Kapitel 13

Imperative Objekte

Der Zustand eines physikalischen Objekts kann sich im Laufe der Zeit verändern. Stellen sie sich eine Uhr mit einem Sekundenzeiger vor. Offensichtlich wechselt die Uhr ihren Zustand sekündlich. Wenn Sie die Uhr durch Drehen der Zeiger anders einstellen, bedeutet das, dass der Zustand der Uhr durch eine externe Einwirkung verändert wird.

Mathematische Objekte sind gedankliche Objekte, die keiner zeitlichen Veränderung unterworfen sind (denken Sie an die Zahl 12). Anders als bei physikalischen Objekten macht es bei mathematischen Objekten keinen Sinn, von der Lebensdauer eines Objekts zu sprechen. Mathematische Objekte existieren unabhängig vom Begriff der Zeit.

Computer sind physikalische Objekte, die uns in die Lage versetzen, gedankliche Objekte für eine gewisse Zeit zu realisieren. Stellen Sie sich dazu einen geöffneten Interpreter auf dem Desktop Ihres Computers vor. Wenn Sie eine Deklaration eingeben, realisiert der Interpreter die dadurch beschriebenen Objekte (z. B. eine Prozedur oder eine Liste).

Mit Computern können wir gedankliche Objekte realisieren, die ähnlich wie physikalische Objekte Zustände haben, die sich im Laufe der Zeit ändern können. Ein gutes Beispiel für ein solches Objekt ist ein auf Ihrem Computer realisierter Interpreter. Zunächst müssen Sie den Interpreter starten (erzeugen), indem sie das den Interpreter beschreibende Programm zur Ausführung bringen. Danach können Sie den Zustand des Interpreters durch die Eingabe von Deklarationen verändern. Schließlich können Sie den Interpreter beenden (eliminieren). Der Interpreter hat dann für die Zeistspanne zwischen seiner Erzeugung und seiner Elimination existiert. Jedes Mal, wenn Sie das den Interpreter beschreibende Programm zur Ausführung bringen, erzeugen sie einen neuen Interpreter. Auf einem Computer können gleichzeitig mehrere Interpreter existieren.

Die bisher betrachteten Standard ML Programme realisieren bei ihrer Ausführung

intern nur unveränderliche Objekte. In diesem Kapitel führen wir Programmierkonstrukte ein, mit denen veränderliche Objekte realisiert werden können. Für die Erklärung der neuen Konstrukte müssen wir unser Ausführungsmodell so erweitern, dass die Ausführungsmaschine über einen *Speicher* verfügt.

Im Kontext von Standard ML bezeichnen wir unveränderliche interne Objekte als **funktional** und veränderliche interne Objekte als **imperativ**.

13.1 Speicher und Referenzen

Zunächst benötigen wir das Konzept des **Speichers**. Unter einem Speicher stellen wir uns einen Kasten vor, der in so genannte **Zellen** unterteilt ist. In jeder Zelle kann ein Wert abgelegt werden. Die Zellen eines Speichers sind durchnummeriert. Die Zellennummern werden als **Referenzen** bezeichnet. Ein Speicher stellt drei Operationen zur Verfügung:

- **Allokation.** Diese Operation legt einen Wert in eine bisher nicht belegte Zelle und liefert die Referenz der Zelle.
- **Dereferenzierung.** Diese Operation liefert zu einer Referenz den in der entsprechenden Zelle abgelegten Wert.
- **Zuweisung.** Diese Operation legt einen Wert in die durch eine Referenz identifizierte Zelle. Falls die Zelle bereits mit einem Wert belegt war, geht dieser verloren.

Die Laufzeit dieser Operationen ist konstant.

Das in Kapitel 1 eingeführte Ausführungsmodell erweitern wir jetzt dahingehend, dass wir die Ausführungsmaschine mit einem Speicher versehen. Die damit verfügbaren Speicheroperationen stellt Standard ML 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**. Eine Referenz des Typs `'a ref` identifiziert eine Zelle, in der Werte des Typs `'a` abgelegt werden können. Durch die abstrakten Referenztypen wird gewährleistet, dass nur solche Referenzen im Umlauf sind, die belegte Zellen identifizieren.

Sie wissen jetzt genug, um die folgende Interaktion mit einem Interpreter zu verstehen:

```
val r = ref 0
val r = ref 0 : int ref

(!r, r:=25, !r)
(0, (), 25) : int * unit * int

r
ref 25 : int ref

!r+6
31 : int
```

Im Zusammenhang mit Speicheroperationen spielt die Reihenfolge, in der die Teilausdrücke eines zusammengesetzten Ausdrucks ausgeführt werden, eine entscheidende Rolle. In Standard ML werden Teilausdrücke grundsätzlich in der textuell angegeben Ordnung ausgeführt (von links nach rechts, von oben nach unten).

Hier sind einige gebräuchliche Sprechweisen:

- Unter dem **Wert einer Referenz** verstehen wir den Wert in der durch die Referenz identifizierten Zelle. Wir sagen auch, dass eine Referenz auf ihren Wert **zeigt**.
- Wir sagen, dass wir eine Referenz auf einen Wert **setzen**, wenn wir den Wert mithilfe der Zuweisungsoperation in die durch die Referenz identifizierte Zelle legen. Wir sagen auch, dass wir der Referenz den Wert **zuweisen**.
- Referenzen bezeichnen wir auch als **Adressen**. Dereferenzierung und Zuweisung bezeichnen wir auch als **Lesen** und **Schreiben**.

Die Allokationsoperation verändert den **Zustand** des Speichers, da sie eine bisher nicht benutzte Zelle mit einem Wert belegt. Man sagt auch, das die Allokationsoperation eine Zelle **alloziert**. Die Zuweisungsoperation kann den Zustand des Speichers dadurch ändern, dass sie eine bereits allozierte Zelle mit einem anderen Wert belegt.

Wenn die Ausführung einer Phrase Speicheroperationen beinhaltet,können diese den Zustand des Speichers verändern. Wenn dies der Fall ist, sagen wir, dass die Ausführung der Phrase einen **Speichereffekt** hat.

Während einer Programmausführung stellt jeder Wert, der allozierte Referenzen enthält, zusammen mit dem Speicher ein imperatives Objekt dar. Der jeweilige Zustand des Objektes wird durch den jeweiligen Zustand des Speichers dargestellt. Das prototypische Beispiel für ein imperatives Objekt ist eine Speicherzelle.

Es ist wichtig, zwischen einer Referenz und einer durch die Referenz identifizierten Speicherzelle zu unterscheiden. Die Referenz ist ein Wert. Wie jeder Wert

ist eine Referenz unveränderlich. Eine Speicherzelle ist kein Wert, sie kann aber mit einem Wert belegt werden. Der Zusammenhang zwischen einer Referenz (ein mathematisches Objekt) und einer Speicherzelle (ein Teil eines laufenden Interpreters) wird durch den Interpreter für den Zeitraum einer Sitzung hergestellt.

Jede Allokation liefert eine neue Referenz. Der Vergleich

```
ref 0 = ref 0
```

liefert den Wert `false`, da die beiden Teilausdrücke verschiedene Referenzen liefern. Dagegen liefert der Vergleich

```
!(ref 0) = !(ref 0)
```

den Wert `true`, da diesmal die Werte der Referenzen verglichen werden (also die Werte in den durch die Referenzen identifizierten Zellen).

In diesem Zusammenhang ist es wichtig, die Ausgabe des Interpreters richtig zu interpretieren. Dieser beschreibt Referenzen mit dem Wort `ref` und ihrem aktuellen Wert im Speicher:

```
(ref 0, ref 0)
(ref 0, ref 0) : int ref * int ref

let val r = ref 0 in (r,r) end
(ref 0, ref 0) : int ref * int ref
```

Bei der ersten Interaktion handelt es sich um zwei verschiedene Referenzen, bei der zweiten zweimal um dieselbe Referenz. Die Ausgabe des Interpreters ist aber in beiden Fällen die gleiche.

Syntaktisches

Die für die Speicheroperationen verwendeten Wörter `ref`, `!` und `:=` spielen die Rolle von Operatoren. Bei `ref` und `!` handelt es sich um einstellige Operatoren (ein bereits bekannter einstelliger Operator ist ~). Bei `:=` handelt es sich um einen Infixoperator, der auf einer höheren Rangstufe steht als die bisher eingeführten Operatoren. Einstellige Operatoren haben syntaktisch denselben Status wie Bezeichner und Konstanten. Daher kann man bei

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

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

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

Before-Notation

Im Zusammenhang mit den Speicheroperationen ist die abgeleitete syntaktische Form

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

oft bequem. Diese Form wertet zuerst den Ausdruck e_1 und dann den Ausdruck e_2 aus und liefert den bei der Auswertung von e_1 erhaltenen Wert. Beispielsweise weist der Ausdruck

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

der Referenz `r` den Wert `x` zu und liefert den bisherigen Wert der Referenz als Ergebnis. Die Prozedur

```
fun swap r r' = r:= (!r' before r':= !r)
val swap : 'a ref → 'a ref → unit
```

vertauscht die unter zwei Referenzen abgelegten Werte.

Ref in Mustern

Das Wort `ref` kann in Mustern wie ein Konstruktor benutzt werden. Ein Muster `ref` x trifft jede Referenz und bindet die Variable x an den aktuellen Wert der Referenz. Hier sind Beispiele:

```
fun deref (ref x) = x
val deref : 'a ref → 'a

deref (ref 7)
7 : int

fun test (ref 1) = true
  | test    _    = false
val test : int ref → bool

(test (ref 1), test (ref 2))
(true, false) : bool * bool
```

13.2 Prozeduren mit Zustand

Mit Referenzen sind wir in der Lage, eine Prozedur

```
counter : unit → int
```

zu schreiben, die mitzählt, wie oft sie aufgerufen wurde und bei ihrem n-ten Aufruf die Zahl n liefert. Wir realisieren `counter` zunächst wie folgt:

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

Damit bekommen wir die folgende Interaktion:

```
(counter(),  counter(), counter())
(1, 2, 3) : int * int * int

(counter(),  counter(), counter())
(4, 5, 6) : int * int * int

counter() + counter()
15 : int
```

Wie jede Prozedur liefert `counter` ein Ergebnis. Zusätzlich hat ein Aufruf von `counter` den Speichereffekt, dass der Wert der Referenz `r` um eins erhöht wird. Während einer Programmausführung stellt die Prozedur `counter` zusammen mit dem Speicher ein imperatives Objekt dar, da sie eine Referenz auf eine Zelle des Speichers enthält. Wir sagen auch, das `counter` eine **Prozedur mit Zustand** ist.

Die vorliegende Realisierung der Prozedur `counter` ist unschön, da es über den Bezeichner `r` möglich ist, den Zustand der Prozedur von außen zu ändern. Diese Sicherheitslücke können wir schließen, wenn wir `counter` wie folgt deklarieren:

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

Wir sagen, dass diese Realisierung der Prozedur `counter` den Zustand **enkapsuliert** (d.h. einkapselt).

Hier ist eine Prozedur

```
newCounter : int → (unit → int)
```

die bei jedem Aufruf eine neue Zählprozedur erzeugt:

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

Der Anfangswert der zu erzeugenden Zählprozedur kann beliebig vorgegeben werden. Wir bekommen die folgenden Interaktionen:

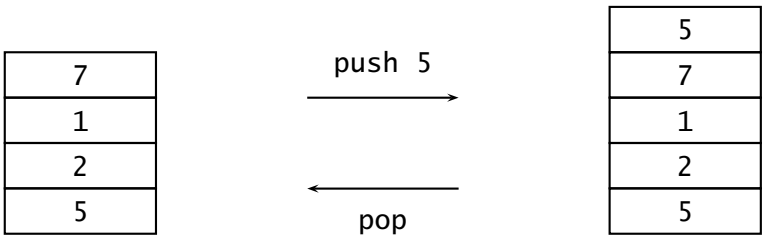


Abbildung 13.1: Grafische Darstellung von Stapeln

```
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 obigen Prozeduren zeigen beispielhaft, wie wir ausgehend von Speicherzellen komplexere imperative Objekte beschreiben können.

13.3 Stapel

Unter einem Stapel (englisch stack) versteht man in der Programmierung ein imperatives Objekt, auf dem mehrere Werte gestapelt werden können. Der zuletzt abgelegte Wert liegt dabei zuoberst. Für Stapel betrachten wir 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

Wir spezifizieren Stapel mit der in Abbildung 13.2 gezeigten Typabstraktion. 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)
() : unit

push(s,2)
() : unit

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

13.4 Reihungen

Eine Reihung (englisch array) ist ein imperatives Objekt, das aus einer Anzahl von Speicherzellen besteht, die als **Komponenten** bezeichnet werden. Die Komponenten einer Reihung werden durch natürliche Zahlen identifiziert, die als **Indizes** bezeichnet werden und sich durch Durchnummerierung beginnend mit Null ergeben. Eine Reihung kann grafisch entweder horizontal

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

oder vertikal dargestellt werden:

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

Die dargestellte Reihung hat vier Komponenten, die mit den Indizes 0 bis 3 identifiziert werden. Jede der vier Komponente enthält eine Zeichenreihe. Generell gilt, dass alle Komponenten einer Reihung Werte des gleichen Typs enthalten müssen.

Wir spezifizieren Reihungen mit der in Abbildung 13.3 gezeigten Typabstraktion. Die Operation `array` liefert zu n und x eine Reihung mit n Komponenten, die zunächst alle mit dem initialen Wert x belegt sind. Mit der Operation `sub` kann der Wert einer Komponente gelesen werden, mit der Operation `update` kann er gesetzt werden. Die Komponente werden dabei durch ihren Index identifiziert. Die Operation `length` liefert die Anzahl der Komponenten einer Reihung (die so genannte **Länge** der Reihung). Unsere Modellimplementierung realisiert eine Reihung durch eine Liste von Referenzen.[1]

Die Standardstruktur `Array` realisiert Reihungen gemäß unserer Spezifikation, ergänzt um zusätzliche Operationen. Die Operationen `sub` und `update` werden von der Standardstruktur mit *konstanter Laufzeit* realisiert (unabhängig von der Länge der Reihung).[2]

[1] Eine Realisierung von Reihungen durch ein Tupel von Referenzen wäre wegen der fixen Länge von Reihungen angemessener. Allerdings benötigt man dafür einen Typkonstruktor für homogene Tupel beliebiger Stelligkeit, wie er von der Standardstruktur `Vector` zur Verfügung gestellt wird.

[2] In der bisher vorgestellten Teilsprache von Standard ML ist eine Realisierung von Reihungen mit konstanter Laufzeit für `sub` und `update` übrigens nicht möglich. Eine Realisierung mit logarithmischer Laufzeit (relativ zur Länge der Reihung) ist mit Rot-Schwarz-Bäumen möglich.

```
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

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

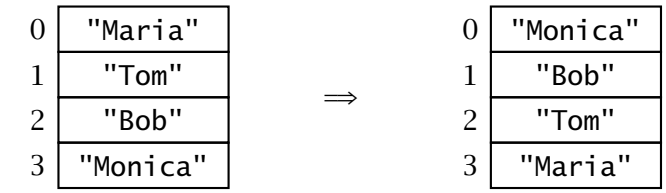
Das Programmieren mit Reihungen erfordert einige neue Techniken. Wichtig ist insbesondere, dass man mit den Indizes einer Reihung rechnen kann. Wir betrachten zwei Beispiele.

Reversieren von Reihungen

Wir wollen eine Prozedur

```
val reverse : 'a array → unit
```

schreiben, die die Werte in den Komponenten einer Reihung so vertauscht, dass sie in reversierter Ordnung erscheinen. Zum Beispiel:



Wir beginnen mit einer Prozedur

```
fun swap a i j = Array.update(a, i, Array.sub(a,j)) before
                 Array.update(a, j, Array.sub(a,i))
```

```
swap : 'a array → int → int → unit
```

die die Werte zweier Komponenten 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 (l steht für lower und u für upper). Damit können wir die angestrebte Prozedur `reverse` wie folgt realisieren:

```
fun reverse a = reverse' a 0 (Array.length a - 1)
```

Diese Prozedur reversiert eine Reihung der Länge n mit der Laufzeit $\Theta(n)$.

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. Immer wenn dies für ein Element der Fall ist, setzen wir die entsprechende Komponente auf `true`. Nachdem wir alle Elemente der Liste getestet haben, liefern wir `true` genau dann, wenn alle Komponenten der Reihung den Wert `true` haben.

```
fun test xs l u =
    let
        val a = Array.array(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 Prozedur `Array.foldl` arbeitet analog zur Faltungsprozedur für Listen.

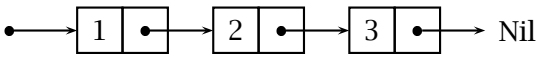
Man kann die Prozedur `test` auch ohne imperative Objekte realisieren. Beispielsweise kann man die Liste zuerst sortieren und danach die Intervall-Eigenschaft testen. Die Realisierung mit einer Reihung ist aber verblüffend einfach und hat zudem lineare Laufzeit bezüglich der Länge der Liste. Dazu kommt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

die Laufzeit für die Erzeugung der Reihung (linear bezüglich der Breite des Intervalls).

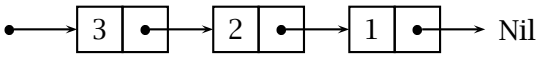
13.5 Imperative Listen

Mit Referenzen kann man die unterschiedlichsten imperative Datenstrukturen beschreiben. Unsere zwei ersten Beispiele waren Stapel und Reihungen. Als drittes Beispiel betrachten wir imperative Listen.

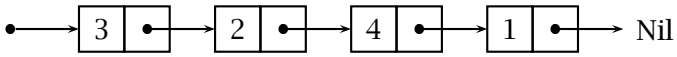
Eine imperative Liste mit den Elementen 1, 2, 3 stellt man üblicherweise grafisch wie folgt dar:



Die Pfeile mit dem dicken Ausgangspunkt bezeichnet man als **Zeiger** (englisch pointer oder link). Jeder der vier Zeiger beschreibt eine Teilliste. Die Kästen bezeichnet man als **Knoten** (englisch node). Für jedes Element der Liste gibt es einen Kasten. Es besteht die Möglichkeit, die Zeiger einer imperativen Liste bei Bedarf auf andere Knoten umzusetzten. Dadurch hat man die Möglichkeit, bestehende Knoten auf unterschiedlich Weise miteinander zu verzeigern. Beispielsweise kann man die obigen Knoten auch wie folgt verzeigern:



Wenn wir einen Knoten mit dem Element 4 hinzunehmen, können wir die folgende imperative Liste bilden:



Wir beschreiben imperative Listen mit zwei Typdeklarationen:

```
datatype 'a state = Nil | N of 'a * 'a state ref
type     'a ilist = 'a state ref
```

Eine imperative Liste stellen wir durch eine Referenz dar, die auf einen **Zustand** (englisch state) zeigt. Ein Zustand ist entweder der Wert `Nil` oder ein Knoten. Ein Knoten wird mit dem Konstruktor `N` aus einem Element und einer imperativen Liste gebildet. Die erste der oben gezeigten Listen können wir wie folgt realisieren:

```
val xs = ref(N(1, ref(N(2, ref(N(3, ref Nil))))))
```

Jeder Zeiger in der grafischen Darstellung einer Liste ist durch eine Speicherzelle realisiert. Der fette Ausgangspunkt eines Zeigers symbolisiert die Referenz der Zelle. Der Pfeil des Zeigers zeigt auf den Zustand, der in der Speicherzelle liegt.

Die Prozedur

```
fun empty (ref Nil) = true
  | empty      _    = false
val empty : 'a ilist → bool
```

testet, ob eine imperative Liste leer ist. Die Prozedur

```
fun ilist () = ref Nil
val ilist : unit → 'a ilist
```

liefert eine neue, zunächst leere, imperative Liste. Die Prozeduren

```
fun head (ref(N(x,_))) = x
  | head       _       = raise Empty
val head : 'a ilist → 'a
fun tail (ref(N(_,xr))) = xr
  | tail       _        = raise Empty
val tail : 'a ilist → 'a ilist
```

liefern den Kopf beziehungsweise den Rumpf einer imperativen Liste. Die Prozedur

```
fun cons x xr = ref(N(x,xr))
val cons : 'a → 'a ilist → 'a ilist
```

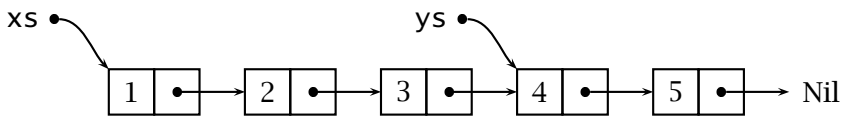
liefert zu einem Wert *x* und zu einer imperativen Liste *xr* eine neue imperative Liste, deren Kopf zunächst *x* und deren Rumpf zunächst *xr* ist. Die Prozedur

```
fun append xs ys = if empty xs then xs:= !ys
                   else append (tail xs) ys
val append : 'a ilist → 'a ilist → unit
```

setzt den letzten Zeiger einer imperativen Liste *xs* auf den ersten Zustand einer imperativen Liste *ys*. Als Beispiel betrachten wir die Deklarationen

```
val xs = cons 1 (cons 2 (cons 3 (ilist())))
val ys = cons 4 (cons 5 (ilist()))
val  _ = append xs ys
```

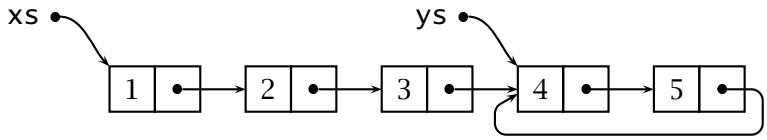
die die folgende Situation liefern:



Wenn wir jetzt den Audruck

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

auswerten, bekommen wir die folgende zyklische Verzeigerung:



Die imperative Listen `xs` und `ys` sind jetzt zyklisch. Solche durch Speicherzellen laufende Zyklen kann es bei funktionalen Listen nicht geben.

Hase-Igel-Algorithmus

Der **Hase-Igel-Algorithmus** testet, ob eine imperative Liste zyklisch ist. Wir erklären den Algorithmus anhand der grafischen Darstellung von imperativen Listen. Wir fassen die grafische Darstellung einer Liste als ein Spielfeld auf, in dem wir zwei Spielfiguren bewegen, die als Igel und Hase bezeichnet werden. Zu Beginn wird der Igel auf die erste Referenz und der Hase auf die zweite Referenz der Liste gesetzt. Bei jedem Spielzug wird der Igel um eine Position und der Hase um zwei Positionen vorgerückt. Es wird solange gezogen, bis einer der folgenden Fälle eintritt:

1. Der Hase kann nicht weitergeschoben werden, da seine Referenz auf `Nil` zeigt. In diesem Fall ist die Liste nicht zyklisch.

2. Der Hase und der Igel stehen beide auf derselben Referenz. In diesem Fall ist die Liste zyklisch.

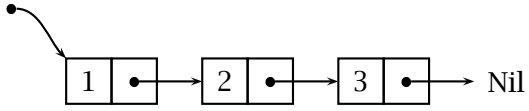
Wir überlegen uns, warum der Algorithmus auch bei zyklischen Listen terminiert. Sei die Liste also zyklisch. Dann befinden sich Hase und Igel nach endlich vielen Schritten beide im Zyklus der Liste. Wir betrachten jetzt den Abstand vom Hasen zum Igel. Da der Hase bei jedem Schritt um zwei Positionen und der Igel nur um eine nach vorne rückt, rückt der Hase bei jedem Schritt um eine Position näher an den Igel heran. Also holt der Hase den Igel schließlich ein und beide stehen auf derselben Referenz. Damit terminiert der Algorithmus. Da der Igel jede Referenz der Liste höchstens einmal besucht, hat der Algorithmus lineare Laufzeit bezüglich der Anzahl der Referenzen der Liste.

Wir realisieren den Hase-Igel-Algorithmus wie folgt:

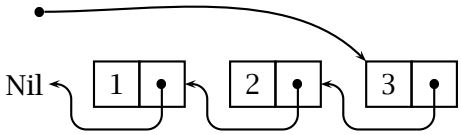
```
fun cyclic' i h = i=h orelse cyclic' (tail i) (tail(tail h))
val cyclic' : 'a ilist → 'a ilist → bool

fun cyclic xs = cyclic' xs (tail xs) handle Empty => false
val cyclic : 'a ilist → bool
```

Reversieren

Wir können eine imperative Liste reversieren, indem wir ihre Zeiger umsetzen. Als Beispiel betrachten wir die Liste



Durch Umsetzen der Zeiger erhalten wir



Die Reversion einer imperativen Liste liefert keine neue Liste, sondern ändert den Zustand der gegebenen Liste (und ihrer Teillisten). Beim Reversieren werden keine neuen Zellen alloziert. Nachdem eine Liste reversiert wurde, bringt erneutes Reversieren sie wieder in den Ausgangszustand.

Hier ist eine Prozedur zum Reversieren von Listen:

```
fun reverse' s      Nil       = s
  | reverse' s (s' as N(_,r)) = reverse' s' (!r before r:=s)
val reverse' : 'a state → 'a state → 'a state

fun reverse xs = xs:= reverse' Nil (!xs)
val reverse : 'a ilist → unit
```

13.6 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 e_1 do e_2
```

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 e_1 do e_2  =  if e_1 then while e_1 do e_2 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 e_1 do e_2

  ⇒

let
  fun loop () = if e_1 then (e_2 ; 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 **imperative 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).

13.7 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 () => ()) ;
    1 + (!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 → int) ref
   ```

3. Der Bezeichner `r` kann nicht polymorph typisiert werden, da die Deklaration von `r` ambig ist (da ihre rechte Seite eine Applikation 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.

13.8 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:

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

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. Referenzen stellen wir durch natürliche Zahlen dar. 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:

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

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). Die Auswertung eines Ausdrucks hat einen Speichereffekt, wenn sich der Endzustand vom Anfangszustand unterscheidet. Ein Speichereffekt 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

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

für

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

Dabei soll $S, V \vdash e \Longrightarrow v, S'$ 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:

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

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

$$\frac{\begin{array}{c} S, V \vdash e_1 \Longrightarrow (x, e, V'),\ S_1 \quad S_1, V \vdash e_2 \Longrightarrow v_2, S_2 \\ S_2,\ V' + \{x \mapsto v_2\} \vdash e \Longrightarrow v, S' \end{array}}{S, V \vdash e_1\ e_2 \Longrightarrow v, S'}$$

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:

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

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

$$\frac{\begin{array}{c} S, V \vdash e_1 \Longrightarrow r, S_1 \quad S_1, V \vdash e_2 \Longrightarrow v, S_2 \\ r \in \mathit{Dom}\, S_1 \qquad S' = S_2 + \{r \mapsto v\} \end{array}}{S, V \vdash e_1 := e_2 \Longrightarrow 0, S'}$$

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:

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

Glossar

Funktionale und imperative Objekte.

Speicher (Zellen, Referenzen (Adressen); Allokation, Dereferenzierung (Lesen), Zuweisung (Schreiben); Wert einer Referenz, Referenz zeigt auf einen Wert; eine Referenz auf einen Wert setzten, einer Referenz einen Wert zuweisen; Zustand des Speichers; allozierte Zellen).

Typkonstruktor `ref` und Referenztypen; Operatoren `ref`, `!` und `:=`; Before-Notation.

Ausführung einer Phrase kann einen Speichereffekt haben.

Prozeduren mit Zustand; imperative Objekte sollen ihren Zustand enkapsulieren.

Stapel (stack, push, pop, top).

Reihungen (array, sub, update, length, foldl, foldr).

Imperative Listen (Zeiger, Zustand, Knoten; zyklische Listen, Hase-Igel-Algorithmus).

Schleifen (Bedingung, Rumpf; Zusammenhang mit iterativer Rekursion; funktionaler und imperativer Programmierstil).

Formale Semantik von Referenzen (Anfangs- und Endzustand des Speichers, Durchfädeln des Speicherzustands; Realisierung von Rekursion mithilfe von Referenzen).