Programmierung und Modellierung

Polymorphe Datentypen

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

SS 2009

Inhalt

- Ergänzung: Termauswertung

  3.4 Termauswertung für Fallunterscheidung, nichtstrikte Boolesche Operatoren und Deklarationen

- Kap. 4 Polymorphe Datentypen
  1. Wiederholung und Ergänzung:
     - Was ist ein Typ?
     - Basistypen
  2. Kartesisches Produkt (n-Tupel)
  3. Benutzerdefinierte Typnamen
  4. Verbunde (Records)
  5. Listen

3.4 Auswertung der Sonderausdrücke

1. Wertdeklarationen
2. if-then-else
3. Boolesche Operatoren `andalso` und `orelse`

3.4.1. Wertdeklarationen (val und fun)

- **(Einfache) Wertdeklaration** `val N = A`:

  Es wird die Gleichung `N = A` in die Umgebung eingefügt.
- **Funktionsdeklaration der Form** `val N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt (also wie oben)
- **Rek. Funktionsdeklaration der Form** `val rec N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt. Zusätzlich werden Vorkehrungen getroffen, dass die Umgebung von `A` auch diese Gleichung für `N` enthält.
- **Funktionsdeklaration der Form** `fun N P = A`:

  wie bei der rekursiven Funktionsdeklaration (vorheriger Fall)
- **Bemerkung:**
  - Wird eine Gleichung zu der Umgebung hinzugefügt, so wird sie auf Korrektheit überprüft:. Eine Deklaration wie z.B. `val zwei = zwei` wird dabei abgelehnt. Die verwendeten Namen im Rumpf (hier: zwei) müssen für die Auswertung einer Wertdeklaration bekannt sein.

3.4.2 if-then-else

- Das Grundprinzip der applikativen Reihenfolge, zunächst immer alle Teilausdrücke auszuwerten, kann zur Nichtterminierung bei rekursiv definierten Funktionen führen.
- Daher muss ein Ausdruck

  ```
  if A1 then A2 else A3
  ```

  als Sonderausdruck mit eigenem (nichtstrikten) Auswertungsmechanismus behandelt werden:
  - Werte von den drei Teilausdrücken `A1, A2, A3` zuerst nur `A1` aus.
  - Hat A1 den Wert true, dann (und nur dann) wird A2 ausgewertet (und A3 wird nicht ausgewertet). Der Wert von A2 wird als Wert des gesamten Ausdrucks geliefert.
  - Hat `A1` den Wert false, dann (und nur dann) wird `A3` ausgewertet (und `A2` wird nicht ausgewertet). Der Wert von `A3` wird als Wert des gesamten Ausdrucks geliefert.

3.4.3 Die Boole'schen Operatoren andalso und orelse

- Ebenso werden die Boole‘schen Operatoren andor und orelse **sequentiell nichtstrikt** ausgewertet.
- **Sequentielle Konjunktion (andalso, sequ. ∧)**
  - Zunächst wird nur der erste Teilausdruck A1 ausgewertet.
    - Ist der Wert von `A1` `true`, so wird auch A2 ausgewertet und dessen Wert als Wert des Ausdrucks `A1 ∧ A2` geliefert.
    - Ist der Wert von `A1` `false`, so wird `false` als Wert des Ausdrucks `A1 ∧ A2` geliefert (und `A2` wird nicht ausgewertet).
- **Sequentielle Disjunktion (orelse, sequ. ∨)**
  - Zunächst wird nur der erste Teilausdruck A1 ausgewertet.
    - Ist der Wert von `A1` `false`, so wird auch A2 ausgewertet und dessen Wert als Wert des Ausdrucks `A1 ∨ A2` geliefert.
    - Ist der Wert von `A1` `true`, so wird `true` als Wert des Ausdrucks `A1 ∨ A2` geliefert (und `A2` wird nicht ausgewertet).

Die Boole'schen Operatoren andalso und orelse

- Wie kann man herausfinden, wie eine Programmiersprache die Boole'schen Operationen auswertet?
- In SML reicht dazu der folgende kleine Test (hier für die Konjunktion):

```
fun roedelBool(n) : bool =
          roedelBool(n + 1);
false andalso roedelBool(0);
```

- Terminiert dieser Ausdruck, dann wertet die Programmiersprache die Konjunktion nichtstrikt aus.

Kap. 4 Polymorphe Datentypen

1. Wiederholung und Ergänzung:
   1. Was ist ein Typ?
   2. Basistypen
2. Kartesisches Produkt (n-Tupel)
3. Benutzerdefinierte Typnamen
4. Verbunde (Records)
5. Listen

4.1 Wiederholung und Ergänzung: Was sind Typen?

- Ein **Typ** (oder Datentyp) ist eine Menge von Werten.
- Mit einem Typ werden Operationen (bzw. Prozeduren) zur Bearbeitung der Daten des Typs angeboten. Eine **Datenstruktur** (oder auch **Rechenstruktur**) besteht aus einem Typ und den dazu angebotenen Operationen.
- Mit einem **vordefinierten Typ** bieten Programmiersprachen die Operationen, Funktionen oder Prozeduren an, die zur Bearbeitung von Daten des Typs üblich sind.
- Moderne Programmiersprachen ermöglichen es, dass man selbst Typen definieren kann (**benutzerdefinierte oder selbst definierte Typen**).

4.1. Die Basistypen von SML

- **Ganze Zahlen (`int`)**
  - Operationen
    - ~, +, *, -, div, mod und Vergleichsoperationen =, <, <=, >, >=
    - real: int –> real zur Konvertierung nach real
- **Reelle Zahlen (`real`) : Gleitkommazahlen**
  - Bemerkung: real entspricht einer endlichen Teilmenge der rationalen Zahlen, wobei bei der Arithmetik Rundungsfehler auftreten können.
  - Operationen:
    - `~, +, *, -, /` und Vergleichsoperationen `<, <=, >, >=` (kein = !)
    - Konvertierungsfunktionen vom Typ `real -> int`:
      `floor` rundet "nach unten“, `ceil` "nach oben“, `trunc` durch Weglassen der Nachkommastellen
    - `inf` Konstante für „infinite“ (unendlich)
    - `nan` Konstante für „not-a-number“ (keine Zahl)
- **Boole'sche Werte (`bool`)**
  - Operationen `not, andalso, orelse`

Die Basistypen von SML (2)

- **Endliche Zeichenfolgen (`string`)**
  - Operationen
    - `""` (leere Zeichenfolge), `\n` (newline), `\t` (tab), `\\` (\)
    - `size, ^` (Konkatenation), Vergleichsoperationen (lexikograph. Ordnung)
- **Zeichen (`char`)**
  - Geschrieben als Zeichenfolge mit vorangestelltem #
    - Beispiel: `#"z"` für das Zeichen z
  - Konvertierung
    - `chr : int -> char` liefert das ASCII-Zeichen mit Code n für n∈{0, …, 255})
    - `ord : char -> int` liefert Ordinalzahl des Buchstaben
    - `str: char -> string` konvertiert Buchstaben in `string`
    - `String.sub : string * int -> char` liefert n-ten Buchstab. einer Zeichenfolge
    - String.substring : `string*int*int -> string` hat ein Wort und zwei ganze Zahlen für den Start und die Länge des Substrings als Argumente und liefert das passende Teilwort
- **Der einelementige Typ `unit`**
  - ( ) der einzige Wert, oft „unity“ ausgesprochen.

Beispiele

- Unendlich und “not-a-number”

```
- 1.0/0.0;
val it = inf : real
- 0.0/0.0;
val it = nan : real
```

- Lexikographischer Vergleich

```
- “abc” < “aalborg”;
val it = false : bool
```

- Länge eines Worts

```
- size “abcd”;
val it = 4 : int
```

- Ein Wort duplizieren

```
- fun double s = s ^ s;
```

- Ersten Buchstaben hinten anfügen

```
- fun f s =
substring(s,1,size s - 1) ^ substring(s,0,1);
- f “abba”;
val it = ”bbaa” : string
```

4.2 Kartesisches Produkt

- Sind A_1 und A_2 Mengen, so bildet man das **kartesische Produkt**

  $A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ und } a_2 \in A_2\}$.

  Die Elemente von $A_1 \times A_2$ heißen Paare (oder 2-Tupel)
- In SML:
  - Sind `t1` und `t2` Typen, so bezeichnet

    `t1* t2` den Typ des **kartesischen Produkts.**
  - Ist `a1` ein Ausdruck vom Typ `t1` und `a2` ein Ausdruck vom Typ `t2` , so ist

    `(a1, a2)` ein Ausdruck vom Typ `t1* t2`
  - Bsp: `-("abc", 44);`

    ```
    val it = ("abc",44) : string * int
    ```
- Ist f eine Funktion von $A_1 \times A_2$ nach B, so kann man f auf Paare anwenden.
  - Beispiel in SML:

    ```
    fun f (x, y) = x + 2*y;
    ```
  - Man schreibt **nicht** `f((x, y))`!
- Eine Funktion von $A_1 \times A_2$ nach B heißt **zweistellig**.

Mehrstelliges Kartesisches Produkt

- Allgemeiner: Sind $A_1, \ldots, A_n$ Mengen (n>= 0), so bildet man das **kartesische Produkt**

  $$A_1 \times \ldots \times A_n = \{(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots, a_n \in A_n\}.$$

  Die Elemente von $A_1 \times \ldots \times A_n$ heißen n-Tupel oder Vektoren.
- In SML:
  - Sind $t_1, \ldots, t_n$ Typen, so bezeichnet

    $t_1 * \ldots * t_n$ den Typ des **kartesischen Produkts.**
  - Ist a_1 ein Ausdruck vom Typ t_1, …, a_n ein Ausdruck vom Typ t_n, so ist

    $(a_1, \ldots, a_n)$ ein Ausdruck vom Typ $t_1 * \ldots * t_n$
  - Es gilt:
    - n= 0: Der Typ des 0-stelligen Tupels ist `unit` .
    - n= 1: Das einstellige Tupel `(a)` ist äquivalent zu `a` .
  - Bsp: `-("abc", 44, fn x => 2*x);`

```
val it = ("abc",44,fn) : string * int * (int -> int)
```

- Eine Funktion von $A_1 \times \ldots \times A_n$ nach B heißt **n-stellig**.

Tupel als Argumente und Werte von Funktionen

- n-Tupel können als **Argumente** n-stelliger Funktionen auftreten
  - `- val paar = („abc“, 2);`
    `- String.sub paar;`
    `val it = „c“ : char`

- n-Tupel können als **Werte** n-stelliger Funktionen auftreten
  - Beispiel

    divmod : N x (N \ {0}) –> N x N

    divmod(a, b) = (q, r),

    wobei a = q*b + r und q, r ∈ N und r < b

    In SML:

    ```
    fun divmod(x, y) = (x div y, x mod y);
    ```

Kartesisches Produkt

- Geschachtelte Tupel: Komponenten von Tupeln können selbst Tupel sein

```
- (("abc", 44), (44, 89e~2));
val it = (("abc",44),(44,0.89))
          : (string * int) * (int * real)
```

- Die Gleichheit von n-Tupeln (derselben Länge!) ist komponentenweise definiert:

```
- (2*2, „abab“) = (4, double „ab“);
val it = true : bool
```

Selektion der Komponenten

- Pattern Matching

```
- val tripel = (1, #"z", "abc");
val tripel = (1,#"z","abc") : int * char * string
- val (komponente1, komponente2, komponente3) = tripel;
val komponente1 = 1 : int
val komponente2 = #"z" : char
val komponente3 = "abc" : string
```

- Selektion durch Angabe der Komponentenposition mit #1, #2, usw.:

```
- #1("a", "b", "c");
val it = "a" : string
- #3("a", "b", "c");
val it = "c" : string
```

4.3 Benutzerdefinierte Typnamen

- Typnamen können in SML vom Benutzer folgendermaßen selbst deklariert werden
  - `type <Name> = Typausdruck;`
  - Beispiel

```
- type punkt = real * real;
- fun abstand(p1: punkt, p2: punkt) =
let fun quadrat(z) = z * z
    val delta_x = #1(p2) - #1(p1)
    val delta_y = #2(p2) - #2(p1)
in
  Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
end;
val abstand = fn : punkt * punkt -> real
- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Benutzerdefinierte Typnamen

- Man beachte, dass `punkt` lediglich ein Synonym für `real * real` ist.
- In der Tat ist `(real * real) * (real * real)` der Typ des aktuellen Parameters der vorangehenden Funktionsanwendung (ihr Argument ist ein Paar von Paaren von Gleitkommazahlen).

4.4 Verbunde (Records)

- Verwendet man Bezeichner (Namen) zur Selektion von Komponenten eines Tupels, so spielt die Position der Komponente keine Rolle.

Allgemein: Verbunde

- Sind $t, f_1, \ldots, f_n$ (unterschiedliche) Bezeichner und $typ_1, \ldots, typ_n$ Typen, so wird durch

  $$type\ t = \{ f_1 : typ_1, \ldots, f_n : typ_n \};$$

  ein neuer Typ t definiert, dessen Werte von der Form

  $$f_1 = w_1, \ldots, f_n = w_n$$

  sind, wobei jeweils w_n ein Wert des Typs typ_i ist.
- Dieser Typ t heißt **Verbundtyp** oder auch **Recordtyp**. Die f_i heißen **Felder** des Verbundtyps.
- Die Reihenfolge $typ_1, \ldots, typ_n$ der Komponententypen ist unerheblich.
- **Bemerkung**

  Ein Verbundtyp ist ein ungeordnetes kartesisches Produkt mit selbst gewählten Selektornamen für die Komponenten.

4.5 Listen

- Ist A eine Menge, so ist

  A **list** = $\bigcup_{n>=0} A^n$

  die **Menge der Listen** über A.
  - Fasst man A als Alphabet auf, so ist A **list** = A*.
  - Man notiert Listen als

    $[a_1, \ldots, a_n]$

    anstelle von $(a_1, \ldots, a_n)$ oder $a_1, \ldots, a_n$.
- Beispiele

```
- [0,1,2,3];
val it = [0,1,2,3] : int list
- [true, false orelse true, not true];
val it = [true, true, false] : bool list
- [(9,2), (3,5)];
val it = [(9,2), (3,5)] : (int * int) list
- [[3], [3, 2], [3, 2, 1]];
val it = [[3], [3, 2], [3, 2, 1]] : int list list
```

Grundoperationen auf Listen

- Die **leere Liste** wird mit `nil` oder `[]` bezeichnet.
- **Hinzufügen eines Elements**: Ist a ∈ A und `l` = $(a_1, \ldots, a_n) \in$ A **list**, so bezeichnet `a :: l` die Liste $(a, a_1, \ldots, a_n)$
- Die **Verkettung von zwei Listen** `l1` und `l2` schreiben wir als `l1 @ l2`.
  - Es gilt

```
[] @ l2 = l2
(a :: l) @ l2 = a :: (l @ l2)
```

- Beispiele

```
- val x = 17::[];
val x = [17] : int list
- val y = 9 :: 2 :: x;;
val y : int list = [9; 2; 17]
- x @ y;
val it = [17; 9; 2; 17] : int list
- [x, y];
val it = [[17]; [9; 2; 17]] : int list list
```

Typvariablen und polymorphe Typen

- Eine **Typvariable** hat die Form `'<Name>` und kann mit beliebigen Typen instantiiert werden.
  - Mögliche Instanzen von `'a` sind z.B.

    `int, bool, int list` oder `int * (bool list)`
- Ein Typausdruck wie `'a list` wird **polymorpher Typ** oder **Polytyp** genannt, weil der Ausdruck für mehrere (griechisch „poly“, „πολυ“) Typen steht;
- `list` ist ein **Typkonstruktor**, der mit unterschiedlichen Typen instantiiert werden kann.
  - Mögliche Instanzen von `'a list` sind z.B.

    `int list, bool list, (int list) list` oder

    `(int * bool) list`
  - Weitere Beispiele für polymorphe Typen

    `'a * 'b, 'a list * 'b, ('a * bool) list`
- Ein Typ, der kein Polytyp ist, wird **Monotyp** genannt.

Muster für Listen

- Man definiert Funktionen über Listen durch **strukturelle Rekursion** mit Hilfe von Mustern der Form
  - `fun f(nil) = …`
    `| f(x :: l) = …`
- Kommt x nicht auf der rechten Seite vor, kann es durch _ („Wildcard“) ersetzt werden (analog für `l`)
- Muster können auch detaillierter sein, z.B.
  - `fun f(nil) = …`
    `| f(x :: nil) = …`
    `| f(x :: y :: l) = …`

Grundlegende Listenfunktionen

- **Länge einer Liste**
  - ```
    - fun laenge(nil) = 0
    | laenge(_ :: L) = 1 + laenge(L);
    val laenge = fn : 'a list -> int
    ```
- **Kopf und Rumpf einer Liste**
  - ```
    - fun head(x :: _) = x;
    Warning: match nonexhaustive x :: _ => ...
    val head = fn : 'a list -> 'a
    ```
  - ```
    - fun tail(_ :: L) = L;
    Warning: match nonexhaustive _ :: L => ...
    val tail = fn : 'a list -> 'a list
    ```
  - Bem. SML bietet die vordefinierten Funktionen hd und `tl` an.
    ```

Grundlegende Listenfunktionen: append

- **Konkatenation zweier Listen (selbstdefiniertes @)**
  - ```
    - fun append(nil, l)    = l
        | append(x :: t, l) = x :: append(t, l);
    val append = fn : 'a list * 'a list -> 'a list
    ```
  - **Beispiel**
    - ```
      - append([1,2,3], [4,5]);
      val it = [1,2,3,4,5] : int list
      ```

Grundlegende Listenfunktionen: append

- **Berechnungsschritte zur Auswertung von `append([1,2,3], [4,5])`:**

```
append(1::(2::(3::nil)),     4::(5::nil))
1 :: append(2::(3::nil),     4::(5::nil))
1 :: (2 :: append(3::nil,    4::(5::nil)))
1 :: (2 :: (3 :: append(nil, 4::(5::nil))))
1 :: (2 :: (3 :: (4::(5::nil))))
```

Es gibt keinen weiteren Berechnungsschritt mehr:

`1 ::(2 ::(3 ::(4 ::(5 :: nil))))`

ist die Liste `[1, 2, 3, 4, 5]`.

Grundlegende Listenfunktionen: append

Zeitbedarf von append

- Als Zeiteinheit kann die Anzahl der Aufrufe der Funktion `::` oder aber die Anzahl der rekursiven Aufrufe von `append` gewählt werden.
- Beide Zahlen stehen einfach miteinander in Verbindung:
  - Wird zur Berechnung von `append(l, l')` die `append`-Funktion n + 1 mal rekursiv aufgerufen, so wird die Funktion `::` n-mal aufgerufen.

(*) Um eine Liste `l` der Länge n mit n>=1 vor einer Liste `l'` einzufügen, ruft die Funktion `append` die Funktion `::` genau n-mal auf.

- Die Länge des zweiten Parameters L' beeinflusst nicht den Zeitbedarf von append.
- Ist n die Länge des ersten Parameters von `append`, so gilt:

  `append` $\in$ O(n).

Grundlegende Listenfunktionen

- **Test auf leere Liste**
  - `- fun is_empty l = (l = []);`

  `val is_empty = fn : ''a list -> bool`
  - Bemerkung: Mit `''a` bezeichnet man Typen, auf denen eine Gleichheitsoperation `=` definiert ist.
- **Test, ob ein Element in einer Liste enthalten ist**
  - `fun is_in(x, nil) = false`
    `| is_in(x, y :: l) = (x = y) orelse is_in(x,l);`

  `val is_in = fn : ''a * ''a list -> bool`
- **Revertieren von Listen**
  - `- fun reverse(nil) = nil`
    `| reverse(x :: t) = reverse(t) @ x :: nil;`

    `val reverse = fn : 'a list -> 'a list`
  - **Bem.:** SML bietet die vordefinierte Funktion `rev` an.

Naives Spiegeln

- Berechnungsschritte zur Auswertung von reverse([1,2,3]):

```
reverse(1::(2::(3::nil))) =
append(reverse(2::(3::nil)), 1::nil) =
append(append(reverse(3::nil), 2::nil), 1::nil) =
append(append(append(reverse(nil),3::nil),2::nil),1::nil)=
append(append(append(nil, 3::nil),2::nil),1::nil) =
append(append(3::nil,              2::nil),1::nil) =
append(3::append(nil,              2::nil),1::nil) =
append(3::(2::nil),                       1::nil) =
3::append(2::nil,                         1::nil) =
3::(2::append(nil,                        1::nil)) =
3::(2::(1::nil))
```

Naives Spiegeln

Zeitbedarf von reverse

- Zur **Schätzung der Größe** des Problems „Spiegeln einer Liste“ `l` bietet es sich an, die **Länge n der Liste `l`** zu wählen.
- Als **Zeiteinheit** wählen wir die **Anzahl der Aufrufe der Funktion `::`** [wie bei `append`].
  - Gegeben sei eine Liste `l` der Länge n mit n >=1.
  - Während des Aufrufes von `reverse(l)` wird die Funktion `reverse` n-mal rekursiv aufgerufen:
    - zunächst mit einer Liste der Länge n-1 als Parameter,
    - dann bei jedem weiteren Aufruf mit einer jeweils um ein Element kürzeren Liste
  - Wegen (*) (siehe Zeitbedarf von `append`) ergibt sich die Anzahl der Aufrufe der Funktion `::` zur Zerlegung der Eingabeliste als:

    (n-1) + (n-2) + … + 1
  - Zum Aufbau der auszugebenden Liste wird `::` zudem n-mal aufgerufen.
- Die Gesamtanzahl der Aufrufe von `::` lautet also:

  n + (n-1) + (n-2) + … + 1 = n * (n + 1) div 2
- Damit ist die Laufzeitkomplexität: `reverse` $\in O(n^2)$

Effizientes Spiegeln

- Idee: Füge eine zweite (rechte) Liste als Argument hinzu, mit der nach und nach die gespiegelte Liste konstruiert wird.
  - Anfangs ist die rechte Liste leer ist.
  - Nach und nach wird das erste Element der linken Liste entfernt und am Anfang der rechten Liste eingefügt.

Schritt	Linke Liste	Rechte Liste
0	`[1, 2, 3]`	`[]`
1	`[2, 3]`	`[1]`
2	`[3]`	`[2, 1]`
3	`[]`	`[3, 2, 1]`

- Einbettung von rev
  - ```
    fun  rev_aux ([], acc) = acc
    | rev_aux (x::l, acc) = rev_aux(l, x::acc);
    ```
  - ```
    fun rev2 l = rev_aux(l, []);
    ```

- Man erprobe `rev(mklist 10000)` und `rev2(mklist 10000)`, wobei `mklist n` eine Liste der Länge n ist.

Effizientes Spiegeln

- Berechnungsschritte zur Auswertung von reverse([1,2,3]):

```
reverse(1::(2::(3::nil))) =
rev_aux(1::(2::(3::nil)),  nil) =
rev_aux(2::(3::nil),    1::nil) =
rev_aux(3::nil,     2::(1::nil)) =
rev_aux(nil,    3::(2::(1::nil))) =
3::(2::(1::nil)))
```

- Zeitbedarf
  - Ist n>=0 die Länge einer Liste `l`, so bedarf die Auswertung von `rev_aux(l, nil)` n rekursiver Aufrufe sowie n Aufrufe der Funktion `::`
  - Es gilt also: `rev_aux` $\in$ O(n)
  - Folglich gilt auch: `reverse` $\in$ O(n)
- Bemerkung
  - Der zweite Parameter der Funktion `rev_aux` ist ein sogenannter Akkumulator.
  - Die Funktion `rev_aux` ist endrekursiv.

Strukturelle Induktion für Listen

- Man beweist Eigenschaften von Listen durch **strukturelle Induktion:**
  - Wenn `P(nil)` gilt und

    wenn für alle `l` vom Typ `t list` aus `P(l)` folgt,

    dass `P(x::l)` für alle `x` vom Typ `t` gilt,

    dann gilt `P(l)` für alle Listen `l` vom Typ `t list`
- **Bemerkung**
  - Strukturelle Induktion ist eine Form der wohlfundierten Induktion mit der Relation

    `l1 R l2 gdw. laenge(l1) < laenge(l2)`

- Man zeige durch strukturelle Induktion, dass für alle `l` und `acc` vom Typ `'a list` gilt:

```
rev_aux(l, acc) = rev(l) @ acc
```

Weitere Beispiele

- Kleinster Wert einer Liste

```
fun minwert [x]   = x
| minwert(x :: t) = Int.min(x,minwert(t));
```

- Kleinster und größter Wert einer Liste

```
fun minmax [x] = (x, x)
| minmax(x :: t) =
    let val (mn, mx) = minmax t
    in (Int.min(x,mn), Int.max(x,mx)) end;
```

- Einfügen in eine Liste

```
fun insert(x,nil)    = [x]
|  insert(x, y ::t ) = if x<=y then x :: y :: t
                       else y :: insert(x,t);
```

- Sortieren durch Einfügen

```
fun sort nil   = nil : int list
| sort(x :: t) = insert(x, sort t);
```

Zusammenfassung

- Ein **Typ** (oder Datentyp) ist eine Menge von Werten. Eine **Datenstruktur** (oder auch **Rechenstruktur**) besteht aus einem Typ und den dazu angebotenen Operationen.
- Basistypen von SML sind `int, real, bool, string, char, unit.`
- Eine **Typvariable** hat die Form `'<Name>` bzw. `''<Name>` und kann mit beliebigen Typen instantiiert werden.
- Ein **polymorpher Typ** oder **Polytyp beinhaltet eine oder mehrere Typvariablen** und kann deshalb mit beliebigen Typen instantiiert werden. Ein Typ, der kein Polytyp ist, wird **Monotyp** genannt.
- **Typnamen** können in SML vom Benutzer selbst deklariert werden.
- Das **kartesische Produkt** ist ein polymorpher Datentyp in SML mit **n-Tupeln** als Elementen. Die Standardselektoren haben die Form `#1, #2`, …
- Ein **Verbundtyp** ist ein ungeordnetes kartesisches Produkt mit selbst gewählten Selektornamen für die Komponenten.
- Der **Listentyp** `'a list` ist ein polymorpher Typ. Eine Typinstanz der Form `t list` beschreibt die Menge der endlichen Sequenzen mit Elementen aus dem Wertebereich von `t` .