Programmierung und Modellierung

Methodisches Programmieren: Ausnahmebehandlung und Strukturen

Martin Wirsing

in Zusammenarbeit mit

Moritz Hammer

SS 2009

Inhalt

9. Ausnahmebehandlung in SML
   1. Der vordefinierte Typ `exn`
   2. Ausnahmekonstruktoren
   3. Ausnahmen auslösen (oder werfen)
   4. Ausnahmen behandeln (oder einfangen)
   5. Prinzip der Auswertung von raise- und handle-Ausdrücken
10. Module in SML
   1. Strukturen
   2. Signaturen
   3. Struktursichten und Angleich an eine Signatur
   4. Grundlegende Rechenstrukturen

9.1 Der vordefinierte Typ exn

- SML enthält einen vordefinierten Typ `exn` für **Ausnahmen** (exceptions).
- Die Werte dieses Typs heißen **Ausnahmewerte** (exception values)
- Besonderheit dieses Typs:
  - Ein Programmierer kann dem Typ neue (Wert-)Konstruktoren hinzufügen, die **Ausnahmekonstruktoren** (exception constructors) genannt werden.
  - (Dies ist für keinen anderen vordefinierten Typ möglich)

9.2 Ausnahmekonstruktoren

- Ein Ausnahmekonstruktor namens `A` wird wie folgt deklariert:
  - Konstanter Ausnahmekonstruktor:

    ```
    exception A;
    ```
  - Ausnahmekonstruktor mit Parameter vom Typ `t`:

    ```
    exception A of t;
    ```
- Beispiel
  - `exception illegal_expression;`
  - `exception negative_argument of int;`
- **Bemerkung**

  Ausnahmen können (mit `let`) auch lokal deklariert werden. Davon wird aber abgeraten, da dies schwer verständlich ist.

9.3 Ausnahmen auslösen (oder werfen)

- Eine Ausnahme `A` vom Typ `exn` kann mit folgendem Ausdruck **ausgelöst** (oder geworfen) werden:
  - `raise A`
- **Beispiel** (Deklaration und Verwendung einer konstanten Ausnahme):

```
- exception negative_integer;
exception negative_integer
- fun factorial x =
    if x < 0 then raise negative_integer
    else   if x = 0 then 1
           else x * factorial(x - 1);
val factorial = fn : int -> int
- factorial ~4;
uncaught exception negative_integer
```

Ausnahmen auslösen (oder werfen)

- **Beispiel** (Deklaration und Verwendung einer Ausnahme mit Parameter):

```
- exception negative_argument of int;
exception negative_argument of int
- fun fac x = if x < 0
    then raise negative_argument(x)
    else   if x = 0 then 1
           else x * fac(x - 1);
val fac = fn : int -> int
- fac ~4;
uncaught exception negative_argument
```

- Der Ausnahmeparameter ~4 wird hier **nicht** angezeigt, da SML/NJ (wie in anderen Fällen) die Ausgabe verkürzt und damit die volle Struktur nicht sichtbar wird.

Ausnahmen auslösen (oder werfen)

- Ausnahmen unterbrechen die normale Auswertung und werden „nach außen“ weitergeleitet:

  Liefert die Auswertung eines Teilausdruckes T eines zusammengesetzten Ausdrucks B eine Ausnahme A als Ergebnis, so wird diese Ausnahme A als Ergebnis der Auswertung des Gesamtausdrucks B geliefert, es sei denn, A wird von einem Ausnahmebehandler eingefangen.

- Beispiel:

```
- fun is_even x = (x mod 2 = 0);
val is_even = fn : int -> bool

- is_even(factorial ~4);
uncaught exception negative_integer
```

  - Aufgrund der applikativen Auswertungsreihenfolge führt die Auswertung von `is_even(factorial ~4)` zunächst zur Auswertung der Teilausdrucks `factorial ~4` und liefert die Ausnahme `negative_integer`, die als Ergebnis des Gesamtausdrucks zurückgegeben wird.

9.4 Ausnahmen behandeln (oder einfangen)

- Ein **Ausnahmebehandler** (engl. exception handler) ist eine Art Funktion, die nur Parameter vom Typ `exn` haben kann. Er hat im einfachsten Fall die Gestalt `handle A => C`.
- Ein Ausdruck `exp` der Form `B handle A => C` wird folgendermaßen ausgewertet:
  - Liefert die Auswertung von `B` (oder eines Teilausdrucks `T` in `B`) die Ausnahme `A`, so wird der Ausnahmebehandler `handle A => C` wirksam. Dann wird `C` ausgewertet und der Wert von `C` als Wert von `exp` geliefert. (Die Ausnahme `A` wird also nicht “nach oben” weitergereicht.)
  - Liefert die Auswertung von `B` etwas anderes als die Ausnahme `A`, so wird der Wert von `B` als Wert von `exp` geliefert. (Der Ausnahmebehandler `handle A => C` hat dann keine Wirkung.)
- Anschaulich kann man sagen, dass ein Teilausdruck eine Ausnahme in Richtung seiner umfassenden Ausdrücke wirft. Die Ausnahme wird von den umfassenden Ausdrücken einfach durchgelassen, bis sie von einem Ausnahmebehandler eingefangen wird.

Ausnahmen behandeln (oder einfangen)

- Das Beispiel der Fakultätsfunktion `factorial` kann wie folgt mit einem Behandler für die Ausnahme `negative_integer` ergänzt werden:

```
- exception negative_integer;
exception negative_integer
- fun factorial x =
    (if x < 0 then raise negative_integer
     else if x = 0 then 1
                   else x * factorial(x - 1)
    )
    handle negative_integer => factorial(~x);
val factorial = fn : int -> int
- factorial ~4;
val it = 24 : int
```

- Die auftretende Ausnahme `negative_integer` wird von dem Behandler eingefangen, was zur Auswertung von `factorial(~(~4))` führt.

Ausnahmen behandeln (oder einfangen)

- Die Ausnahmebehandlung kann auch anders erfolgen:

```
- exception negative_argument of int;
exception negative_argument of int
-  fun fac x =
    if x < 0 then raise negative_argument(x)
    else   if x = 1 then 1
           else x * fac(x - 1);
val fac = fn : int -> int
- fac ~4 handle negative_argument(y) => fac(~y);
val it = 24 : int
```

Ausnahmen behandeln (oder einfangen)

- In beiden Beispielen tritt in den gleichen Fällen eine Ausnahme auf und ihre Behandlung führt auch zu den gleichen Ergebnissen:
  - Im ersten Beispiel wird die Ausnahme im Rumpf der Funktion eingefangen, d.h. im Geltungsbereich des formalen Parameters `x` der Funktion `factorial`.
  - Im zweiten Beispiel ist der Behandler außerhalb des Geltungsbereiches des formalen Parameters `x` der Funktion `fac`. Der Wert `~4` des aktuellen Parameters des Aufrufes wird über die einstellige Ausnahme `negative_argument` an den Behandler weitergereicht.
- Ein Behandler wird i.Allg. mittels Pattern Matching definiert:

```
handle <Muster1> => <Ausdruck1>
|      <Muster2> => <Ausdruck2>
.
.
.
|      <Mustern> => <Ausdruckn>
```

9.5 Prinzip der Auswertung von raise- und handle-Ausdrücken

- Bei der Auswertung eines zusammengesetzten Ausdrucks werden zunächst die Teilausdrücke ausgewertet.
- Ist einer der dabei ermittelten Werte ein Ausnahmewert (auch bezeichnet als “Ausnahmepaket”), wird die weitere Auswertung der Teilausdrücke abgebrochen und der Ausnahmewert als Wert geliefert.
- Dabei wird der Ausnahmewert als Ergebnis der Auswertungen sämtlicher umfassender Ausdrücke weitergereicht, bis ein Behandler gefunden wird.
- Bei der Auswertung eines Ausdrucks

  `<Ausdruck1> handle <Muster> => <Ausdruck2>`

  wird zunächst `<Ausdruck1>` ausgewertet.
  - Ist der Wert **kein** Ausnahmewert, wird dieser Wert geliefert.
  - Ist der Wert ein Ausnahmewert `ex`, erfolgt Pattern Matching zwischen `<Muster>` und dem Ausnahmewert `ex`.
  - Bei Erfolg wird `<Ausdruck2>` ausgewertet und dessen Wert geliefert, bei Misserfolg wird der Ausnahmewert `ex` geliefert.

Prinzip der Auswertung von raise- und handle-Ausdrücken

- **Beispiel:**

```
- exception negative_zahl;
- fun f x =
    if x = 0 then true
    else   if x > 0 then f(x - 1)
           else raise negative_zahl;
val f = fn : int -> bool
- fun g true = "wahr"
  |  g false = "falsch";
val g = fn : bool -> string
- g(f ~3) handle negative_zahl => "Fehler";
val it = "Fehler" : string
```

Prinzip der Auswertung von raise- und handle-Ausdrücken

- Die Auswertung des letzten Ausdrucks kann unter Anwendung des Substitutionsmodells wie folgt erläutert werden:

```
(* "Zunächst Auswertung von g(f ~3)": *)
g(f ~3) handle negative_zahl => "Fehler"
                                    =[Einsetzen Rumpf f]
g(if ~3 = 0 then true
  else if ~3 > 0 then f(~3 - 1)
       else raise negative_zahl )
handle negative_zahl => "Fehler"   =[Auswertung ~3 = 0]
g(if false then true
   else if ~3 > 0 then f(~3 - 1)
        else raise negative_zahl )
handle negative_zahl => "Fehler"  =[Auswertung if]
g( if ~3 > 0 then f(~3 - 1)
    else raise negative_zahl )
handle negative_zahl => "Fehler"  =[Auswertung ~3 > 0]
```

Prinzip der Auswertung von raise- und handle-Ausdrücken

```
g( if false then f(~3 - 1)
   else raise negative_zahl )
handle negative_zahl => "Fehler“   =[Auswertung if]
g( raise negative_zahl )
handle negative_zahl => "Fehler“ =[Auswertung raise]
g( negative_zahl )
handle negative_zahl => "Fehler“ =[Hochreichen Ausnahme]
negative_zahl          (* = Wert von g(f ~3) *)
handle negative_zahl => "Fehler“
                       =[Pattern Match erfolgreich]
"Fehler"
```

Prinzip der Auswertung von raise- und handle-Ausdrücken

- Ausnahmen sind ihrer Natur nach eng an die Auswertungsreihenfolge gekoppelt.
- Mit SML-Ausnahmen kann man damit z.B. herausfinden, in welcher Reihenfolge Teilausdrücke ausgewertet werden:

```
- exception a;
exception a
- exception b;
exception b
- ((raise a) + (raise b))
  handle b => 2 | a => 1;
val it = 1 : int
```

- Der linke Teilausdruck wird also zuerst ausgewertet.

Vordefinierte Ausnahmen von SML

- Einige Ausnahmen sind in SML vordefiniert, z.B. die Ausnahmen `Match` und `Bind`, die bei Fehlern während des Pattern Matching erhoben werden.
- Die Standardbibliothek von SML beschreibt die vordefinierten Ausnahmen.

10. Module in SML

- Module ermöglichen die **hierarchische Strukturierung größerer Programme**. Die Modulbegriffe von SML heißen:
  - **Struktur,**
  - **Signatur,**
  - **Funktor**.
- In einer **SML-Struktur** können Werte (Funktionen sind in SML auch Werte), Typen und Ausnahmen deklariert werden.
- Eine **SML-Signatur** beschreibt eine Menge von Namen mit den dazu gehörigen Typen.
  - Damit ist eine Signatur eine Art **Typ einer Struktur**. Die Signatur beschreibt, welche Namen die Struktur deklariert, ohne deren Implementierungen preiszugeben.
- Ein **SML-Funktor** ist ein parametrisiertes Modul. Ein SML-Funktor kann als eine Art Funktion angesehen werden, womit Strukturen auf Strukturen abgebildet werden.

Module in SML

- Die folgende Ähnlichkeit gilt zu objekt-orientierten Begriffen:
  - Strukturen entsprechen Klassen.
  - Signaturen entsprechen Schnittstellen (Interfaces).
  - Funktoren entsprechen generischen Klassen.
- Im Folgenden werden SML-Strukturen und SML-Signaturen näher erläutert.

10.1 SML-Strukturen

- Eine **SML-Struktur** `Struc` besteht aus einer Menge von Deklarationen und hat die Form:

```
structure Struc =
  struct
    . . .
  end;
```

- **Beispiel:** Struktur zur Definition eines Typs "komplexe Zahlen":

```
structure Complex =
  struct
    type t = real * real;
    val zero = (0.0, 0.0) : t;
    fun sum ((x1,y1):t, (x2,y2):t) = (x1 + x2, y1 + y2) : t;
    fun difference((x1,y1):t, (x2,y2):t) = (x1-x2, y1-y2) : t;
    fun product ((x1,y1):t, (x2,y2):t) =
            (x1 * x2 - y1 * y2, x1 * y2 + x2 * y1) : t;
    fun reciprocal((x,y) : t) =
      let val r = x * x + y * y
        in (x/r, ~y/r) : t end;
    fun quotient (z1 : t, z2 : t) = product(z1, reciprocal z2)
  end;
```

SML-Strukturen

- In einer Struktur deklarierte Namen sind außerhalb der Struktur nicht (direkt) sichtbar; z.B.:

```
- reciprocal(1.0, 0.0);
Error: unbound variable or constructor: reciprocal
```

- Aber jede Struktur bildet einen **Namensraum**:
  - Ein Name N, der in einer Struktur S deklariert ist, kann außerhalb von S als S.N verwendet werden; z.B.:

```
- Complex.reciprocal(1.0, 0.0);
val it = (1.0,0.0) : Complex.t
```

  - Dient eine Struktur S zur Definition eines Typs t, so wird dieser Typ
    - in der Struktur als `t`,
    - außerhalb dieser Struktur als `S.t`

    bezeichnet:

```
- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t
- Complex.product(i, i);
val it = (~1.0,0.0) : Complex.t
```

10.2 SML-Signaturen

- Ein **SML-Signatur** SIG hat die Form

```
signature SIG =
  sig
   type t1; …
   val n1 : typ1; …
  end
```

- Die Ausdrücke einer Signatur, die zwischen den reservierten Wörtern `sig` und `end` vorkommen, heißen **(Signatur-) Spezikationen**.
  - Zum Beispiel ist `type t` eine Spezifikation.
- Es ist üblich (aber nicht von SML erzwungen), dass
  - die Namen von Strukturen mit einem Großbuchstaben beginnen und
  - die Namen von Signaturen ganz aus Großbuchstaben bestehen.
- Eine Signatur kann vom SML-System aus einer Strukturdeklaration ermittelt oder vom Programmierer selbst deklariert werden.

SML-Signaturen

- **Beispiel:** Signatur für Strukturen, die einen Typ t sowie die grundlegenden arithmetischen Operationen über t implementieren:

```
- signature ARITHMETIC =
    sig
        type t
        val zero : t
        val sum : t * t -> t
        val difference : t * t -> t
        val product : t * t -> t
        val reciprocal : t -> t
        val quotient : t * t -> t
    end;
```

- **Bemerkung**

  Um den Unterschied zwischen Spezifikationen und Deklarationen zu unterstreichen, darf das reservierte Wort `fun` in einer Signatur nicht vorkommen, sondern nur das reservierte Wort `val` wie etwa:

```
    val sum : t * t -> t
```

SML-Signaturen

- Die Signatur ARITHMETIC kann in sogenannten **Signatur-Constraints** verwendet werden, wenn eine Struktur definiert wird, die alle in der Signatur spezifizierten Komponenten deklariert:

```
- structure Rational : ARITHMETIC =
    struct
      type t = int * int;
      val zero = (0, 1) : t;
      fun sum ((x1,y1):t, (x2,y2):t) = (x1*y2+x2*y1, y1*y2) :t;
      fun difference((x1,y1):t, (x2,y2):t) = (x1*y2-x2*y1, y1*y2) :t;
      fun product ((x1,y1):t, (x2,y2):t) = (x1 * x2, y1 * y2) : t;
      fun reciprocal((x,y) : t) = (y,x) : t;
      fun quotient (z1 : t, z2 : t) = product(z1, reciprocal z2)
    end;
structure Rational : ARITHMETIC
```

- **Bemerkung**

  Die Typabkürzungen `Rational.t` und `Complex.t` bezeichnen unterschiedliche Typen,

  nämlich `int * int` und `real * real`,

  obwohl beide Strukturen dieselbe Signatur `ARITHMETIC` haben.

10.3 Angleich einer Struktur an eine Signatur: Struktursichten

- Eine Struktur `Struk` kann an eine Signatur `SIG` angeglichen werden, wenn alle Komponenten, die in `SIG` spezifiziert werden, in `Struk` deklariert sind.
- Wenn eine Struktur mehr Komponenten enthält als eine Signatur fordert, können eingeschränkte **Sichten (views)** auf die Struktur definiert werden.
- **Beispiel** Die folgende Signatur spezifiziert nur einen Teil der Namen, die in der Struktur `Complex` deklariert sind:

```
- signature RESTRICTED_ARITHMETIC =
    sig
        type t
        val zero : t
        val sum : t * t -> t
        val difference : t * t -> t
    end;
```

Angleich einer Struktur an eine Signatur: Struktursichten

- Mit der Signatur RESTRICTED ARITHMETIC kann eine Einschränkung der Struktur Complex definiert werden, die **nur** die in RESTRICTED ARITHMETIC vorkommenden Namen zur Verfügung stellt.

```
- structure RestrictedComplex : RESTRICTED_ARITHMETIC =
  Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC
- val i = (0.0, 1.0) : RestrictedComplex.t;
val i = (0.0,1.0) : Complex.t
- RestrictedComplex.sum(RestrictedComplex.zero, i);
val it = (0.0,1.0) : Complex.t
- RestrictedComplex.product(RestrictedComplex.zero, i);
Error: unbound variable or constructor: product in path
RestrictedComplex.product
- Complex.product(RestrictedComplex.zero, i);
val it = (0.0,0.0) : Complex.t
```

- In der Deklaration ist “ : RESTRICTED_ARITHMETIC” ein sogenanntes **Signatur-Constraint.** Mit Signatur-Constraints kann man **information hiding** erreichen.

10.4 Grundlegende Rechenstrukturen

- Im Folgenden behandeln wir die grundlegenden Rechnestrukturen
  - Listen
  - Keller mit Implementierung durch Listen bzw. einen rekursiven Datentyp
  - Mengen (mit Implementierung durch Listen)
- Darüberhinaus werden Signaturen mit `eqtype`- und `datatype`-Spezifikationen eingeführt

Keller (Stack)

- Kellerstrukturen sind lineare Strukturen, die nach dem LiFo-Prinzip („last in – first out“) arbeiten.
- Signatur

```
signature STACKSIG =
  sig
  type ‘a stack
  exception Stack_empty
  val empty_stack: ‘a stack
  val push: ‘a * ‘a stack -> ‘a stack
  val pop:  ‘a stack -> ‘a stack
  val top:  ‘a stack -> ‘a
  val is_empty:  ‘a stack -> bool
end;
```

F.L. Bauer
*1924,
Diss 1952 LMU
Entwickler von Algol 60,
Patent auf Kellerprinzip,
„Vater“ der dtsch. Informatik

Keller (Stack): Implementierung durch rekursiven Datentyp

```
structure Stack_By_Rec =
  struct
  datatype ‘a stack =
    empty_stack | push of ‘a * ‘a stack
  exception Stack_empty;
  fun pop empty_stack = raise Stack_empty
  |   pop (push (x, s)) = s;
  fun top empty_stack = raise Stack_empty
  |   top (push (x, s)) = x;
  fun is_empty empty_stack = true
  |   is_empty (push (x, s)) = false;
end;
```

datatype-Spezikationen in SML-Signaturen

- In einer SML-Signatur kann die Spezifikation eines Typs `t` auch in Form einer `datatype`-Deklaration erfolgen.
- In diesem Fall ist nicht nur spezifiziert, dass `t` ein Typ ist, sondern auch, welche Wertkonstruktoren der Typ `t` hat.
- **Beispiel:** Signatur `LIST` der Struktur `List` aus der Standardbibliothek von SML (nächste Folie)

Signature LIST of Standard Structure List

```
signature LIST =
 sig
  datatype 'a list = :: of 'a * 'a list | nil
  exception Empty
  val null : 'a list -> bool
  val hd : 'a list -> 'a
  val tl : 'a list -> 'a list
  val length : 'a list -> int
  val rev : 'a list -> 'a list
    . . .
  val @ : 'a list * 'a list -> 'a list
  val filter : ('a -> bool) -> 'a list -> 'a list
  val all : ('a -> bool) -> 'a list -> bool
 end
```

Implementierung der Kellersignatur durch Listen

```
structure Stack_By_List =
 struct
  datatype ‘a stack = Stack of ‘a list;
  exception Stack_empty;
  val empty_stack = Stack nil;
  fun push(x, Stack l) = Stack(x::l);
  fun pop (Stack nil) = raise Stack_empty
  |   pop (Stack(x::l)) = Stack l;
  fun top (Stack nil) = raise Stack_empty
  |   top (Stack(x::l)) = x;
  fun is_empty (Stack nil) = true
  |   is_empty (Stack(x::l)) = false;
end;
```

Eqtype-Spezikationen in SML-Signaturen

- In einer SML-Signatur kann die Spezifikation eines Typs `t` mit dem reservierten Wort `eqtype` statt `type` eingeführt werden, wenn die Gleichheit über `t` definiert sein muss.
- Funktionstypen (also mit dem Typkonstruktor ->) können natürlich nicht mit `eqtype` deklariert werden, da es keine Gleichheit über Funktionstypen gibt.
- **Beispiel** Signatur der (endlichen) Mengen

```
signature SETSIG =
 sig
  type ''a set
  val empty_set: ''a set
  val insert: ''a -> ''a set -> ''a set
  val delete: ''a -> ''a set -> ''a set
  val is_elem: ''a -> ''a set -> bool
  val union: ''a set -> ''a set -> ''a set
  val intersect: ''a set -> ''a set -> ''a set
  val eq_set: ''a set -> ''a set -> bool
 end
```

Implementierung vom Mengen durch ungeordnete Listen

```
structure Set_List: SETSIG =
 struct
   fun mem nil e = false                (*Hilfsfunktion*)
   |   mem (x::l) e = (x=e) orelse (mem l e);

   datatype ''a set = Set of ''a list;
   val empty_set = Set nil;
   fun is_elem e (Set l) = mem l e;
   fun insert e (Set l) = Set (e::l);
   fun union (Set l1) (Set l2) = Set(l1 @ l2);

   fun delete e (Set l)=
         Set( List.filter (fn y => not(e=y)) l )
   fun intersect (Set l1) (Set l2) =
         Set((List.filter (mem l1) l2));
   fun eq_set (Set l1) (Set l2) =
         (List.all (fn x => mem l1 x) l2) andalso
         (List.all (fn x => mem l2 x) l1);
 end;
```

- **Übung:** Implementierung von Mengen durch Suchbäume

Beispiele Set_List

```
- val s0 =
   Set_List.insert 3 (Set_List.insert 5
   (Set_List.insert 7  (Set_List.insert 3 Set_List.empty_set)));
val s0 = Set [3,5,7,3] : int Set_List.set
- val s1 =  Set_List.insert 4  (Set_List.insert 3
                        Set_List.empty_set);
val s1 = Set [4,3] : int Set_List.set
- val s2 = Set_List.union s0 s1;
val s2 = Set [3,5,7,3,4,3] : int Set_List.set
-  val s3 = Set_List.intersect s0 s1;
val s3 = Set [3] : int Set_List.set;
-  val s4 = Set_List.delete 3 s3;
val s4 = Set [] : int Set_List.set
-  val s5 = Set_List.eq_set s0 s1;
val s5 = false : bool
-  val s6 =  Set_List.insert 3  (Set_List.insert 4
                (Set_List.insert 3 Set_List.empty_set));
val s6 = Set [3,4,3] : int Set_List.set
-  Set_List.eq_set s6 s1;
val s6 =true : bool
```

Implementierung vom Mengen durch ungeordnete Listen ohne Wiederholungen

```
structure Set_List: SETSIG =
 struct
   fun mem nil e = false          (*Hilfsfunktion*)
   |   mem (x::l) e = (x=e) orelse (mem l e)

   datatype “a set = Set of “a list;
   val empty_set = Set nil;
   fun is_elem e (Set l) = mem l e;
   fun insert e (Set l) =
         if mem l e then Set l else Set (e::l);
   fun intersect (Set l1) (Set l2) =
         Set((filter (mem l1) l2));
   fun delete e (Set l)=
         Set( filter (fn y => not(e=y)) l )
   fun union (Set l1) (Set l2) =
         Set((filter (not o (mem l2)) l1) @ l2);
   fun eq_set (Set l1) (Set l2) =
         (forall (fx x => mem l1 x) l2) andalso
         (forall (fx x => mem l2 x) l1);
 end;
```

Implementierung vom Mengen durch ungeordnete Listen ohne Wiederholungen

- Es ändern sich nur die Implementierungen von `insert` und `union`:

```
fun insert e (Set l) =
  if mem l e then Set l else Set (e::l);
fun union (Set l1) (Set l2) =
  Set((filter (not o (mem l2)) l1) @ l2);
```

Zusammenfassung

- Ausnahmebehandlung
  - Ausnahmen haben in SML den vordefinierten Typ **exn** und werden mit Hilfe von konstanten oder einstelligen **Ausnahmekonstruktoren** spezifiziert.
  - Eine Ausnahme A wird ausgelöst durch `raise A` und kann durch einen Ausdruck der Form
    - handle A1 => C1 | … | An => Cn

    behandelt werden.
- Module ermöglichen die **hierarchische Strukturierung größerer Programme**.
  - Eine **SML-Struktur** deklariert eine Menge von Werten, Typen und Ausnahmen.
  - Eine **SML-Signatur** spezifiziert eine Menge von Namen mit den dazu gehörigen Typen, ohne deren Implementierungen preiszugeben.
  - Damit ist eine Signatur eine Art **Typ einer Struktur**.
  - Beispiele für Signaturen und Strukturen sind die grundlegenden Rechenstrukturen Keller, Liste und Menge.