Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Aufgaben aus den Übungsgruppen 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]
```

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 $(int \to int)$ *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?

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.~~

Deklarieren Sie eine Datenstruktur $Hector$, die folgende Bedingungen erfüllt:

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

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

- 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 $Hector$s.
- $fromList$ 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 $Vector$.

Aufgabe 13.4 (*exceptionally evil*)
Ist folgendes Programm gültig? Woran wird x gebunden bzw. welche Exception wird geworfen?

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

val x = Evil.test () handle Schlau => 5
```