

## Programmierung 1 (Wintersemester 2012/13)

---

### Lösungsblatt 13

(Kapitel 14 und 15)

---

**Hinweis:** Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.

*Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant, noch irrelevant.*

#### **Aufgabe 13.1** (*Quiz*)

Zu was werten die folgenden Ausdrücke aus? Machen Sie sich zuerst zu jedem Aufgabenteil Gedanken und vergewissern Sie sich erst dann mit dem Interpreter!

- (a) `ref 1 = ref 2`
- (b) `ref 1 = ref 1`
- (c) `val a = ref 1`  
`val b = a`  
`a = b`
- (d) `!(ref 0) = ref 2`
- (e) `! ref 1 = ! ref 1`
- (f) `map ! [ref 1, ref 2, ref 3] = [ref 1, ref 2, ref 3]`
- (g) `map ! [ref 1, ref 2, ref 3] = [1, 2, 3]`
- (h) `map ! []`
- (i) `map ! [] = []`
- (j) `ref [1, 2, 3] = [ref 1, ref 2, ref 3]`

*Lösung 13.1:*

- (a) `false`
- (b) `false`
- (c) `true`
- (d) `TypeClash: int <> int ref`
- (e) `!` wird auf `ref` angewendet, deshalb gibt es einen Typfehler.  
`!(ref 1) = !(ref 1)` würde `true` liefern
- (f) `TypeClash: int list <> int ref list`
- (g) `true`
- (h) `nil`
- (i) `true`
- (j) `TypeClash: int list ref <> int ref list`

# 1 Dieter-Schlau-Ecke

## Aufgabe 13.2 (*Dieter Schlau vs. Typsystem*)

- (a) Dieter Schlau versucht das Typsystem auszutricksen:

```
signature MYLIST = sig
  eqtype 'a mylist
  val empty : 'a mylist
  val cons : 'a -> 'a mylist -> 'a mylist
end

structure MyList :> MYLIST = struct
  type 'a mylist = 'a list
  val empty = []
  fun cons x tl = x :: tl
end
```

Da *mylist* als *eqtype* freigegeben wird, glaubt Dieter jetzt, den Interpreter durch den Vergleich zweier Werte des Type ( $int \rightarrow int$ ) *mylist* in Bedrängnis bringen zu können. Wo irrt Dieter?

- (b) Dieter Schlau hat folgende, noch effizientere Implementierung seiner „Write-only-Lists“ gefunden:

```
structure MyList :> MYLIST = struct
  type 'a mylist = int
  val empty = 0
  fun cons x tl = tl + 1
end
```

Wie unterscheidet sich - aus Sicht des Typsystems - die neue Datenstruktur von der aus Teil 1?

Lösung 13.2:

- (a) Auch wenn *mylist* als *eqtype* deklariert wurde, ist  $\alpha$  *mylist* nur dann Typ mit Gleichheit, wenn  $\alpha$  Typ mit Gleichheit ist.
- (b) Aus Sicht des Typsystems besteht kein Unterschied, auch wenn es hier theoretisch generell möglich wäre,  $\alpha$  *mylist* als Typ mit Gleichheit zu betrachten.

## Aufgabe 13.3 (*Vektorrechnung mit Dieter Schlau*)

**Warnung:** Diese Aufgabe ist fehlerhaft.

So ein Mist! Dieters Attacke auf das Typsystem in Aufgabe 2 ist gescheitert. Frustriert schlägt Dieter sein Buch auf:

*„Wir halten fest, dass Vektoren in Bezug auf Effizienz einen neuen Aspekt in die Sprache einbringen, der nicht auf die bisher eingeführten Sprachkonstrukte zurückgeführt werden kann.“*

Kapitel 14.4

~~Begeistert stellt Dieter fest, dass sich hier – zumindest bei geeignetem Verständnis von „Effizienz“ – das Buch austricksen lassen könnte. Natürlich freuen Sie sich darauf, Dieter helfen zu können.~~

Deklariert Sie eine Datenstruktur *Hector*, die folgende Bedingungen erfüllt:

- Die Signatur von *Hector* soll mindestens folgende Typen und Prozeduren beinhalten:

```
eqtype  $\alpha$  hector
val fromList :  $\alpha$  list  $\rightarrow$   $\alpha$  hector
val length :  $\alpha$  hector  $\rightarrow$  int
val sub :  $\alpha$  hector * int  $\rightarrow$   $\alpha$ 
val map : ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\alpha$  hector  $\rightarrow$   $\beta$  hector
```

- Die Deklaration verwendet keine Sprachkonstrukte, die erst in oder nach Kapitel 14.3 eingeführt werden.
- ~~*sub* hat konstante Komplexität zur Größenfunktion  $\lambda(h, n).n$~~
- *length* hat konstante Komplexität zur Größenfunktion des übergebenen *Hectors*.
- *fromList* hat lineare Komplexität in der Länge der übergebenen Liste.
- *map* hat lineare Komplexität in der Länge der Liste.
- Die Prozeduren *sub*, *fromList* und *map* liefern die gleichen Ergebnisse wie die entsprechenden Prozeduren von *Vector*.

### Lösung 13.3:

Wir verwenden intern Listen, benutzen aber zwei Tricks:

- Wir speichern die Länge der Liste, um *length* konstant zu machen.
- Bei *sub* durchsuchen wir immer die gesamte Liste, so dass die Laufzeit nicht von *n* abhängt (wohl aber von der Größe des *Hectors*.)

```

structure Hector :>
sig
  eqtype 'a hector
  val fromList : 'a list -> 'a hector
  val length : 'a hector -> int
  val sub : 'a hector * int -> 'a
  val map : ('a -> 'b) -> 'a hector -> 'b hector
end
= struct
  type 'a hector = int * 'a list
  fun fromList xs = (length xs, xs)
  fun length (l, xs) = l
  fun sub ((l, nil ), n) = raise Subscript
    | sub ((l, x::xr), 0) = (sub ((l, xr), ~1) handle Subscript => x)
    | sub ((l, x::xr), n) = sub ((l, xr), (n-1))
  fun map' f nil = nil
    | map' f (x::xr) = (f x) :: map f xr
  fun map f (l, xs) = (l, map' f xs)
end

```

### Aufgabe 13.4 (*exceptionally evil*)

Ist folgendes Programm gültig? Woran wird *x* gebunden bzw. welche Exception wird geworfen?

```

structure Evil :> sig
  exception Dieter
  exception Schlaue
  val test : unit -> int
end = struct
  exception Dieter
  exception Schlaue
  fun test () = raise Dieter
end

val x = Evil.test () handle Schlaue => 5

```

### Lösung 13.4:

*x* wird an 5 gebunden.