

  Dies kann durch vollständige Induktion gezeigt werden.

Vollständige Induktion

Sei P(n) eine Eigenschaft der natürlichen Zahlen.

- **Prinzip der vollständigen Induktion**
  - Induktionsanfang: Beweise P(0)
  - Induktionsannahme: P(k)
  - Induktionsschluss k => k+1: Zeige P(k+1) unter der Annahme, dass P(k) gültig ist.

  Dann gilt die Eigenschaft P(n) für alle $n \in N$

Beweis durch vollständige Induktion

- Die Eigenschaft summe(n) = n(n+1) div 2 gilt für alle n $\in$ N.

  Beweis durch vollständige Induktion:

  - Induktionsanfang: Für n = 0 gilt die Gleichung summe(0) = 0 = 0(0+1) div 2.
  - Induktionsannahme : Für k gelte summe(k) = k(k+1) div 2.
  - Induktionsschritt: k => k+1

    Nun zeigen wir, dass die Gleichung summe(k+1) = (k+1)(k+2) div 2

    für die Nachfolgerzahl k + 1 gilt:

    summe(k + 1) = [Def. Summe]

    k + 1 + summe(k) = [Ind.annahme]

    k + 1 + (k(k+1) div 2) = [Algebraische Umformung]

    (2(k+1)+k(k+1)) div 2 = [Algebraische Umformung]

    (k+1)(k+2) div 2 qed

Technik der Einbettung

- Häufig muss man vor einer rekursiven Lösung das Problem generalisieren.
- Diese Technik bezeichnet man als **Einbettung.**
- **Beispiel: Primzahlen**
  - Man soll bestimmen, ob n eine Primzahl ist.
  - Gesucht

    istPrim : **nat** -> **bool** mit istPrim(n) = **true** gdw. n ist prim.
  - Lösungsansatz durch Fallunterscheidung

```
fun istPrim(n) =
 if n = 0 orelse n = 1 then false
 else if n = 2 then true
        else ???
```

Einbettung: Beispiel Primzahltest

- **Zwei Schritte**
  - Bette die Operation istPrim(n) durch einen zusätzlichen Parameter in eine Funktion keineTeiler(n, k) ein, die prüft, ob die Zahl n keine Teiler im Bereich k, . . . , n-1 besitzt.
  - Definiere den Zusammenhang zwischen istPrim und der Einbettung:
    istPrim(n) = n>1 $\wedge$ keineTeiler(n, 2)
- **Lösung in SML**
  - ```
    fun keineTeiler(n, k) : bool =
      if k >= n-1 then true
      else not((n mod k) = 0) andalso
           keineTeiler(n, k+1);
    ```
  - ```
    fun istPrim(n) = n > 1 andalso keineTeiler(n, 2);
    ```

Einbettung: Endständig rekursive Fakultätsfunktion

- Einbettung der Fakultätsfunktion in eine endständig rekursive Funktion `fak1`
  - ```
    fun fak1(n,res) =
      if n=0 then res
      else fak1(n-1, n * res);
    ```
  - `fun fak(n) = fak1(n, 1);`
- **Bemerkung**

  Endständige Rekursion entspricht der Iteration mit einer while-Schleife
  - Beispiel: fak1(n1, res) entspricht

    ```
    while not(n1 = 0) {res = res*n1; n1 = n1-1;}
    return res;
    ```
  - Damit entspricht `fak(n)` dem while-Programm:

    ```
    n1=n; res=1;
    while not(n1 = 0) {res = res*n1; n1 = n1-1;}
    return res;
    ```
    ```

Induktionsbeweis

- Um die Äquivalenz der beiden Definitionen der Fakultät nachzuweisen, zeigen wir

  `fak1(n, res) = 1! * res`

- Beweis durch vollständige Induktion über k.
  - Induktionsanfang: Für k = `0` gilt die Gleichung

    `fak1(0,res) = res= 1*res = 1! *res` wegen 1! = 1.
  - Induktionsannahme : Für k gelte `fak1(k,res) = k! * res`
  - Induktionsschritt: k => k+1

    Nun zeigen wir, dass die Gleichung `fak1(k+1) = (k+1)! * res` für die Nachfolgerzahl `k + 1` gilt:

    `fak1(k+1, res)` = [Def. fac1]

    `fak1(k, (k+1)*res)` = [Ind.annahme]

    `k! * (k+1)*res` = [wg. (k+1)! = k! * (k+1)]

    `(k+1)! * res` qed

Terminierungsbeweis mit Abstiegsfunktion

- Um festzustellen, ob eine rekursiv definierte Funktion terminiert, kann man eine Abstiegsfunktion verwenden.
- Sei `fun f(x) = E(f, x)`

  eine rekursive Definition einer Funktion f : A –> B
- Wir wollen zeigen, dass `f(x)` für alle x aus einer Teilmenge A0 von A terminiert.

- **Abstiegsfunktion**

  Eine Funktion m : A –> N heißt **Abstiegsfunktion (für f und A0)**, falls für alle x ∈ A0 gilt:
  - Im Term `E(f,x)` wird f nur für solche y ∈ A0 **aufgerufen**, für die gilt: m(y) < m(x).

  Dann gilt `f(x)` terminiert für alle x ∈ A0

Beispiele für Abstiegsfunktionen

- **Fakultät**
  - `fun fac(n) = if n=0 then 1 else n * fac(n-1);`
  - Hier nehmen wir A0 = **N** und m(x) = x.
- **Summe, Fibonacci**
  - Hier wählen wir jeweils ebenso A0 = **N** und m(x) = x
- **keineTeiler**
  - ```
    fun keineTeiler(n, k) : bool =
      if k >= n-1 then true
      else not((n mod k) = 0) andalso
           keineTeiler(n, k+1);
    ```
  - Hier setzen wir

    A0 = {(n, k) $\in$ N ^{x} N | k <= n} und

    m(n, k) = if k=n then 0 else n-k.
    ```

Wohlfundierte Relationen

- Sei M eine Menge.

  Eine Relation $R \subseteq M \times M$ heißt **wohlfundiert**, wenn es

  keine unendliche Folge $a_1, a_2, a_3, \ldots$ von Elementen in M gibt,

  so dass $a_{i+1}\ R\ a_i$ für alle $i \in N$.

- Ist R eine wohlfundierte Relation auf einer Menge M, so kann man anstelle einer Abstiegsfunktion m : A –> N auch eine

  Abstiegsfunktion m : A –> M

  wählen, derart dass immer

  m(y) R m(x) gilt, wenn `f(y)` im Rumpf `E(f, x)` aufgerufen wird.

Beispiele für wohlfundierte Relationen

- Kleiner-Relation auf natürlichen Zahlen
  - M = N und xRy gdw. x < y
- Nachfolger-Relation auf natürlichen Zahlen
  - M = N und xRy gdw. y = x + 1
- Lexikographische Ordnung
  - $M_{lex} = N^x N$ und
  - (x1, x2) R (y1, y2) gdw x1 < y1 oder (x1 = x2 und y1 < y2).

- Mit dieser wohlfundierten Relation kann man die Terminierung der Ackermannfunktion beweisen
  - ```
    fun ack(n, m) =
      if n = 0 then m+1
      else if m = 0 then ack(n-1, 1)
         else ack(n-1, ack(n, m-1));
    ```
- Hier wählen wir die Abstiegsfunktion m: $M = N^x N \rightarrow M_{lex}$ und m(x, y) = (x, y).
    ```

Induktionsprinzip für wohlfundierte Relationen

- Sei
  - R eine wohlfundierte Relation auf einer Menge M,
  - m : A –> M eine Funktion,
  - P eine Eigenschaft vom Elementen aus A.
- **Induktionsprinzip**
  - Zeige für alle a ∈ A die Eigenschaft P(a) unter der Annahme, dass P(y) gilt für alle y ∈ A mit m(y) R m(a).
  - Dann gilt P(a) für alle a ∈ A

Beispiel Induktion über natürlichen Zahlen

1. **Nachfolger-Relation über nat. Zahlen**

   d.h. A = M = N, m(n) = n und xRy, falls y = x + 1

   Dann ist das Induktionsprinzip äquivalent zur vollständigen Induktion, denn für die Gültigkeit von P(n) für alle $n \in N$ muss man zeigen:

   - P(0) ohne Voraussetzungen (da 0 keinen Vorgänger besitzt) und
   - P(k+1) unter der Annahme, dass P(k) gilt.

2. **Kleiner-Relation über nat. Zahlen**

   d.h. A = M = N , m(n) = n und xRy gdw. x < y

   Dann ist Folgendes für die Gültigkeit von P(n) für alle $n \in N$ zu zeigen:

   - P(0) ohne Voraussetzungen (da 0 keinen Vorgänger besitzt) und
   - P(k) unter der Annahme, dass P(m) gilt für alle m<k.

Beispiel: Schnelle Potenzbildung

- Betrachte folgende Exponentialfunktion

```
fun exp(x: real, n) =
    if n=0 then 1.0
    else if  gerade(n) then sq(exp(x, n div 2))
         else x * sq(exp(x, n div 2));
```

- Hier ist `sq` die Quadratfunktion `sq(x:real) = x*x;`
- Es gilt exp(x, n) = x^n.

  Beweis: Nächste Folie

Beispiel: Schnelle Potenzbildung

Es gilt $exp(x, n) = x^n$.

Beweis:

- Wähle als wohlfundierte Ordnung die Kleiner-Relation auf natürlichen Zahlen, d.h. M = N und xRy gdw. x < y
- Zeige für alle n $\in$ N: P(n) = ($\forall$ x:real. $exp(x, n) = x^n$)

  Beweis:

  1. Fall n=0: $exp(x, 0) = 1.0 = x^0$.

  2. Fall n>0:

  Induktionsannahme: Es gilt $exp(x, m) = x^m$ für alle m<n

  a) n gerade:

  $exp(x, n) = sq(exp(x, n \text{ div } 2)) =$ [Ind.Ann. für n div 2]

  $sq(x^{n \text{ div } 2}) = x^n$

  b) n ungerade:

  $exp(x, n) = x * sq(exp(x, n \text{ div } 2)) =$ [Ind.Ann. für n div 2]

  $x * sq(x^{n \text{ div } 2}) = x^n$

Zusammenfassung

- Fallunterscheidung bildet man in SML mit
  - If-then-else oder
  - **Pattern Matching**
- Rekursive Funktionen
  - Wichtige Typen rekursiver Funktionen sind
    lineare, mehrfache und verschachtelte Rekursion
  - Beispiele für rek. Definitionen sind die **Fakultät, Fibonacci, Ackermann und Collatz-Funktion**
  - **Endständige Rekursion** ist ein Spezialfall linearer Rekursion und **entspricht der Iteration mit While-Schleifen**
- Programmiermethodik für rekursive Funktionen
  - Die **Technik der Einbettung** wird häufig zum Aufstellen rek. Gleichungen sowie zur Umformung in endständige Rekursion benötigt
  - Eigenschaften von rek. Definitionen zeigt man meist durch Induktion
    - Techniken dafür sind die **vollständige Induktion** und die allgemeinere Technik der **wohlfundierten Induktion**
  - Die **Terminierung rekursiver Definitionen** wird zeigt man mittels **Abstiegsfunktion** bzgl.
    - N oder allgemeiner einer
    - wohlfundierten Relation