Kapitel 13

Imperative Objekte

Wir führen jetzt Sprachkonstrukte ein, mithilfe derer wir Objekte erzeugen können, die wir im Laufe einer Berechnung gezielt verändern können. Diese sogenannten *imperativen Objekte* ermöglichen eine Vielzahl von neuen Programmiertechniken.

13.1 Zustände und Referenzen

Der bisher betrachteten Teilsprache von Standard ML liegt ein *Berechnungsmodell* zugrunde, dass sich grafisch wie folgt darstellen lässt:

$$e_1 \;\rightarrow\; e_2 \;\rightarrow\; \cdots \;\rightarrow\; e_n$$

Eine Berechnung wird mit einem Ausdruck e_1 gestartet. Auf den Startausdruck werden solange Auswertungsschritte angewendet, bis ein kanonischer Ausdruck e_n erreicht wird. Die Auswertungsschritte überführen Ausdrücke deterministisch in Folgeausdrücke. Der am Ende erreichte kanonische Ausdruck stellt den Wert des Startausdrucks dar.

Wir haben dieses Berechnungsmodell unter dem Namen Einschrittsematik formalisiert. Unter dem Namen Ergebnissemantik haben wir eine alternative Formalisierung dieses Modells kennengelernt. Die Ergebnissematik stellt Werte nicht als kanonische Ausdrücke dar, sondern direkter als mathematische Objekte. Außerdem realisiert sie das Konzept der Bezeichnerbindung nicht durch Substitution, sondern expliziter durch Umgebungen. Die beiden Formalisierungen realisieren sich ergänzende Sichtweisen auf das Konzept Berechnung. Im Folgenden werden wir von beiden Sichtweisen Gebrauch machen. Damit die Gemeinsamkeiten besser hervortreten, werden wir die kanonischen Ausdrücke auch als Werte be-

zeichnen. Das ist zwar formal gesehen ungenau, erhält uns aber die intuitiven Sprechweisen, die wir in Kapitel 1 eingeführt haben.

Wir erweitern unser Berechnungsmodell jetzt um sogenannte *Speicherzustände*. Eine Berechnung hat dann die Form

$$(e_1, S_1) \;\rightarrow\; (e_2, S_2) \;\rightarrow\; \cdots \;\rightarrow\; (e_n, S_n)$$

Jeder *Berechnungszustand* (e_i, S_i) besteht aus einem Ausdruck e_i und einem Speicherzustand S_i. Eine Speicherzustand ist eine endliche Funktion, die sogenannte *Referenzen* an Werte bindet. Die bisherigen Sprachkonstrukte machen von den Speicherzuständen keinen Gebrauch und werden wie bisher augewertet. Wir führen aber drei neue Operationen ein, die Zustände lesen und verändern können:

1. *Allokation.* Diese Operation bekommt einen Wert v und ersetzt den aktuellen Zustand S durch den Zustand $S[v/r]$, wobei r die kleinste Referenz ist, die nicht in *Dom* S ist. Sie liefert die „neue“ Referenz r als Ergebnis.

2. *Dereferenzierung.* Diese Operation bekommt eine Referenz r und liefert den Wert $S(r)$, wobei S der aktuelle Zustand ist.

3. *Zuweisung.* Diese Operation bekommt eine Referenz r und einen Wert v und ersetzt den aktuellen Zustand S durch den Zustand $S[v/r]$. Sie liefert das leere Tupel () als Ergebnis.

Standard ML stellt die neuen Operationen wie folgt zur Verfügung:

`eqtype 'a ref`	Referenztypen
`ref : 'a -> 'a ref`	Allokation
`!  : 'a ref -> 'a`	Dereferenzierung
`:=  : 'a ref * 'a -> unit`	Zuweisung

Der Typkonstruktor `ref` liefert unendlich viele *Referenztypen*. Da die Referenztypen abstrakt sind, ist gewährleistet, dass erreichbare Berechnungszustände (e, S) nur Referenzen enthalten, die durch den Speicherzustand S gebunden sind.

Sie wissen jetzt genug, um die folgenden Interaktionen mit einem Standard ML

Interpreter zu verstehen:

`val r = ref 0` *`val r : int ref`*	$r \mapsto \overline{7}$	$\overline{7} \mapsto 0$
`!r` *`0 : int`*		
`r:=25` *`() : unit`*		$\overline{7} \mapsto 25$
`val r' = ref 25` *`val r' : int ref`*	$r' \mapsto \overline{8}$	$\overline{8} \mapsto 25$
`r=r'` *`false : bool`*		

Die rechtsstehenden Annotationen zeigen Bindungen, die in die Wertumgebung und den Speicherzustand des Interpreters eingetragen werden. Referenzen stellen wir durch überstrichene natürliche Zahlen dar (zum Beispiel $\overline{7}$). Wir nehmen an, dass $\overline{7}$ und $\overline{8}$ die zwei kleinsten Referenzen sind, die im initialen Speicherzustand ungebunden sind.

Als Nächstes betrachten wir das Auswertungsprotokoll für den Berechnungszustand

(`((fn r => (!r, r:=7, !r)) (ref 2)`, $\emptyset$)

Dabei stellt $\emptyset$ den leeren Speicherzustand dar.

`(fn r => (!r, r:=7, !r)) (ref 2)`	$\emptyset$
$\rightarrow$ `(fn r => (!r, r:=7, !r))` $\overline{0}$	$\{\overline{0} \mapsto 2\}$
$\rightarrow$ `(!`$\overline{0}$`, `$\overline{0}$`:=7, !`$\overline{0}$`)`	$\{\overline{0} \mapsto 2\}$
$\rightarrow$ `(2, `$\overline{0}$`:=7, !`$\overline{0}$`)`	$\{\overline{0} \mapsto 2\}$
$\rightarrow$ `(2, (), !`$\overline{0}$`)`	$\{\overline{0} \mapsto 7\}$
$\rightarrow$ `(2, (), 7)`	$\{\overline{0} \mapsto 7\}$

Wir sagen, dass eine Referenz r auf einen Wert v *zeigt*, wenn der aktuelle Speicherzustand r an v bindet. Wenn wir einer Referenz r einen Wert v zuweisen, sagen wir auch, dass wir r auf v *setzen*.

Die Operationssymbole `!` und `ref` werden von der konkreten Syntax wie das Negationssymbol `~` als Konstanten behandelt (siehe Abschnitt 12.5). Daher kann man bei

```
! (! (ref (ref 1)))
1 : int
```

keines der Klammerpaare weglassen. Andererseits ist die folgende Eingabe zulässig:

```
map ! (map ref [1, 2, 3])
[1, 2, 3] : int list
```

Im Zusammenhang mit Zuweisungen ist manchmal die abgeleitete Form

e_1 `before` e_2 $\quad\Longrightarrow\quad$ `#1(`e_1`,` e_2`)`

bequem. Hier ist ein Beispiel:

```
!r before r:=x
```

Dieser Ausdruck weist der Referenz `r` den Wert `x` zu und liefert den bisherigen Wert der Referenz als Ergebnis.

13.2 Zähler

Mit Referenzen und Speicherzuständen sind wir in der Lage, eine Prozedur

```
counter : unit -> int
```

zu schreiben, die beim n-ten Aufruf die Zahl n liefert. Die Prozedur zählt also mit, wie oft sie aufgerufen wurde. Wir realisieren die Prozedur wie folgt:

```
val r = ref 0
fun counter () = (r:= !r+1 ; !r)
```

Nehmen Sie an, dass wir die zwei Deklarationen, mit dem Speicherzustand $\{\overline{0} \mapsto 66\}$ beginnend, auswerten. Dann bekommen wir den Speicherzustand $\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 0\}$ und der Bezeichner `counter` wird an die Prozedur

```
fn () => (1̄:= !1̄+1 ; !1̄)
```

gebunden. Das folgende Berechnungsprotokoll erklärt die Arbeitsweise der Pro-

zedur:

`(fn () => (`$\overline{1}$`:= !`$\overline{1}$`+1 ; !`$\overline{1}$`)) ()`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 0\}$
$\rightarrow$ `(`$\overline{1}$`:= !`$\overline{1}$`+1 ; !`$\overline{1}$`)`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 0\}$
$\rightarrow$ `(`$\overline{1}$`:= 0+1 ; !`$\overline{1}$`)`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 0\}$
$\rightarrow$ `(`$\overline{1}$`:= 1 ; !`$\overline{1}$`)`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 0\}$
$\rightarrow$ `(() ; !`$\overline{1}$`)`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 1\}$
$\rightarrow$ `!`$\overline{1}$	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 1\}$
$\rightarrow$ `1`	$\{\overline{0} \mapsto 66,\ \overline{1} \mapsto 1\}$

Ein Aufruf der Prozedur `counter` liefert also nicht nur ein Ergebnis, sondern hat zusätzlich den *Effekt*, dass er den Wert der Referenz $\overline{1}$ im Speicherzustand um eins erhöht.

Die obige Realisierung der Prozedur `counter` ist unschön, da es über den Bezeichner `r` möglich ist, den Wert der Referenz der Prozedur von außen zu ändern. Diese Sicherheitslücke lässt sich schließen, wenn man `counter` wie folgt deklariert:

```
val counter =
    let
        val r = ref 0
    in
        fn () => (r := !r+1 ; !r)
    end
```

Wir sagen, dass die Prozedur `counter` bei dieser Realisierung ihren Zustand *enkapsuliert* (d.h. einkapselt).

Hier ist eine Prozedur

```
newCounter : int -> (unit -> int)
```

die bei jedem Aufruf einen neuen Zähler mit einem gegebenen Anfangswert erzeugt:

```
fun newCounter i =
    let
        val r = ref(i-1)
    in
        fn () => (r := !r+1 ; !r)
    end
```

Damit haben wir:

```
val c = newCounter 0
val c : unit -> int

c()
0 : int

c() + c()
3 : int

val c' = newCounter ~3
val c' : unit -> int

c'()
~3 : int

c'() + c()
1 : int
```

Die Prozedur `newCounter` ist also ein Generator, der bei jedem Aufruf einen neuen Zähler liefert.

13.3 Funktionale und imperative Objekte

Physikalische Objekte haben die Eigenschaft, dass sie ihren Zustand mit der Zeit verändern. Ein Beispiel ist eine Uhr mit einem Sekundenzeiger. Offensichtlich wechselt die Uhr mit jeder Sekunde in einen neuen Zustand. Außerdem kann man den Zustand der Uhr dadurch ändern, dass man eine andere Zeit einstellt.

Physikalische Objekte unterscheiden sich also gravierend von mathematischen Objekten, die keinen zeitlichen Veränderungen unterworfen sind (denken Sie an die Zahl 12). Andererseits ist es so, dass wir uns den Zustand, in dem sich ein physikalisches Objekt zu einem bestimmten Zeitpunkt befindet, als ein mathematisches Objekt vorstellen können.

Im programmiersprachlichen Kontext bezeichnet man unveränderliche Objekte als *funktionale Objekte* und veränderliche Objekte als *imperative Objekte*. Mithilfe von Referenzen sind wir in der Lage, imperative Objekte zu realisieren. Das liegt daran, dass der Wert, an den eine Referenz gebunden ist, durch Zuweisung geändert werden kann. Für imperative Objekte verwendet man Sprechweisen, die sich an die Sprechweisen für physikalische Objekte anlehnen. Im Gegensatz zu funktionalen Objekten, die unabhängig von der Zeit existieren, existieren imperative Objekte nur für eine bestimmte Zeitdauer. Wir sagen, dass imperative Objekte *erzeugt* und *eliminiert* werden. Die Zeitdauer zwischen Erzeugung und Elimination ist die *Lebensdauer* eines imperativen Objektes.

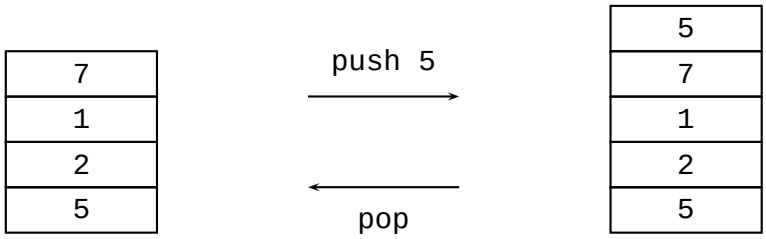


Abbildung 13.1: Grafische Darstellung von Stapeln

Wir können uns eine Referenz als den Namen eines primitiven imperativen Objekts vorstellen. Die durch Referenzen identifizierten imperativen Objekte bezeichnet man oft als *Zellen*. Unter einer Zelle kann man sich einen Behälter vorstellen, der zu jedem Zeitpunkt genau einen Wert enthält. Mit der Dereferenzierungsoperation kann man den Wert in einer Zelle lesen. Mit der Zuweisungsoperation kann man den Wert in einer Zelle durch einen anderen Wert ersetzen. Die Allokationsoperation erzeugt eine neue Zelle. Ein Programmiersystem kann eine Zelle eliminieren, wenn sichergestellt ist, dass die Zelle nicht mehr zugegriffen werden kann. Beispielsweise kann die durch die Ausführung des Ausdrucks

```
(ref 0 ; 1)
```

erzeugte Zelle sofort wieder eliminiert werden.

Mithilfe von Referenzen kann komplexere imperative Objekte realisieren. Ein erstes Beispiel sind die im letzten Abschnitt besprochenen Zählerprozeduren.

13.4 Stapel

Unter einem Stapel (englisch „stack“) versteht man in der Programmierung ein imperatives Objekt, in das mehrere Werte abgelegt werden können. Für Stapel gibt es drei Operationen:

- *push* legt einen Wert auf einen Stapel.
- *top* liefert den obersten Wert eines Stapels.
- *pop* nimmt den obersten Wert von einem Stapel.

Abbildung 13.1 zeigt eine grafische Darstellung dieser Ideen. Man sagt, dass Stapel nach der *LIFO-Strategie* („last in, first out“) verwaltet werden.

```
signature STACK =
  sig
    eqtype 'a stack

    val stack : unit -> 'a stack
    val push  : 'a stack * 'a -> unit
    val top   : 'a stack -> 'a           (* Empty *)
    val pop   : 'a stack -> unit         (* Empty *)
    val empty : 'a stack -> bool
  end

structure Stack :> STACK =
  struct
    type 'a stack = 'a list ref

    fun stack () = ref nil

    fun push (s,x) = s:= x:: !s

    fun top s = hd(!s)

    fun pop s = s:= tl(!s)

    fun empty s = null(!s)
  end
```

Abbildung 13.2: Spezifikation von Stapeln

Abbildung 13.2 spezifiziert die Typabstraktion Stapel mit einer Signatur und einer Modellimplementierung. Die Modellimplementierung realisiert einen Stapel mit einer Referenz, die auf die Liste der jeweils auf dem Stapel liegenden Werte zeigt. In der Liste stehen neuere Einträge vor älteren Einträgen. Die Operation `stack` liefert bei jedem Aufruf einen neuen Stapel, auf dem noch keine Werte liegen. Die Operation `empty` testet, ob sich auf einem Stapel Werte befinden. Wenn die Operationen `top` und `pop` auf einen leeren Stapel angewendet werden, liefern sie die Ausnahme `Empty`.

Hier ist ein Experiment:

```
open Stack

val s : int stack = stack()
val s : int stack

(push(s,1) ; push(s,2))
() : unit

(top s, pop s, top s, empty s)
(2, (), 1, false) : int * unit * int * bool
```

13.5 Reihungen

Eine Reihung (englisch „array“) ist ein imperatives Tupel, dessen Komponenten alle den gleichen Typ haben. Die Komponenten einer Reihung sind mit null beginnend durchnummeriert, man spricht vom *Index* einer Komponente. Die *Länge* einer Reihung (die Anzahl der Komponenten) wird bei der Erzeugung festgelegt und kann nicht geändert werden. Mit der Operation *sub* kann der Wert einer Komponente gelesen werden, mit der Operation *update* kann er ersetzt werden. Dabei wird die Komponente durch ihren Index identifiziert.

Für die Zustände von Reihungen sind zwei grafische Darstellungen gebräuchlich. Die horizontale Darstellung

`"Maria"`	`"Tom"`	`"Bob"`	`"Monica"`
0	1	2	3

ist an die Darstellung von Tupeln angelehnt. Die vertikale Darstellung ist um neunzig Grad gedreht:

0	`"Maria"`
1	`"Tom"`
2	`"Bob"`
3	`"Monica"`

Abbildung 13.3 spezifiziert Reihungen mit einer Signatur und einer Modellimplementierung. Die Operation array liefert zu (n, x) eine Reihung mit n Komponenten, die alle den Wert x haben. Die Operation length liefert die Länge einer Reihung. Die Modellimplementierung realisiert eine Reihung durch eine Liste von Referenzen. Eine Realisierung durch ein Tupel von Referenzen wäre angemessener, benötigt aber einen Typkonstruktor für homogene Tupel beliebiger Stelligkeit, den wir nicht eingeführt haben.[1]

Es gibt eine Standardstruktur `Array`, die Reihungen gemäß unserer Spezifikation, aber erweitert um zusätzliche Operationen, zur Verfügung stellt. Im Gegensatz zu unserer Modellimplementierung realisiert die Standardstruktur `Array` die Operationen sub und update aber mit konstanter Laufzeit (also unabhängig vom Index einer Komponente). Diese schnelle Realisierung ist möglich, da der Hauptspeicher eines Computers eine durch Hardware realisierte Reihung ist, deren Komponenten in konstanter Zeit gelesen und geschrieben werden können.

In der bisher vorgestellten Teilsprache von Standard ML ist eine Realisierung

[1] Die Standardstruktur `Vector` stellt einen solchen Typ zur Verfügung.

```
signature ARRAY =
  sig
    type 'a array

    val array  : int * 'a -> 'a array         (* Size *)
    val sub    : 'a array * int -> 'a         (* Subscript *)
    val update : 'a array * int * 'a -> unit  (* Subscript *)
    val length : 'a array -> int
  end

structure Array :> ARRAY =
  struct
    type 'a array = 'a ref list

    fun array (n,x) = List.tabulate(n, fn _ => ref x)

    fun sub (a,n) = !(List.nth(a,n))

    fun update (a,n,x) = List.nth(a,n) := x

    val length = List.length
  end
```

Abbildung 13.3: Spezifikation von Reihungen

von Reihungen mit konstanter Laufzeit für sub und update unmöglich. Mit Rot-Schwarz-Bäumen ist eine Realisierung mit logarithmischer Laufzeit möglich (relativ zur Länge der Reihung).

Reihungen sind ein wichtiger Ausgangspunkt für die Realisierung effizienter imperativer Datenstrukturen. Man findet sie daher in fast jeder Programmiersprache.

Reversieren von Reihungen

Um zu sehen, wie man mit Reihungen programmiert, entwickeln wir eine Prozedur `reverse`, die Reihungen reversiert. Reversion bedeutet, dass der Zustand der Reihung so geändert wird, dass die Werte den Komponenten in reversierter Reihenfolge zugeordnet sind. Wir werden `reverse` ohne Listen und ohne Hilfsreihungen realisieren.

Wir beginnen mit einer Prozedur

```
fun swap a i j = let val x = Array.sub(a,i)
                 in  Array.update(a, i, Array.sub(a, j)) ;
                     Array.update(a, j, x)
                 end
swap : 'a array -> int -> int -> unit
```

die die Werte zweier Komponenten i und j einer Reihung a vertauscht. Die Prozedur

```
fun reverse' a l u = if l<u then
                      (swap a l u ; reverse' a (l+1) (u-1))
                    else ()
val reverse' : 'a array -> int -> int -> unit
```

reversiert die Werte der Komponenten l bis u einer Reihung a. Damit können wir `reverse` wie folgt schreiben:

```
fun reverse a = reverse' a 0 (Array.length a - 1)
val reverse : 'a array -> unit
```

Man sieht sofort, dass `reverse` eine Reihung der Länge n mit der Laufzeit $\Theta(n)$ reversiert.

Intervall-Test

Wir wollen eine Prozedur

```
test : int list -> int -> int -> bool
```

schreiben, die testet, ob eine Liste alle Zahlen enthält, die zwischen zwei gegebenen Zahlen l und u liegen. Wir werden die Prozedur mit einer Reihung realisieren, die für jede Zahl zwischen l und u eine Komponente hat. Zu Beginn tragen alle Komponenten der Reihung den Wert `false`. Wir testen dann für jedes Element der Liste, ob es zwischen l und u liegt. Wenn dies für ein Element der Fall ist, setzen wir die entsprechende Komponente der Reihung auf `true`. Nachdem wir alle Elemente der Liste getestet haben, liefern wir `true` genau dann, wenn alle Komponenten der Reihung den Wert `true` haben. Damit bekommen wir die folgende Prozedur:

```
fun test xs l u =
  let
    val a = Array.array(Int.abs(l-u)+1, false)

    fun test' x = if l<=x andalso x<=u
                  then Array.update(a, x-l, true)
                  else ()
  in
    app test' xs ;
    Array.foldl (fn (b,b') => b andalso b') true a
  end
val test : int list -> int -> int -> bool
```

Die Faltungsprozedur `Array.foldl` arbeitet analog zur Faltungsprozedur für Listen.

Man kann die Prozedur `test` auch ohne imperative Objekte realisieren. Beispielsweise kann man die Liste zuerst sortieren und danach die Intervall-Eigenschaft testen. Die Realisierung mit einer Reihung ist aber verblüffend einfach und hat zudem lineare Laufzeit bezüglich der Länge der Liste. Dazu kommt allerdings die Laufzeit für die Erzeugung der Reihung (linear bezüglich der Breite des Intervalls).

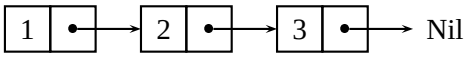
Die Prozedur `test` erzeugt bei jedem Aufruf eine neue Reihung. Nachdem ein Aufruf beendet ist, wird die erzeugte Reihung automatisch eliminiert, da sie dann nicht mehr erreichbar ist.

13.6 Imperative Listen

Die Werte des Typs

```
datatype 'a ilist = Nil | Cons of 'a * 'a ilist ref
```

stellen imperative Objekte dar, die man als *imperative Listen* bezeichnet. Imperative Listen unterscheiden sich von funktionalen Listen dadurch, dass der Rumpf einer nichtleeren Liste durch eine Referenz realisiert wird, die auf die Restliste zeigt. Die imperative Version der Liste $[1, 2, 3]$ können wir grafisch wie folgt darstellen:



Die Kästen stellen Werte dar. Die Pfeile stellen Bindungen von Referenzen im Speicherzustand dar und werden als *Zeiger* bezeichnet. Diese Bindungen können durch Zuweisung geändert werden.

Die Prozedur

```
fun toIlist   nil   = Nil
  | toIlist (x::xr) = Cons(x, ref(toIlist xr))
```

```
val toIlist : 'a list -> 'a ilist
```

liefert die imperative Version einer Liste. Beispielsweise liefert

```
toIlist [1, 2, 3]
```

```
Cons(1, ref(Cons(2, ref(Cons(3, ref Nil))))) : int ilist
```

die oben dargestellte imperative Liste. Die Ausgabe des Moscow ML Interpreters stellt Referenzen, wie oben gezeigt, durch das Operationssymbol `ref` und die aktuellen Bindungen im Speicherzustand dar.

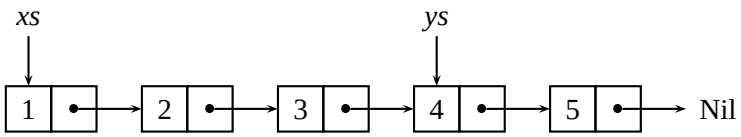
Die Prozedur

```
fun append (Nil,       ys) = raise Empty
  | append (Cons(x,r), ys) = case !r of
                               Nil => r:=ys
                             | xs  => append(xs,ys)
val append : 'a ilist * 'a ilist -> unit
```

verlängert eine nichtleere imperative Liste dadurch, dass sie den auf `Nil` zeigenden Zeiger des letzten Kastens auf eine gegebene imperative Liste *ys* setzt. Beispielsweise liefert

```
val xs = toIlist [1, 2, 3]
val xs = Cons(1, ref(Cons(2, ref(Cons(3, ref Nil))))) : int ilist
val ys = toIlist [4,5]
val ys = Cons(4, ref(Cons(5, ref Nil))) : int ilist
append(xs,ys)
() : unit
```

die folgende Situation:

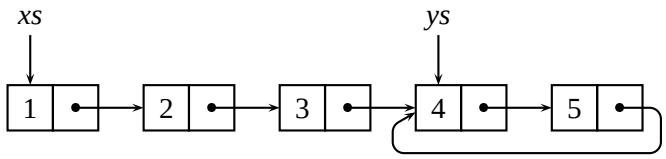


Die von den Bezeichnern *xs* und *ys* ausgehenden Pfeile stellen die Bindungen der Bezeichner dar.

Mit der Prozedur `append` können wir zyklische imperative Listen konstruieren. Ausgehend von der obigen Situation liefert

```
append(ys,ys)
```

die folgende Situation:



Solche durch Referenzen laufenden Zyklen kann es bei funktionalen Listen nicht geben.

Hase-Igel-Algorithmus

Wir wollen jetzt eine Prozedur

```
cyclic : 'a ilist -> bool
```

schreiben, die testet, ob eine imperative Liste zyklisch ist. Dazu verwenden wir den *Hase-Igel-Algorithmus*. Dieser arbeitet mit zwei Zeigern h und i, die auf Referenzen der zu testenden imperativen Liste zeigen. Zu Beginn wird i auf die erste und h auf die zweite Referenz der Liste gesetzt. Danach geht der Algorithmus wie folgt vor:

1. Wenn $i = h$, dann ist die Liste zyklisch.
2. Wenn $i \neq h$, dann prüfe, ob h um zwei Positionen weiter gesetzt werden kann:
   a) Wenn dies nicht möglich ist, dann ist die Liste nicht zyklisch.
   b) Wenn dies möglich ist, setze i um eine und h um zwei Positionen weiter.

Wenn der Algorithmus terminiert, liefert er offensichtlich das richtige Ergebnis. Für azyklische Listen ist die Terminierung offensichtlich. Wenn die zu testende Liste einen Zyklus enthält, dann befinden sich die Zeiger nach endlich vielen Schritten beide in diesem Zyklus. Wir betrachten jetzt den Abstand von h zu i. Da h bei jedem Schritt um zwei Positionen und i nur um eine nach vorne rückt, rückt h bei jedem Schritt um eine Position näher an i heran. Also holt h den Zeiger i schließlich auf und beide zeigen auf die dieselbe Referenz. Damit terminiert der Algorithmus. Da i jede Referenz der Liste höchstens einmal besucht, hat der Algorithmus lineare Laufzeit bezüglich der Anzahl der Referenzen der Liste.

Den Hase-Igel-Algorithmus realisieren wir wie folgt:

```
fun cyclic'(ref(Cons(_,i)), ref(Cons(_,ref(Cons(_,h)))))
              = i=h orelse cyclic'(i,h)
  | cyclic' _ = false
val cyclic' : 'a ilist ref * 'a ilist ref -> bool

fun cyclic Nil          = false
  | cyclic (Cons(_, r)) = cyclic'(r,r)
val cyclic : 'a ilist -> bool
```

Die Deklaration von `cyclic'` benutzt das Muster

```
ref(Cons(_,i))
```

Dieses passt auf alle Referenzen, die im Speicherzustand an einen mit `Cons` konstruierten Wert gebunden sind.

13.7 Schleifen

Schleifen sind ein Sprachkonstrukt, das zusammen mit Referenzen die Formulierung von rekursiven Berechnungen ermöglicht. Als Beispiel betrachten wir eine Prozedur `sum`, die zu $n \in \mathbb{N}$ die Summe $1 + 2 + \cdots + n$ liefert:

```
fun sum n =
  let
    val i = ref 1
    val a = ref 0
  in
    while !i <= n do
      (a:= !a + !i ;
       i:= !i + 1 ) ;
    !a
  end
val sum : int -> int
```

Eine Schleife ist ein Ausdruck der Form

```
while e1 do e2
```

der aus zwei Unterausdrücken e_1 und e_2 besteht. Der Ausdruck e_1 muss den Typ `bool` haben und wird als *Bedingung* der Schleife bezeichnet. Der Ausdruck e_2 kann einen beliebigen Typ haben und heißt *Rumpf* der Schleife. Eine Schleife hat immer den Typ `unit`.

Die Semantik einer Schleife lässt sich am besten mit der rekursiven Gleichung

```
while e1 do e2  =  if e1 then while e1 do e2 else ()
```

erklären. Eine Schleife wertet also ihre Bedingung und ihren Rumpf wiederholt so lange aus, bis die Bedingung nicht mehr zu `true` evaluiert. Eine terminierende Schleife liefert immer den trivialen Wert `()` und ist folglich nur dann interessant, wenn der Rumpf oder die Bedingung einen nach außen sichtbaren Effekt haben.

In Standard ML sind Schleifen eine abgeleitete Form, da sie sich auf iterative Rekursion mit Prozeduren zurückführen lassen:

```
while e1 do e2

  ⟹

let
  fun loop () = if e1 then (e2 ; loop()) else ()
in
  loop()
end
```

Andererseits ist es immer möglich, iterative Rekursion mit Prozeduren durch Schleifen und Referenzen zu ersetzen. Wir zeigen das am Beispiel einer Prozedur, die die Länge von Listen berechnet:

```
fun length' (xr,n) = if not(null xr)
                     then length'(tl xr, n+1)
                     else n

fun length xs = length'(xs,0)
```

```
val length : 'a list -> int
```

Um die iterative Hilfsprozedur `length'` durch eine Schleife zu ersetzen, führen wir für jedes Argument von `length'` eine Referenz ein:

```
fun length xs =
  let
    val xr = ref xs
    val n  = ref 0
  in
    while not(null(!xr)) do
      (xr:= tl(!xr) ;
       n:= !n+1 ) ;
    !n
  end
```

```
val length : 'a list -> int
```

In Standard ML spielen Schleifen keine große Rolle, da die Verwendung einer iterativ-rekursiven Hilfsprozedur notational bequemer und in Bezug auf Effizienz gleichwertig ist. Bei Sprachen wie Java und C ist die Situation anders, da die üblichen Übersetzer für diese Sprachen iterative Rekursion weniger effizient als Schleifen realisieren. Damit ist der Programmierer gezwungen, iterative Rekursion durch Schleifen zu ersetzen. Als Ausgleich haben diese Sprachen eine konkrete Syntax, mit der Programme mit Schleifen und Referenzen eleganter als in Standard ML geschrieben werden können.

Man unterscheidet zwischen dem funktionalen und dem imperativen Programmierstil. Beim *funktionalen Programmierstil* setzt man imperative Objekte nur dann ein, wenn dies klare Vorteile bringt. Folglich verzichtet man hier völlig auf Schleifen. Dagegen betrachtet der *imperativen Programmierstil* imperative Objekte und Schleifen als grundlegende Ausdrucksmittel. Standard ML ist für die Programmierung im funktionalen Stil, Java und C sind für die Programmierung im imperativen Stil entworfen. Je nachdem, für welchen Programmierstil eine Sprache entworfen ist, spricht man von einer *funktionalen* oder *imperativen Sprache*. Die moderne Ausprägung von imperativen Sprachen bezeichnet man als *objektorientiert*. Objektorientierte Sprachen (Java, C++) erweitern imperative Sprachen um einige Konzepte aus funktionalen Sprachen.

Der funktionale Programmierstil basiert auf mathematischen Konzepten. Der imperative Programmierstil orientiert sich an den durch die Hardware eines Computers realisierten Strukturen. Nachdem man Computer bauen konnte, wurden die ersten imperativen Programmiersprachen unter großem Zeitdruck entwickelt (Fortran, Cobol, Algol, etwa 1960). Funktionale Sprachen wurden erst ab etwa 1975 entwickelt. Ein einflußreicher Vorgänger der funktionalen Sprachen war Lisp (um 1960).

13.8 Referenzen und ambige Deklarationen

In Abschnitt 3.4 haben wir gelernt, dass Standard ML zwischen monomorphen, polymorphen, und ambigen Deklarationen unterscheidet. Jetzt können wir erklären, warum ambige Deklarationen nicht als polymorphe Deklarationen behandelt werden.

Betrachten Sie dazu den Ausdruck

```
let
    val r = ref (fn x => x)
in
    r:= (fn () => ()) ;
    !r 4
end
```

Dieser Ausdruck ist aus drei Gründen unzulässig:

1. Das erste benutzende Auftreten von `r` verlangt den Typ

   ```
   (unit -> unit) ref
   ```

2. Das zweite benutzende Auftreten von `r` verlangt einen Typ

   ```
   (int -> 'a) ref
   ```

3. Der Bezeichner `r` kann nicht polymorph typisiert werden, da die Deklaration von `r` ambig ist.

Um zu zeigen, dass die Unterscheidung zwischen polymorphen und ambigen Deklarationen sinnvoll ist, nehmen wir an, dass die Deklaration von `r` wie eine polymorphe Deklaration behandelt wird. Dann wird `r` mit dem Typschema

```
∀ 'a ('a -> 'a) ref
```

typisiert. Damit können den zwei benutzenden Auftreten von `r` jeweils passende Typen zugeordnet werden und der Ausdruck ist zulässig. Das sollte er aber auf keinen Fall sein, da bei der Ausführung des Ausdruck `!r 4` eine Prozedur vom Typ

```
unit -> unit
```

auf ein Argument vom Typ `int` angewendet wird.

13.9 Semantik von F mit Referenzen

Wir zeigen jetzt, wie man die formale Definition von F (siehe Kapitel 11) um Referenzen erweitert.

Zuerst erweitern wir die abstrakte Syntax von F um neue Varianten für Typen und Ausdrücke:

$$t \in \mathit{Ty} \;=\; \cdots \mid\; t\ \texttt{ref} \mid\; \texttt{unit}$$
$$e \in \mathit{Exp} \;=\; \cdots \mid\; \texttt{ref}\ e \mid\; !\,e \mid\; e_1 := e_2$$

Die Erweiterung der Definition der statischen Semantik ist einfach. Es werden lediglich drei zusätzliche Inferenzregeln für die neuen Ausdrucksvarianten benötigt. Die Regeln ergeben sich sofort aus den in Abschnitt 13.1 angegeben Typen für die neuen Operationen.

Damit sind wir bei der dynamischen Semantik. Wir zeigen, wie man die Ergebnisemantik aus Abschnitt 11.3 erweitert. Referenzen stellen wir durch natürliche Zahlen dar:

$$\mathit{Ref} \;=\; \mathbb{N}$$

Das führt zu keinen Schwierigkeiten, da die Typdisziplin eine Verwechslung von Referenzen mit den eigentlichen Zahlen, also den Werten des Typ `int`, verhindert. Zudem erspart uns das die Einführung neuer Werte.

Da wir F um den Typ `unit` erweitert haben, müssen wir das leere Tupel als Wert darstellen. Dafür wählen wir die Zahl 0.

Speicherzustände definieren wir erwartungsgemäß wie folgt:

$$S \in \mathit{Sta} \;=\; \mathbb{N} \xrightarrow{\mathit{fin}} \mathit{Val}$$

Für die Auswertung eines Ausdrucks benötigen wir jetzt eine Wertumgebung und einen Speicherzustand (den *Anfangszustand* der Auswertung). Wenn die Auswertung terminiert, liefert sie einen Wert und zusätzlich einen Speicherzustand (den *Endzustand* der Auswertung). Wir sagen, dass die Auswertung eines Ausdrucks

einen sogenannten *Effekt* hat, wenn sich der Endzustand vom Anfangszustand unterscheidet. Ein Effekt kann durch die Allokation einer neuen Referenz oder durch die Zuweisung eines neuen Werts an eine existierende Referenz erreicht werden.

Diese Überlegungen legen nahe, die dynamische Semantik als eine Menge

$$DS \subseteq Sta \times VE \times Exp \times Val \times Sta$$

zu definieren. Wir schreiben

$$S, V \vdash e \Rightarrow v, S'$$

für

$$(S,\ V,\ e,\ v,\ S') \in DS$$

Dabei soll $S, V \vdash e \Rightarrow v, S'$ genau dann gelten, wenn die Ausführung des Ausdrucks e in der Wertumgebung V, beginnend mit dem Anfangszustand S, terminiert und den Wert v sowie den Endzustand S' liefert.

Für die Definition der Menge DS benötigen wir auch für die bisherigen Ausdrucksvarianten von F neue Inferenzregeln. Die neuen Regeln erweitern die bisherigen Regeln in Abbildung 11.9 um das *Durchfädeln* des Zustands. Wir zeigen hier die erweiterten Regeln für `false`, Addition und Prozeduranwendung:

$$\frac{}{S, V \vdash \texttt{false} \Rightarrow 0, S}$$

$$\frac{S, V \vdash e_1 \Rightarrow v_1, S_1 \qquad S_1, V \vdash e_2 \Rightarrow v_2, S' \qquad v_1, v_2 \in \mathbb{Z} \qquad v = v_1 + v_2}{S, V \vdash e_1 + e_2 \Rightarrow v, S'}$$

$$\frac{S, V \vdash e_1 \Rightarrow (x, e, V'),\ S_1 \qquad S_1, V \vdash e_2 \Rightarrow v_2, S_2 \qquad S_2,\ V' + \{x \mapsto v_2\} \vdash e \Rightarrow v, S'}{S, V \vdash e_1\ e_2 \Rightarrow v, S'}$$

Beachten Sie, dass der Zustand jeweils gemäß der Ausführungsreihenfolge durchgefädelt wird. Damit legen die erweiterten Regeln die Ausführungsreihenfolge vollständig fest. Die Erweiterung des restlichen Regeln für F erfolgt analog.

Damit sind wir bei den Regeln für die neuen Ausdrucksvarianten:

$$\frac{S, V \vdash e \Rightarrow v, S' \qquad r = \min(\mathbb{N} - \mathit{Dom}\ S') \qquad S'' = S' + \{r \mapsto v\}}{S, V \vdash \texttt{ref}\ e \Rightarrow r, S''}$$

$$\frac{S, V \vdash e \Rightarrow r, S' \qquad v = S'(r)}{S, V \vdash\ !\, e \Rightarrow v, S'}$$

$$\frac{\begin{array}{ll} S, V \vdash e_1 \Rightarrow r, S_1 & S_1, V \vdash e_2 \Rightarrow v, S_2 \\ r \in \mathit{Dom}\ S_1 & S' = S_2 + \{r \mapsto v\} \end{array}}{S, V \vdash e_1 := e_2 \Rightarrow 0, S'}$$

Beachten Sie, dass nur diese Regeln auf die Zustände zugreifen.

Proposition 11.3.1 und der Typerhaltungssatz 11.3.3 gelten auch für das um Referenzen erweiterte F. Der Auswertbarkeitssatz 11.3.2 ist aber nicht mehr gültig, da man mit Referenzen rekursive Prozeduren simulieren kann. Wir zeigen das am Beispiel der Fakultätsprozedur in Standard ML:

```
fun fak n =
  let
    val r = ref (fn x => x)
    fun f x = if x<2 then 1 else x * !r(x-1)
  in
    r:=f ;
    f n
  end
val fak : int -> int
```

Übungen

Aufgabe 13.1 (Referenzen) Zuwelchem Wert evaluiert der Ausdruck

```
ref 1 = ref 1
```

Begründen Sie ihre Aussage.

Aufgabe 13.2 (Generatoren) Ein Generator für eine Folge $x_1, x_2, \ldots$ von Werten eines Typs t ist eine Prozedur $\texttt{unit} \to t$, die beim n-ten Aufruf x_n liefert.

1. Schreiben Sie einen Generator `nextSquare` für die Folge $0, 1, 4, 9, \ldots$ der Quadratzahlen.
2. Schreiben Sie eine Prozedur

```
newNextSquare : unit -> unit -> int
```

die bei jedem Aufruf einen neuen Generator für die Folge der Quadratzahlen liefert.

3. Schreiben Sie eine Prozedur

```
newGenerator : (int -> 'a) -> unit -> 'a
```

die zu einer Prozedur f einen Generator für die Folge

$$f(0),\ f(1),\ f(2),\ \ldots$$

liefert.

4. Schreiben Sie eine Prozedur

```
newNextPrime : unit -> unit -> int
```

die bei jedem Aufruf einen neuen Generator für die Folge $2, 3, 5, 7, \ldots$ der Primzahlen liefert.

Aufgabe 13.3 (Effiziente imperative Schlangen) Schreiben Sie eine Struktur, die imperative Schlangen gemäß der folgenden Spezifikation realisiert:

```
type 'a queue
val queue  : unit -> 'a queue
val insert : 'a * 'a queue -> unit
val head   : 'a queue -> 'a         (* Empty *)
val remove : 'a queue -> unit       (* Empty *)
val empty  : 'a queue -> bool
```

Die Operation `queue` liefert eine neue Schlange, die noch keine Einträge enthält. Die Operation `insert` trägt einen Wert in eine Schlange ein. Die Operation `head` liefert den ältesten Eintrag in einer Schlange. Die Operation `remove` entfernt den ältesten Eintrag aus einer Schlange. Die Operation `empty` testet, ob eine Schlange Einträge enthält.

Beachten Sie unbedingt die folgenden Hinweise:

1. Alle Operationen sollen konstante Laufzeit haben.
2. Verwenden Sie imperativen Listen gemäß der folgenden Typdeklaration:

```
datatype 'a ilist = Nil | Cons of 'a * 'a ilist ref
```

3. Stellen Sie eine imperative Schlange durch ein Paar des Typs

```
'a ilist ref * 'a ilist ref ref
```

dar, das aus einer imperativen Liste und einem Zeiger auf die letzte Referenz der Liste besteht (wenn die Schlange Einträge enthält).

Aufgabe 13.4 (Reversieren von Reihungen) Schreiben Sie eine Prozedur

```
reverse : 'a array -> unit
```

die eine Reihung reversiert. Die Prozedur soll mit einer Schleife und ohne Rekursion realisiert werden. Verwenden Sie die Prozedur `swap` aus Abschnitt 13.5.

Aufgabe 13.5 (Rotieren von Reihungen) Sie sollen eine Prozedur

```
rotate : 'a array -> unit
```

schreiben, die die Komponenten einer nichtleeren Reihung um eine Position nach rechts schiebt. Dabei soll die letzte Komponente an die Stelle der ersten Komponente rücken.

1. Schreiben Sie `rotate` mithilfe einer iterativ rekursiven Hilfsprozedur `rotate'`.
2. Schreiben Sie `rotate` mithilfe einer Schleife.

Aufgabe 13.6 (Binäre Suche in Reihungen) Eine Reihung des Types `int array` heißt sortiert, wenn ihre Komponenten aufsteigend sortiert sind. Schreiben Sie eine Prozedur

```
member : int array -> int -> bool
```

die zu einer sortierten Reihung a und einer Zahl x entscheidet, ob eine Komponente von a den Wert x hat. Für Reihungen der Länge n soll `member` die Laufzeit $O(\log n)$ haben.

Die logarithmische Laufzeit läßt sich mithilfe von binäre Suche erreichen, eine Technik, die wir bereits im Zusammenhang mit Rot-Schwarz-Bäumen kennengelernt haben. Dazu stellen wir uns die sortierte Reihung als eine Folge vor, die von links nach rechts läuft und gehen wie folgt vor:

1. Bestimme einen Index m, der etwa in der Mitte der Reihung liegt.
2. Wenn x der Wert der m-ten Komponente ist, liefere `true`.
3. Wenn x kleiner als der Wert der m-ten Komponente ist, suche in der linken Teilreihung weiter.
4. Wenn x größer als der Wert der m-ten Komponente ist, suche in der rechten Teilreihung weiter.
5. Wenn die zu durchsuchende Teilreihung leer ist, liefere `false`.

Eine Teilreihung können wir durch das Paar aus ihrem kleinsten und größten Index darstellen.

Aufgabe 13.7 (Reversieren von imperativen Listen) Schreiben Sie eine Prozedur

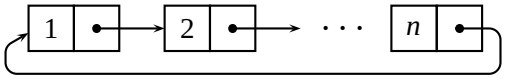
```
reverse : 'a ilist -> 'a ilist
```

die eine imperative Liste reversiert indem sie die Zeiger umsetzt und den letzten Kasten liefert. Die Prozedur soll keine neuen Referenzen einführen. Für zyklische Listen soll die Prozedur divergieren.

Aufgabe 13.8 (Zyklische imperative Listen) Schreiben Sie eine Prozedur

```
circle : int -> int ilist
```

die zu $n \geq 1$ eine zyklische Liste mit n Knoten liefert, die mit den Zahlen $1, \ldots, n$ markiert sind:



Die Prozedur soll den mit 1 markierten Knoten liefern.

Aufgabe 13.9 (Größe von imperativen Listen) Schreiben Sie eine Prozedur

```
size : 'a ilist -> int
```

die die Größe einer imperativen Liste liefert (das ist die Anzahl der Referenzen der Liste). Schreiben Sie dazu zuerst eine Prozedur

```
reflist : 'a ilist -> 'a ilist ref list
```

die die Liste aller Referenzen einer imperativen Liste liefert. Beide Prozeduren sollen auch für zyklische Listen funktionieren. Geben Sie die Laufzeit Ihrer Prozedur `size` an.

Aufgabe 13.10 (Listenreversion mit Schleifen) Schreiben Sie mithilfe einer Schleife eine nichtrekursive Prozedur

```
reverse : 'a list -> 'a list
```

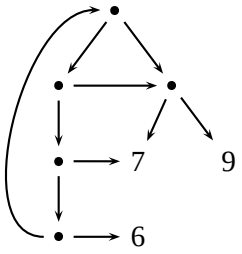
die Listen mit linearer Laufzeit reversiert.

Aufgabe 13.11 (Statische Semantik von F mit Referenzen) Sei F wie in Abschnitt 13.9 gezeigt um Referenzen erweitert. In Abschnitt 11.2 wird die statische

Semantik von F mithilfe von Inferenzregeln definiert. Geben Sie die für Referenzen zusätzlich erforderlichen Regeln an.

Aufgabe 13.12 (Fakultät in F plus Referenzen) Schreiben Sie in dem um Referenzen erweiterten F einen zulässigen Ausdruck, der zu einer Prozedur evaluiert, die die Fakultätsfunktion berechnet.

Aufgabe 13.13 (Cons-Zellen) Cons-Zellen sind die Hauptdatenstruktur der Programmiersprache Lisp, die um 1959 von John McCarthy entworfen wurde. Dabei handelt es sich um imperative Objekte, die Graphen darstellen können. Hier ist ein Beispiel:



Die Knoten dieser Graphen sind entweder ganze Zahlen, oder Cons-Zellen (mit • dargestellt). Von einer Cons-Zelle gehen immer genau zwei Kanten aus, wobei zwischen der ersten und der zweiten Kante unterschieden wird. Die Kanten können umgesetzt werden.

1. Schreiben Sie eine Struktur `Cons`, die Cons-Zellen mit der folgenden Signatur implementiert:

```
eqtype cons

val number:    int -> cons
val cons:      cons * cons -> cons

val isNumber:  cons -> bool
val value:     cons -> int           (* Domain *)
val fst:       cons -> cons          (* Domain *)
val snd:       cons -> cons          (* Domain *)

val setFst:    cons * cons -> unit   (* Domain *)
val setSnd:    cons * cons -> unit   (* Domain *)
```

   Dabei soll die Operation `cons` eine neue Cons-Zelle erzeugen, deren Kanten auf die als Argumente übergebenen Objekte zeigen. Die Operationen `fst` und `snd` sollen den erste beziehungsweise zweiten Nachfolger einer Cons-Zelle liefern. Die Prozeduren `setFst` und `setSnd` sollen die von einer Cons-Zelle ausgehenden Kanten umsetzen. Die Operation `value` soll für ein mit `number` *n* erzeugtes Objekt die Zahl *n* liefern.

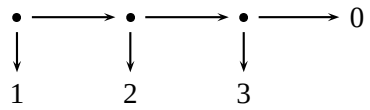
Gleichheit soll für den Typ `cons` wie folgt definiert sein:

a) Jeder Aufruf der Operation `cons` liefert eine neue Cons-Zelle.

b) Die von `number` gelieferten Objekte sind genau dann gleich, wenn sie die gleiche Zahl tragen.

2. Schreiben Sie einen Ausdruck, der den oben gezeigten Graphen erzeugt und die oberste Cons-Zelle liefert.

3. Mit Hilfe von Cons-Zellen kann man Listen mit ganzzahligen Elementen darstellen. Die Liste `[1,2,3]` wird dabei wie folgt dargestellt:



Schreiben Sie eine Prozedur

```
listToCons : int list -> cons
```

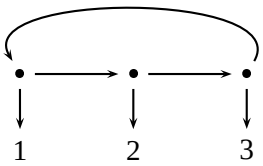
die zu einer Liste ein Objekt liefert, das die Liste darstellt.

4. Schreiben Sie eine Prozedur

```
isList : cons -> bool
```

die testet, ob ein Objekt eine Liste darstellt. Die Prozedur darf divergieren, wenn das Objekt einen zyklischen Graph darstellt.

5. Mit Cons-Zellen kann man sogenannte Kreise darstellen:



Schreiben Sie eine Prozedur

```
circle : cons -> unit
```

die eine durch ein Objekt dargestellte nichtleere Liste zu einem Kreis verkettet. Die Prozedur darf divergieren, wenn ihr Argument keine Liste darstellt.

6. Schreiben Sie eine Prozedur

```
isCircle : cons -> bool
```

die testet, ob ein Objekt einen Kreis darstellt. Die Prozedur darf divergieren, falls das nicht der Falls ist.