Inhalt Kapitel 4: Datentypen und Polymorphie

1. Wiederholung und Ergänzung
2. Kartesische Produkte
3. Benutzerdefinierte Typnamen
4. Records (Verbunde)
5. Listen
6. Typvariablen und polymorphe Typen
7. Funktionen auf Listen

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

Basisdatentypen und Produkte

- SML bietet u.a. die Basisdatentypen `int`, `real`, `bool`, `string`, `char` an.
- Kartesische Produkte werden mit `*` notiert

```
- (37.5,"Betty",9);
val it = (37.5,"Betty",9) : real * string * int
```

- Grundfunktionen `+, -, *, /, andalso, orelse, not, <, >, <=, >=, div, mod,....`
- `<` für `string` bezeichnet die lexikographische Ordnung:

```
- "Martin" < "Martina";
val it = true : bool
"AAAAAAA" < "Anderer Schluesseldienst"
val it = true : bool
```

Tupel als Werte von Funktionen

n-Tupel können als Werte n-stelliger Funktionen auftreten

- Beispiel

$$\mathrm{divmod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

- Beispielwerte: $\mathrm{divmod}(13, 3) = (4, 1)$, $\mathrm{divmod}(17, 5) = (3, 2)$
- Formal: $\mathrm{divmod}(a, b) = (q, r)$, wobei $a = qb + r$ und $q, r \in \mathbb{N}$ und $0 \leq r < b$.
- In SML:

```
- fun divmod(x, y) = (x div y, x mod y);
val divmod = fn: int * int -> int * int
```

Geschachtelte Tupel

- Komponenten von Tupeln können selbst Tupel sein

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

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

```
- fun double s = s ^ s;
val it = fn: string -> string
- (2*2, abab) = (4, double ab);
val it = true : bool
```

Auswahl 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
```

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

```
- #1 tripel;
val it = 1 : int
- #3 tripel;
val it = "abc" : string
```

Typabkürzungen

Typnamen können in SML vom Benutzer folgendermaßen selbst deklariert werden

type Name = Typausdruck;

Beispiel

```
 - type punkt = real * real;
type punkt = real * real
- fun abstand(p1: punkt, p2: punkt) =
Math.sqrt(square(#1(p2) - #1(p1)) +
square(#2(p2) - #2(p1)));
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

  ```
  abstand((4.5, 2.2), (1.5, 1.9));
  ```

  (das Argument ist ein Paar von Paaren von Gleitkommazahlen).
- Der Vorteil benutzerdefinierter Typen, wie `punkt`, liegt in der besseren Strukturierung und Dokumentierung.

Records (Verbunde)

Zunächst ein Beispiel: Mit

```
type studi = {name:string, alter:int}
```

definieren wir einen Typ von Studierenden.

```
- val maier1 : studi = {name = "Maier", alter = 23};
val maier1 : studi = {name = "Maier", alter = 23}
- val maier2 : studi = {name = "Maier", alter = 22};
val maier2 : studi = {name = "Maier", alter = 22}
- fun studi2string (x : studi) = #name x ^ ", " ^
            Int.toString (#alter x) ^ " Jahre alt.";
val studi2string = fn : studi -> string
- studi2string maier1;
val it = "Maier, 23 Jahre alt." : string
- studi2string maier2;
val it = "Maier, 22 Jahre alt." : string
```

Weitere Beispiele

```
type complex = {re:real, im:real}
type rational = {num:int, denum:int}
type point = {x:real, y:real}
type name_t = {vorname:string, initial:char, nachname:string}
type datum = {tag: int, monat: int, jahr: int}
type mitarbeiter = {name: name_t,
                    mitarb_nr: int,
                    gehalt: real,
                    dienstantritt: datum}
```

Allgemein

Sind $t, f_1, \dots, f_n$ Bezeichner und $typ_1, \dots, typ_n$ Typen, so wird durch

$$\texttt{type } t \texttt{ = } \{f_1 : typ_1 , \dots , f_n : typ_n\}$$

ein neuer Typ t definiert, dessen Werte von der Form

$$\{f_1 = w_1 , \dots , f_n = w_n\}$$

sind, wobei jeweils w_i ein Wert des Typs typ_i ist.

Dieser Typ t heißt Recordtyp oder auch Verbundtyp. Die f_i heißen Felder des Recordtyps.

Abschließendes zu Records

- Mit #f wählt man das Feld f eines Records aus
- Die Reihenfolge der Felder eines Records ist ohne Bedeutung
- Records sind im wesentlich dasselbe wie kartesische Produkte; der Unterschied liegt in der Notation für Auswahl und Konstruktion von Elementen.
- In SML müssen Records nicht unbedingt an eine Typabkürzung gebunden werden:

```
val x = {titel="tschick",ISBN="3871347108"};
val x = {ISBN="3871347108",titel="tschick"} :
                                 {ISBN:string, titel:string}
```

Listen

Ist A eine Menge, so ist A **list** $= \bigcup_{n \geq 0} A^n$ die Menge der Listen über A.
Fasst man A als Alphabet auf, so ist A **list** $= A^*$.
Man notiert Listen als $[a_1; \ldots; a_n]$ anstatt $(a_1, \ldots, a_n)$ oder $a_1 \cdots a_n$.
Die leere Liste wird mit **nil** oder [] bezeichnet.
Ist $a \in A$ und $l = [a_1; \ldots; a_n] \in A$ **list**, so bezeichnet **cons**(a, l) die Liste $[a; a_1; \ldots; a_n]$.
Man schreibt auch $a :: l$ für **cons**(a, l).
Die Verkettung von Listen l_1 und l_2 schreiben wir $l_1 @ l_2$. Es gilt

$$
\begin{aligned}
[] @ l_2 &= l_2 \\
(a :: l) @ l_2 &= a :: (l @ l_2)
\end{aligned}
$$

Beispiele

```
- val x = 17::[];
val x = [17] : int list
- val y = 9 :: 2 :: x;
val y = [9,2,17] : int list
- x = y;
val it = false : bool
- x @ y;
val it = [17,9,2,17] : int list
- [x,y];
val it = [[17],[9,2,17]] : int list list
- [(9,2),(3,5)];
val it = [(9,2),(3,5)] : (int * int) list
- [2,true];
stdIn:22.1-22.9 Error: operator and operand don't agree [literal
```

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 \verb`'a list+ wird polymorpher Typ oder Polytyp genannt, weil der Ausdruck für mehrere Typen steht;
- `list` und `*` sind Typkonstruktoren, die aus gegebenen Typen neue bilden.
- Weitere Beispiele für polymorphe Typen

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

Pattern matching

- Man definiert Funktionen über Listen durch strukturelle Rekursion mit Hilfe von Mustern (Patterns) der Form

```
fun f(nil) =
| f(x :: l) =
```

- Kommt `x` nicht auf der rechten Seite vor, kann es durch _ (Unterstrich, Wildcard) ersetzt werden (analog für `l`)
- Beispiel:

```
- fun proj1 (x, _) = x;
val proj1 = fn : 'a * 'b -> 'a
```

- Muster können auch detaillierter sein, z.B.

```
 fun f(nil) =
| f(x :: nil) =
| f(x :: y :: l) =
```

Grundlegende Funktionen auf Listen

- 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
```

- SML bietet die vordefinierten Funktionen `hd` und `tl` an.

Grundlegende Listenfunktionen II

- 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
- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

Grundlegende Listenfunktionen III

- 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
```

- Umdrehen 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.

Effizientes Umdrehen

- Idee: Füge eine zweite (rechte) Liste als Akkumulator für das Resultat hinzu, womit 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]
0	[]	[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, []);
```

Strukturelle Induktion

- Man beweist Eigenschaften von Listen durch strukturelle Induktion:
  - Um $P(\texttt{l})$ für alle Listen `l` eines Typs `t list` zu zeigen,
    - Zeige `P([])`
    - Zeige `P(a::l)` unter der Annahme `P(l)`
- Spezialfall des allgemeinen Induktionsprinzip für die wohlfundierte Relation: $l_1 R l_2 \iff l_1 = \texttt{tl}(l_2)$.
- Beispiel Man zeige durch strukturelle Induktion, dass für alle `l` und `acc` vom Typ `'a list` gilt:

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

Sortieren durch Einfügen

Eine Liste $[x_1; \ldots; x_n]$ von heißt sortiert, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$.

```
- fun insertel(x, []) = [x]
=   | insertel(x,y::l) = if x <= y then x::y::l
                                  else y::insertel(x,l);
val insertel = fn : int * int list -> int list

- fun inssort [] = []
= | inssort (x::l) = insertel(x, inssort l);}
val inssort = fn : int list -> int list
```

Ist l sortiert, so auch `insertel`(a,l) und es enthält dieselben Elemente wie $a :: l$.
Für beliebiges l ist `inssort`(l) sortiert und enthält dieselben Elemente wie l.

Einschub: Lokale Definitionen: `let` und `local`

- Es bezeichne *defs* eine Folge von Definitionen (`val`, `fun`).
- Wenn *e* ein Ausdruck ist, dann ist auch `let` *defs* `in` *e* `end` ein Ausdruck. Die Definitionen in *defs* dürfen in *e* benutzt werden, sind aber außerhalb nicht sichtbar.

```
- let val x = 9;
      fun f(y) = 2*y
  in f(10)+x end;
val it = 29 : int
- f;
stdIn:34.1 Error: unbound variable or constructor: f
```

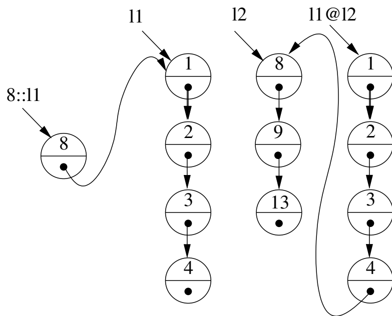
- Ist *defs'* eine weitere Folge von Definitionen, so bewirkt `local` *defs* `in` *defs'* `end`, dass die Definitionen in *defs'* getätigt werden. Die Definitionen in `defs` dürfen dabei benutzt werden, sind von außen aber nicht sichtbar.

Beispiel: Sortieren durch Mischen

```
local
  fun split [] = ([],[])
  |   split [a] = ([a],[])
  | split (a::b::l) = let val (l1,l2) = split l
                      in (a::l1,b::l2) end;
  fun merge (l,[]) = l
  |   merge ([],l)=l
  |   merge(a::l,b::k) = if a<=b then
            a::merge(l,b::k)else b::merge(a::l,k)
in
  fun mergesort [] = []
  | mergesort [a]=[a]
  | mergesort l = let val (l1,l2)=split l in
                        merge(mergesort l1,mergesort l2) end
end;
```

Anmerkungen zur Laufzeit

- Listen sind im Rechner als Ketten von Einträgen gespeichert und können vom Kopf her angesehen werden
- Ein "cons", also `x::l` benötigt konstante Zeit
- Abgleich gegen ein Muster der Form `kopf::rumpf` benötigt konstante Zeit (ein paar Zykeln)
- Eine Verkettung `l1 @ l2` benötigt Zeit proportional zur Länge von `l1`.



NB: Diese Folie ist nicht prüfungsrelevant

Anmerkungen zur Laufzeit

NB: Diese Folie ist nicht prüfungsrelevant

- Zeitaufwand "reverse":
  $T_{\texttt{reverse}}(n+1) = T_{\texttt{reverse}}(n) + O(n) \leadsto T_{\texttt{reverse}(n)} = O(n^2)$
- Zeitaufwand "rev_aux": $T_{\texttt{rev_aux}}(m+1, n) = T_{\texttt{rev_aux}}(m, n) + O(1) \leadsto T_{\texttt{rev_aux}}(m, n) = O(n)$
- Zeitaufwand
  $T_{\texttt{ins_el}}(n+1) = T_{\texttt{ins_el}}(n) + O(1) \leadsto T_{\texttt{inssort}}(n) = O(n)$
- Zeitaufwand
  $T_{\texttt{inssort}}(n+1) = T_{\texttt{inssort}}(n) + O(n) \leadsto T_{\texttt{inssort}}(n) = O(n^2)$
- Zeitaufwand $T_{\texttt{mergesort}}(n) = 2T_{\texttt{mergesort}}(n/2) + O(n) \leadsto T_{\texttt{mergesort}}(n) = O(n \log(n))$