Inhalt Kapitel 9: Ausnahmen und Seiteneffekte

Ausnahmen

Mit

`exception` *ausnahme*;

wird eine Ausnahme namens *ausnahme* (engl.: exception) deklariert.
Der Ausdruck

`raise` *ausnahme*

bewirkt dann, dass die Berechnung an dieser Stelle abgebrochen wird.
Dieser Ausdruck hat den Typ `'a`, kann also an beliebiger Stelle auftreten.

Beispiel

```
- exception Fehler;;
exception Fehler
-fun fakt n = if n < 0 then raise Fehler else
                  if n = 0 then 1 else n * fakt(n-1);
val fakt = fn : int -> int
- fakt 3;
val it = 6 : int
- fakt 0;
val it = 1 : int
- fakt ~1;
uncaught exception Fehler
  raised at: stdIn:5.34-5.40
-
```

Auffangen von Ausnahmen

Man kann eine Ausnahme durch das `try with` Konstrukt auffangen:

```
- fun f x = Int.toString(fakt x)
      handle Fehler => "Falsches Argument!";
val f = fn : int -> string
- f 9;
val it = "362880" : string
- f ~1;
val it = "Falsches Argument!" : string
-
```

Man kann mehrere verschiedene Ausnahmen in einem `handle` abfangen (durch | getrennt.
Typisierung: Die Typen aller `handle`-Klauseln müssen mit dem Typ des voranstehenden Audrucks übereinstimmen. Dieser gemeinsame Typ ist dann der Typ des gesamten Ausdrucks.

Simulation von Ausnahmen

Man kann Ausnahmen wie folgt simulieren
Ausnahme `Fehler` entspricht einem bestimmten Wert, z.B. `~1`.

```
fun fakt n = if n<0 then ~1 else
       if n=0 then 1 else n * fakt (n-1);

fun f x = let val res = fakt x in
      if res = ~1 then "Falsches Argument!"
      else Int.toString res end;
```

Nachteile: Umständlicher, nicht von vornherein klar, welche Werte den Ausnahmen entsprechen.

Ausnahmen mit Werten

Man kann einer Ausnahme einen Wert beigeben:

```
exception Fehler of int ;
```

Hier steht ein Typ

```
fun fakt x = if x < 0 then raise (Fehler x) else
       if x = 0 then 1 else x * fakt (x-1);;
fun f x =  Int.toString (fakt x)
          handle Fehler z => "fakt: falsches Argument: " ^
                             Int.toString z;
```

Test:

```
# f (-27);;
- : string = "fakt: falsches Argument: -27"
```

Der Typ exn

Ausnahmen sind selbst Werte des eingebauten Typs `exn`

```
- exception Fehler of string;
exception Fehler of string
- val benzin = Fehler "Benzin";
val benzin = Fehler(-) : exn
- val oel    = Fehler "Oelstand";
val oel = Fehler(-) : exn
- - fun f x = if x = 0 then raise benzin else 9;
val f = fn : int -> int
```

Kann sinnvoll sein, um Ausnahmen zusammenzubauen.

Typisierung von Ausdrücken mit Ausnahmen

- Hat e den Typ `exn`, so hat `raise` e den Typ `'a`, also beliebigen Typ.
- Haben e_0 und e_1 denselben Typ A, so hat auch e_0 `handle` $exn \Rightarrow e_1$ diesen Typ (vgl. Typisierung von if-then-else).
- In leichter Verallgemeinerung gilt dies auch für Ausnahmen mit Werten und mehrere Ausnahmeklauseln in einem `handle`-Block.

Auswertung von Ausdrücken mit Ausnahmen

- Die Auswertung eines jeden Ausdrucks liefert entweder einen regulären Wert, oder eine Ausnahme.
- Bei der Auswertung zusammengesetzter Ausdrücke werden wie üblich zunächst die Teilausdrücke ausgewertet. Tritt hier bereits eine Ausnahme auf, so wird die reguläre Auswertung abgebrochen und das Ergebnis ist die Ausnahme.
- Ein Ausdruck `raise` e wird ausgewertet, indem zunächst e ausgewertet wird. Das Ergebnis muss ein Wert des Typs `exn` sein. Der raise-Ausdruck liefert dann diesen Wert als Ausnahme, hat also keinen regulären Wert.

- Ein Ausdruck e_0 `handle` exn `=>` e_1 wird ausgeweertet, indem zunächst e_0 ausgewertet wird. Liefert das einen regulären Wert, so ist dieser auch das reguläre Ergebnis des Gesamtausdrucks.
  Liefert das die Ausnahme exn, so wird e_1 ausgewertet und das Ergebnis (ob regulär oder nicht) ist das Ergebnis des Gesamtausdrucks.
  Liefert e_0 eine andere Ausnahme als exn, so ist diese Ergebnis des Gesamtausdrucks.

Es gilt nunmehr: liefert ein Ausdruck e des Typs A einen regulären Wert v, so gehört v zum Typ A.
Über Ausnahmen macht der Typ A keine Aussage. Insofern ist der Typ `'a` für `raise` e gerechtfertigt.

Seiteneffekte

Ein *Seiteneffekt* liegt vor, wenn die Auswertung eines Ausdrucks neben dem eigentlichen Ergebnis noch weitere Auswirkungen hat. Beispiele von Seiteneffekten:

- Ausnahmen
- Nicht-termination (wird nicht immer den Seiteneffekten zugerechnet)
- Wertzuweisung

In Gegenwart von Seiteneffekten spielt die Reihenfolge und Anzahl der Auswertungen von Ausdrücken eine Rolle und auf die Auswertung von Ausdrücken, deren Wert nicht benutzt wird, kann im allgemeinen nicht verzichtet werden.

Referenzen

- Ist A ein Typ, so ist A `ref` auch ein Typ; seine Werte sind Speicherzellen (Referenzen), die Werte des Typs A beinhalten.
- Ist e vom Typ A `ref`, so ist !e vom Typ A und bezeichnet den Inhalt der Speicherzelle e
- Ist e_0 vom Typ A `ref` und e_1 vom Typ A, so ist $e_0 := e_1$ vom Typ `unit`; die Auswertung dieses Ausdrucks hat als Seiteneffekt die Zuweisung des Wertes von e_1 an e_0 (oder besser an dessen Wert).
- Ist e vom Typ A, so ist `ref` e vom Typ A `ref`. Ergebnis ist eine neue Speicherzelle, die mit dem Wert von e initialisiert ist. Vgl. “new”.

Anwendungsbeispiel: Symbolgenerator

```
- fun mkgensym () = let val n = ref 0 in
                      fn () => (n := !n+1;
                                "x_" ^ Int.toString (!n))
                    end
val mkgensym = fn : unit -> unit -> string
- val mygensym = mkgensym () ;
val mygensym = fn : unit -> string
- mygensym ();
val it = "x_1" : string
- mygensym ();
val it = "x_2" : string
- mygensym ();
val it = "x_3" : string
```

- Mit `mkgensym ()` erzeugt man einen Symbolgenerator.
- Dieser liefert bei jedem Aufruf ein neues Symbol.
- Ist z.B. bei der automatischen Codeerzeugung nützlich.

Pseudozufallsgenerator

```
- fun mkrandom seed range = let val n = ref seed in
   fn  () => let val x = (1291 * !n + 9991) mod 32768 in
    n:=x;(x mod range) + 1 end end;
val mkrandom = fn : int -> int -> unit -> int
- val  wuerfel = mkrandom 110965 6;
val wuerfel = fn : unit -> int
- wuerfel ()
val it = 1;
- wuerfel ()
val it = 2;
- wuerfel ()
val it = 5;
```

- `mksetup seed range` liefert einen Pseudozufallsgenerator, welcher bei jedem Aufruf eine "Zufalls"-zahl im Bereich $\{1, \ldots, \texttt{range}\}$ berechnet.

Anwendung: Memoisierung

```
- fun memoize f = let val x = ref 0  in
            let val fx = ref (f 0) in
              fn y => if y = !x then !fx else
                let val z = f y in x:=y;fx:=z;z  end end end
val memoize = fn : (int -> 'a) -> int -> 'a
- fun g n = if n = 0 then 0 else n * g(n-1);
val g = fn : int -> int
- val gm = memoize g;
val gm = fn : int -> int
```

- `memoize` berechnet aus einer Funktion `f:int->'a` eine "memoisierte Version" `fm`,
- Die memoisierte Version berechnet dasselbe Ergebnis wie `f`, merkt sich aber jeweils den letzten Aufruf.
- Die Seiteneffekte von `fm` sind nach außen hin nicht feststellbar,
- Hat `f` selbst Seiteneffekte, so darf man nicht memoisieren

Weitere Anwendungen von Referenzen

- Protokollierung von Aufrufen ("logging")
- Imperative Algorithmen, z.B. Hashtabellen und Fibonacci Heaps. (Es gibt aber meist auch funktionale Alternativen)
- Dynamische Programmierung (Verallgemeinerung von Memoisierung)

Reine Funktionen

- Eine SML-Funktion heißt rein, wenn sie keine Seiteneffekte hat
- Bei reinen Funktionen spielt nur das Ergebnis eine Rolle, nicht wann und wie oft es berechnet wird.
- Programme mit reinen Funktionen erlauben Optimierungen durch Codeumstellung und Parallelisierung und enthalten weniger Fehlerquellen
- Manchmal sind Seiteneffekte sinnvoll (Anwendungen)
- Manchmal verhält sich eine Funktion, die im Inneren Seiteneffekte verwendet nach außen hin rein
- In der imperativen Programmierung sind reine Funktionen die Ausnahme; in der funktionalen Programmierung sind sie die Regel
- Reine funktionale Sprachen wie Haskell erlauben überhaupt keine Seiteneffekte.

Zusammenfassung

- Ausnahmen erlauben die kontrollierte Behandlungen von Fehlern und Ausnahmesituationen
- Die Auslösung von Ausnahmen bricht die aktuelle Berechnung ab
- Ausnahmen können abgefangen und behandelt werden
- Referenzen sind Speicherzellen, die Daten eines bestimmten Typs enthalten und beschrieben und gelesen werden können
- Referenzen sind manchmal sinnvoll, sollten aber vermieden werden, wenn eine rein funktionale Lösung ebenso möglich ist.
- Reine Funktion sind Funktionen ohne (nach außen hin sichtbare) Seiteneffekte. Sie erlauben Optimierungen und einfachere Korrektheitsbeweise