SML-Kurs, SS 2003

Hans Leiß

30. April 2003

Inhaltsverzeichnis

Der Typ `bool`

Der Typ der Boole'schen Wahrheitswerte ist vordefiniert:

$\langle bool \rangle \equiv$

```
datatype bool = false | true
val not : bool -> bool
```

Der Wert `val` ist die Negation, die man sich durch

$\langle bool \rangle{+}\equiv$

```
fun not(true) = false
  | not(false) = true
```

deklariert denken kann. Weitere Funktionen kann man sich selber definieren:

$\langle bool \rangle{+}\equiv$

```
fun xor(true,true) = false
  | xor(true,false) = true
  | xor(false,true) = true
  | xor(false,false) = false
```

Für Konjunktion und Disjunktion gibt es `andalso` und `orelse`. Sie werden infix geschrieben (und sind keine Werte, sondern Abkürzungen). Vorsicht: `and` ist ein reserviertes Schlüsselwort.

Die Auswertung des Ausdrucks

```
(true andalso (xor(false,true)));
```

bewirkt eine *Wertdeklaration*, `val` *Name* `=` *Ausdruck*,

`val it = true : bool`,

d.h. SML berechnet den Wert `true` und bindet ihn an `it`, einen Wertnamen, der mit dem zuletzt berechneten Wert belegt ist.

Funktionsdeklarationen

Die Auswertung der Funktionsdeklaration

$\langle xor \rangle \equiv$

```
fun xor(true,true) = false
  | xor(true,false) = true
  | xor(false,true) = true
  | xor(false,false) = false
```

durch den Compiler ergibt:

```
val xor = fn : bool * bool -> bool
```

Daran sieht man:

1. Für SML sind Funktionen Werte (durch `val` markiert).
2. Mit `bool` sind auch das *kartesische Produkt* `bool * bool` und der Raum `bool * bool -> bool` der *Funktionen* von `bool * bool` nach `bool` Typen.
3. SML erkennt, welchen Typ die deklarierte Funktion hat.
4. Werte, die Funktionen sind, werden nicht gedruckt, sondern durch `fn` angedeutet.
5. Durch `fun` wird der Wert an den Namen `xor` gebunden.

Bem: Will man `xor` infix schreiben, so erklärt man

$\langle xor \rangle + \equiv$

```
infix xor;
```

und kann dann z.B. `(true xor false)` schreiben.

Anonyme Funktionen

In SML muß eine Funktion nicht an einen Namen gebunden werden. Der *Funktionsausdruck*

⟨*xor*⟩+≡

```
      fn (true,true) => false
       | (true,false) => true
       | (false,true) => true
       | (false,false) => false
```

hat als Wert (extensional) dieselbe Funktion, die durch die Deklaration von `xor` entsteht, aber sie erhält keinen Namen.

Die frühere *Funktionsdeklaration*

⟨⟩≡

```
    fun xor(true,true) = false | ...
      | xor(false,false) = false
```

ist eine Abkürzung für die *Wertdeklaration*

⟨⟩+≡

```
     val xor = fn (true,true) => false | ...
                | (false,false) => false
```

Allgemein haben *Funktionsausdrücke* die Form

`fn` *Regel* `|` ... `|` *Regel*

und jede Regel hat die Form: *Muster* `=>` *Ausdruck* .

Einfache Muster *pat* sind Variable und Konstante, z.B. `x`, `27`. Komplexe Muster sind z.B. Paare `(`*pat*`,`*pat'*`)` von Mustern.

Der Typ `order`

Noch einfacher ist der Typ `order` mit seinen drei Werten:

⟨*order*⟩≡

```
datatype order = LESS | EQUAL | GREATER
```

Er dient dazu, Ordnungen auf einem Typ `t` als Funktionen

```
compare : t * t -> order
```

zu schreiben, z.B.:

⟨*cmp intsq*⟩≡

```
fun compare ((n,m),(k,l)) =
  if n < k orelse (n = k andalso m < l)
    then LESS
  else if (n = k andalso m = l)
         then EQUAL
       else GREATER
```

Damit hat man

```
- compare ((2,3),(1,7));
val it = GREATER : order
```

Ohne `order` bräuchte man mindestens zwei Relationen

```
< : t * t -> bool,   <= : t * t -> bool
```

Vorsicht: `<,<=,=,>,>=` sind vordefinierte Bezeichner (infix).

Der Typ `unit`

Er hat nur ein Element, das als Wert von Seiteneffekten dient:

⟨*unit*⟩≡

```
datatype unit = ()
```

Der Typ `t list` der homogenen Listen

Der Typ `(t list)` der *Listen von Werten des Typs* `t` existiert zu jedem Typ `t`.

Ausdrücke für Listen lassen sich aus Ausdrücken `e` für Werte vom Typ `t` mit den beiden *Wertkonstruktoren*

⟨*list*⟩≡
```
     nil : 'a list,
      :: : 'a * ('a list) -> ('a list)
```

aufbauen, wobei für `::` die Infix-Schreibweise festgelegt ist.

Hier ist `'a` eine *Typvariable*: die Konstruktoren funktionieren für jeden beliebigen Typ `t`.

⟨*expl*⟩≡
```
  - val intlist = 3 :: (6 :: (1 :: nil));
  val intlist = [3,6,1] : int list
  - val boollist = (true :: []);
  val boollist = [true] : bool list
```

Die Antwort wird in der lesbareren Schreibweise [a_1,...,a_m] wiedergegeben, die auch zur Eingabe benutzt werden kann.

Auf Listen sind einige Funktionen vordefiniert, darunter die Umkehrung einer Liste und die Verkettung zweier Listen:

⟨*list*⟩+≡
```
     rev : 'a list -> 'a list
       @ : 'a list * 'a list -> 'a list
```

⟨*expl*⟩+≡
```
  - (rev [1,2,3]) @ [4,5,6,7];
  val it = [3,2,1,4,5,6,7] : int list
```

Funktionen auf Listen

Weitere eingebaute Funktion zum Umgang mit Listen sind

⟨*list*⟩+≡

```
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val map : ('a -> 'b) -> 'a list -> 'b list
    val app : ('a -> unit) -> 'a list -> unit
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Die Bedeutung der ersten vier sieht man an folgenden Beispielen:

⟨*expl*⟩+≡

```
  - hd [1,2,3,4];
  val it = 1 : int
  - tl [1,2,3,4];
  val it = [2,3,4] : int list
  - map (fn x => x+3) [1,2,3,4];
  val it = [4,5,6,7] : int list
  - app print ["Aa","bb ","CCC","\n"];
  Aabb CCC
  val it = () : unit
```

Die Bedeutung der Links- und Rechts*faltungen* `foldl` und `foldr` ist wie folgt:

⟨*foldr*⟩≡

```
  fun foldl f e [] = e
    | foldl f e (x::xs) = foldl f (f (x,e)) xs
  fun foldr f e [] = e
    | foldr f e (x::xs) = f (x, foldr f e xs)
```

Beispiele für Faltungen

Der Typ der Faltungen `foldl,foldr` ist

$\langle$*fold*$\rangle\equiv$

```
 ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Schreiben wir f zwischen die Argumente, so erhalten wir

$\langle$*foldl*$\rangle\equiv$

```
foldl f e [a,b,c] = foldl f (a f e) [b,c]
                  = foldl f (b f (a f e)) [c]
                  = foldl f (c f (b f (a f e))) []
                  =         (c f (b f (a f e)))
foldr f e [a,b,c] = a f (b f (c f e))
```

Daher können wir die Umkehrung einer Liste mit Hilfe von `op:: : 'a * 'a list -> 'a list` für `f` so definieren:

$\langle$*reverse*$\rangle\equiv$

```
- val reverse = foldl op:: ([] : int list);
val reverse = fn : int list -> int list
- reverse [0,1,2,3,4];
val it = [4,3,2,1,0] : int list
```

Hier ist `op::` die Präfixschreibweise von `::` (= `cons`).

Analog für die Summe und das Produkt einer Zahlenliste:

$\langle$*sum*$\rangle\equiv$

```
- foldr op+ 0 [1,2,3,4];
val it = 10 : int
- foldr op* 1 [1,2,3,4];
val it = 24 : int
```

Akkumulatoren

Oft erhält man kein effizientes Programm, wenn man eine Listenfunktion direkt durch Rekursion über ein Listenargument definiert. Bei

$\langle$*reverse, direkt*$\rangle\equiv$

```
fun reverse (x::xs) = (reverse xs) @ [x]
  | reverse [] = [];
```

wird die Hilfsfunktion `@` (= Listenverkettung) in jedem Rekursionsaufruf von `reverse` aufgerufen, und `ys @ zs` braucht mindestens `|ys|` Rechenschritte. Daher braucht `reverse xs` etwa $|\texttt{xs}|^2$ Schritte.

In vielen Fällen kann man Zwischenergebnisse auf einer *Akkumulator*liste sammeln und dadurch Rekursionsaufrufe vermeiden. Die eigentliche Funktion ruft eine Hilfsfunktion mit passend initialisierter Akkumulatorliste auf.

$\langle$*reverse*$\rangle{+}\equiv$

```
fun reverse xs = rev xs []
and rev (x::xs) acc = rev xs (x::acc)
  | rev [] acc = acc;

- rev [3,4] [2,1];
val it = [4,3,2,1] : int list
- rev [1,2,3,4] [];
val it = [4,3,2,1] : int list
```

Durch `and` werden beide Funktionen *simultan* definiert. (Wir hätten hier die Hilfsfunktion `rev` vorher auch allein definieren können.)

Automatische Typisierung und Typfehler

SML achtet darauf, daß Ausdrücke *typkorrekt* gebildet sind; sonst werden sie gar nicht ausgewertet.

1. Jeder Ausdruck wird auf Typkorrektheit überprüft; die Typen der eingebauten Wertkonstruktoren werden benutzt, um jedem Teilausdruck einen Typ zu geben.

2. Eine Anwendung `f(e)` ist typkorrekt, wenn `f : s -> t` und `e : s` ist.

3. Ein Ausdruck kann als Typ ein Typ*schema* haben.

Listen sind in SML *homogen*, d.h. alle Elemente einer Liste müssen denselben Typ haben. Ist das nicht der Fall, wird ein *Typfehler* gemeldet:

⟨*expl*⟩+≡

```
[3,true];
- Error: operator and operand don't agree [literal]
  operator domain: int * int list
  operand:         int * bool list
  in expression:
    3 :: true :: nil
```

Da das erste Element `3` den Typ `int` hat, wird der (schematische) Argumenttyp des Operators `::` von

`'a * 'a list`   zu   `int * int list`

spezialisiert. Die Restliste `true :: nil` = `[true]` hat den Typ `bool list`. Also hat das Argument `(3,[true])` den Typ `int * bool list`, was nicht zum Argumenttyp von `::` paßt.