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

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 5

(Kapitel 3)

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.

1 Kapitel 3.8

Aufgabe 5.1
Geben Sie geschlossene Abstraktionen an, die die folgenden Typen haben:

(a) $(int * int) \to (bool * bool) \to (bool * int)$

(b) $\alpha * \alpha \to \beta * \beta \to bool \to (\beta * \alpha)$

(c) $int \to \alpha * (int * \alpha) \to \beta$

Geben Sie die Typen nicht explizit an und verwenden Sie keine Konstanten!
Hinweis: Schauen Sie sich die Typregel für *if* nochmals an!

Lösung 5.1:

(a) `fun f (a,b) (x,y) = if x then if y then (x,a * a) else (y,b + a) else (x,a + b)`

(b) `fun f (a,b) (x,y) c = if c then (x,a) else (y,b)`

(c) `fun f n (a,(m,b)) = f n (b,(m+n,a))`

Aufgabe 5.2
Wie kann man Prozeduren mit völlig neuem Ergebnistypen erzeugen, d.h. einem Typen, der nicht der Typ eines der Argumente ist?

Lösung 5.2:
SML typt divergierende Prozeduren ohne Hinweis auf den Typ dessen, was zurückgegeben wird, mit einem neuem völlig neuen Ergebnistypen.

Aufgabe 5.3
Geben Sie Fälle an, in denen Bezeichner ohne explizite Typangabe auf einen bestimmten Typen festgelegt werden.

Lösung 5.3:

- Addition, Subtraktion, Multiplikation, mod und div zweier Bezeichner legen diese auf *int* fest
- Operation mit einer Konstanten legt einen Bezeichner auf den Typen der Konstante fest
- Anwendung eines Bezeichners auf einen Ausdruck legt den Bezeichner als Prozedurtypen fest.

Aufgabe 5.4
Geben Sie Deklarationen an, die die monomorph getypte Bezeichner wie folgt deklarieren:

(a) $int * int * int$

(b) $bool \rightarrow bool$

(c) $int * int \rightarrow bool$

(d) $unit * unit \rightarrow bool$

Verwenden Sie dabei keine expliziten Typangaben, Operator- oder Prozeduranwendungen.

Lösung 5.4:

(a)
```
val a = (7,4,2)
```

(b)
```
val b = fn x => if x then x else x
```

(c)
```
val c = fn (0,0) => false
          | (x,y) => true
```

oder beispielsweise

```
val c = fn (a,b) => #2( [0,a,b] , false )
```

(d)
```
val d = fn ((),()) => true
```

Aufgabe 5.5 (*für harte Typen*)
Schreiben Sie eine Prozedur mit dem Typ $\forall\alpha, \beta.\alpha \rightarrow \beta$. Diskutieren Sie Ihren Vorschlag mit Ihren Kommilitonen.

Lösung 5.5:

```
fun f x = f x
```

2 Bezeichnerbindungen

Aufgabe 5.6
Betrachten Sie die folgenden Programme:

- ```
  fun x x = x
  fun y x = fn x => x
  ```

- ```
  fun x y z =
    if z > 3
      then
        let
          val y = (fn z => x 5 3 + z)
        in
          y z
        end
      else
        y
  ```

(a) Markieren Sie die definierenden Bezeichnerauftreten durch Überstreichen.

(b) Stellen Sie die lexikalischen Bindungen durch Pfeile dar.

(c) Geben Sie alle Bezeichner an, die in dem Ausdruck frei auftreten.

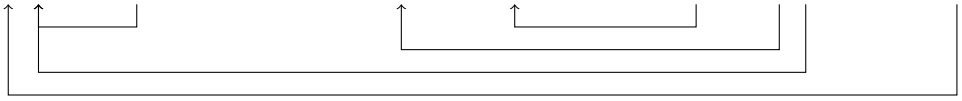
(d) Bereinigen Sie den Ausdruck durch Indizieren der gebundenen Bezeichnerauftreten.

(e) Geben Sie nun die statischen Bezeichnerbindungen an, die durch die semantische Analyse bestimmt werden.

Lösung 5.6:

(a)
- ```
  fun x̄ x̄ = x
  fun ȳ x̄ = fn x̄ => x
  ```
- ```
  fun x̄ ȳ z̄ =
      if z > 3
        then
          let
            val ȳ = (fn z̄ => x 5 3 + z)
          in
            y z
          end
        else
          y
  ```

(b)
- ```
  fun x x = x
  ```
  
  ```
  fun y x = fn x => x
  ```

- ```
  fun x y z = if z > 3 then let val y = (fn z => x 5 3 + z) in y z end else y
  ```
  

(c)
- keine
- keine

(d)
- ```
  fun x₁ x₂ = x₂
  fun y₁ x₃ = fn x₄ => x₄
  ```
  ```

- `fun x1 y1 z1 = if z1 > 3 then let val y2 = (fn z2 => x1 5 3 + z2) in y2 z1 end else y1`

(e)
- $x_1 : \forall\alpha.\, \alpha \to \alpha$
  $x_2 : \alpha$
  $y_1 : \forall\beta, \gamma.\, \beta \to \gamma \to \gamma$
  $x_3 : \beta$
  $x_4 : \gamma$
- $x_1 : int \to int \to int$
  $y_1 : int$
  $z_1 : int$
  $y_2 : int \to int$
  $z_2 : int$

3 Sortieren

Aufgabe 5.7 (*Induktive Argumente*)
Im Folgenden sollen Sie immer wieder induktive Argumente dafür angeben, dass die Sortieralgorithmen tatsächlich korrekt sind, also Listen sortieren.

Beispiel: *Insertionsort* arbeitet korrekt, da die leere Liste und die einelementige Liste nicht verändert werden und so wie sie sind, also insbesondere sortiert, wieder zurückgegeben werden. Außerdem wird jeweils ein Element aus der Liste entfernt und in den rekursiv bereits sortieren, also korrekten, Rest der Liste unter Beibehaltung der Ordnung eingefügt.

Finden Sie ein Argument dafür, warum Mergesort korrekt arbeitet!

Aufgabe 5.8
Im Folgenden sollen Sie den Sortieralgorithmus *Bubblesort* implementieren. Sei dazu die folgende Erklärung gegeben:

(a) Zunächst wird getestet, ob die Liste bereits sortiert ist.

(b) Ist die Liste leer oder enthält nur ein Element, so können Sie die Liste ohne Veränderung zurückgeben.

(c) Enthält die Liste zwei oder mehr Elemente, so werden die vordersten zwei verglichen und das größere steigt dann in der Liste wie in einer Blase (daher bubblesort) auf, bis es auf ein noch größeres Element trifft oder an das Ende der Liste gelangt.

(d) Das größere Element steigt dann wieder auf bis auf ein noch größeres Element trifft oder zum Ende der Liste gelangt.

(e) Dies wird fortgesetzt bis das größte Element am Ende der Liste angekommen ist.

(f) Die Liste wird solange mit den Schritten (*b*) bis (*d*) verändert bis sie sortiert ist.

Im Folgenden wird nun eine Schritt-für-Schritt Vorgehensweise angegeben, wie sie sinnvoll sein kann, um einen neuen Algorithmus zu verstehen und implementieren zu können.

(a) Suchen Sie sich ein bis zwei Beispielliste und gehen Sie die Schritte von Bubblesort für diese Liste durch, bis Sie sortiert sind (oder sie 100%-ig sicher sind, dass sie die Vorgehensweise vollständig verstanden haben. ;-)).

(b) Erklären Sie, wie Bubblesort eine beliebige Liste sortiert.

(c) Finden Sie ein induktives Argument dafür, dass Bubblesort tatsächlich Listen sortiert.

(d) Schreiben Sie eine Prozedur *sorted*: *int list* $\to$ *bool*, die testet, ob eine Liste sortiert ist.

(e) Schreiben Sie eine regelbasierte Prozedur *bubble*: *int list* → *int list*, die die Liste unverändert zurückgibt, falls sie weniger als zwei Elemente enthält. Ansonsten soll sie die ersten beiden Elemente vergleichen und das größere davon solange durch die Liste nach oben tragen, bis es es auf ein größeres trifft oder das Ende der Liste erreicht. Findet man ein größeres so steigt dieses weiter auf (siehe Punkte (b) bis (d) des Algorithmus).

(f) Implementieren Sie nun *bubblesort*: *int list* → *int list*, die auf eine Liste solange rekursiv bubble anwendet, bis sie sortiert ist.

Lösung 5.8:

(d)
```
fun sorted nil = true
  | sorted [x] = true
  | sorted (x::y::xs) = if x <= y then sorted(y::xs) else false
```

(e)
```
fun bubble nil = nil
  | bubble [x] = [x]
  | bubble (x::y::xs) = if x > y then y :: bubble (x::xs) else x :: bubble(y::xs)
```

(f)
```
fun bubblesort xs = if sorted xs then xs else bubblesort (bubble xs)
```

Aufgabe 5.9
Im Folgenden sollen Sie den Sortieralgorithmus *Selectionsort* implementieren. Sei dazu die folgende Beschreibung gegeben:

(a) Falls die Liste leer oder einelementig ist, wird die Liste unverändert zurückgegeben.

(b) Suche im nicht-sortierten Teil der Liste das Minimum, entferne es von seinem aktuellen Platz und stelle es an den Anfang der betrachteten Liste.

(c) Beginne bei Schritt (a) mit dem Rest der Liste.

Im Folgenden wird nun eine Schritt-für-Schritt Vorgehensweise angegeben, wie sie sinnvoll sein kann, um einen neuen Algorithmus zu verstehen und implementieren zu können.

(a) Suchen Sie sich ein bis zwei Beispielliste und gehen Sie die Schritte von Selectionsort für diese Liste durch, bis Sie sortiert sind (oder sie 100%-ig sicher sind, dass sie die Vorgehensweise vollständig verstanden haben. ;-)).

(b) Erklären Sie, wie Selectionsort eine beliebige Liste sortiert.

(c) Finden Sie ein induktives Argument dafür, das Selectionsort tatsächlich Listen sortiert.

(d) Schreiben Sie eine Prozedur *selectMinimum*: *int list* → *int*, die das kleinste Element einer Liste bestimmt.

(e) Schreiben Sie mit Hilfe von *List.filter* eine Prozedur *delete*: $'\alpha \rightarrow \ '\alpha\ list \rightarrow \ '\alpha\ list$, die ein gegebenes Element aus einer Liste entfernt.

(f) Implementieren Sie nun die Prozedur *selectionSort*: *int list* → *int list*, die eine Liste sortiert, indem sie zunächst das kleinste Element der Liste sucht, es dann aus der Liste entfernt und es vor den rekursiv sortierten Rest der Liste hängt.

Lösung 5.9:

(d)
```
fun selectMin' compare a   nil   = a
  | selectMin' compare a (x::xs) =
        if compare(x,a) = LESS then selectMin' compare x xs else selectMin' compare a xs

fun selectMinimum compare   nil   = raise Empty
  | selectMinimum compare (x::xr) = selectMin' compare x xr
```

(e)
```
fun delete compare a xs = List.filter(fn x => compare(x,a) <> EQUAL) xs
(* Sortiert strickt *)
```

(f)
```
fun selectionsort compare nil = nil
  | selectionsort compare xs  = let
                                    val a = (selectMinimum compare xs)
                                in
                                    a :: selectionsort compare (delete compare a xs)
                                end
```

Aufgabe 5.10
Schätzen Sie: Welcher der Sortieralgorithmen, die Sie hier bzw. in der Vorlesung oder auf dem offiziellen Übungsblatt kennengelert haben, ist der schnellste?

Hinweis: Beziehen Sie folgende Ideen mit ein: Wie verhalten sich die Algorithmen auf schon sortierten Listen, wie auf absteigend sortierten? Was passiert, wenn die Liste schon fast sortiert ist, also z.B. nur ein Element am falschen Platz steht? Was passiert auf "vollständig" unsortieren Listen?

4 Darstellung von Mengen

Aufgabe 5.11
Im Folgenden wird eine andere Darstellungform von Mengen implementiert: Mengen sollen mithilfe einer Prozedur *set* vom Typ $int \to bool$ dargestellt werden. $set(n) = true$ (mit $n \in \mathbb{Z}$) soll genau dann gelten, wenn n in der Menge enthalten ist, die dargestellt werden soll.

(a) Geben Sie eine Prozedur $emptySet\colon int \to bool$ an, die die leere Menge beschreibt.

(b) Geben Sie eine Prozedur $mixedSet\colon int \to bool$ an, die die Menge mit den Zahlen 1, -716 und 42367 darstellt.

(c) Deklarieren Sie eine Prozedur $element\colon int \to (int \to bool) \to bool$, welche testet, ob eine gegebene Zahl $z \in \mathbb{Z}$ in einer gegeben Menge vorhanden ist.

(d) Welchen Unterschied sehen Sie zwischen der Prozedur *set* und der Prozedur *element* (bis auf die unterschiedlichen Typschemen)?

(e) Deklarieren Sie nun auch die anderen Mengenprozeduren, also $intersection$, $difference$, $union$ und $subset$. Überlegen Sie sich dazu zunächst ein passendens Typschema.

Hinweis: Sie brauchen keine Rekursion!

Lösung 5.11:

(a) `fun emptySet (x:int) = false`

(b) `fun mixedSet x = 1 = x orelse ~716 = x orelse 42367 = x`

(c) `fun element x (m: int -> bool) = m x`

(e) `(int -> bool) -> (int -> bool) -> int -> bool`

- `fun intersection m1 m2 = fn (x:int) => m1 x andalso m2 x`
- `fun difference m1 m2 = fn (x:int) => m1 x andalso not(m2 x)`
- `fun union m1 m2 = fn (x:int) => m1 x orelse m2 x`
- ```
  fun subset f g = let
  val SOME x = Int.minInt
  val SOME y = Int.maxInt
  fun help a = if a < y
  then ((f a) = (g a) orelse (g a)) andalso help (a+1)
  else (f y) = (g y) orelse (g y)
  in help x
  end
  ```
  Achtung: Dies ist keine schöne Lösung, da sie sehr lange braucht, um berechnet zu werden und es Warnungen bei der Deklaration der Prozedur gibt.

5 Knobelecke

Aufgabe 5.12 (*The Dark Dieter rises*)
Disclaimer: Die folgende Aufgabe wird *wirklich* knifflig. Lassen Sie sich von dieser Aufgabe auf keinen Fall entmutigen.

Rufen Sie sich noch einmal Aufgabe 5.16 vom aktuellen Übungsblatt in Erinnerung. Dabei sollten Sie eine Reihe von Prozeduren schreiben, die verschiedene Mengenoperationen für strikt sortierte Listen implementieren.
Ihr bester Freund, Dieter Schlau, fordert Sie heraus, diese Aufgaben erneut zu lösen. Natürlich gibt es dabei einige Haken und Ösen:

- Die Prozedur, die sie schreiben, darf nicht selbst rekursiv sein.
- Sie dürfen pro Prozedur, die Sie implementieren jeweils nur eine Deklaration benutzen und dabei keine let-Ausdrücke verwenden.
- Sie dürfen nur Hilfsprozeduren benutzen, die explizit erlaubt werden. Falls nicht anders vorgegeben, dürfen Sie jede Hilfsprozedur nur genau ein Mal benutzen, d.h. in ihrer Deklaration darf maximal ein Aufruf der Hilfsprozedur vorkommen.
- Sie dürfen eventuelle Lücken in der Formulierung dieser Einschränkungen nicht ausnutzen. :-)
  Dieter Schlau sieht alles!

Bereit? Auf in den Kampf! Dieter Schlau erwartet Sie.

(a) Deklarieren Sie $subset : int\,list \to int\,list \to bool$ mit $foldl$ und $member$.

(b) Auf zum Endgegner?
Mit welchen Hilfsprozeduren können Sie $member : int \to int\,list \to bool$ deklarieren?

(i) mit $iter$, nth, hd und tl

(ii) mit $first$, nth und $length$

Eine dieser beiden Varianten ist nicht lösbar. Welche? Begründen Sie, warum. Geben Sie für die andere Variante eine Lösung an.
  ```

(c) Auf zum *wahren* Endgegner: *Dark Dieter*.
Deklarieren Sie *union* mit zwei Vorkommen von $foldr$ und einem Vorkommen von *op*@.

Lösung 5.12:

(a) `fun subset xs ys = foldl (fn (x,s) => s andalso member x ys) true xs`

(b) Mit *iter* und *nth* ist keine Lösung möglich. Um *iter* zu benutzen, muss ein n angegeben werden, was der Größe der Liste entsprechen muss. Die Gößе der Liste lässt sich aber nur mit *nth* und ohne Rekursion nicht bestimmen.

```
fun member (x : int) xs =
(first 0 (fn n => n = length xs orelse nth (xs, n) = x)) < length xs
```

(c) Diese Musterlösung wurde von *Dark Dieter* gesperrt.

Sie müssen leider alleine knobeln. Wenn Sie *wirklich* nicht weiterwissen und die Antwort *unbedingt* wissen wollen, fragen Sie Ihren Tutor.