



Einführung in die Informatik II
Univ.-Prof. Dr. Andrey Rybalchenko, M.Sc. Ruslán Ledesma Garza

Dieses Blatt behandelt Kapitel 14.1 – 14.4 und 14.7 aus dem Buch zur Vorlesung. Lesen Sie diese Kapitel!

Aufgabe 14.1

Sei die Struktur `ISet` gegeben.

```
signature ISET =  
  sig  
    type set  
    val set : int list → set  
    val union : set → set → set  
    val subset : set → set → bool  
  end  
  
structure ISet :> ISET =  
  struct  
    type set = int list  
    fun set xs = xs  
    fun union xs ys = xs@ys  
    fun elem ys x = List.exists (fn y ⇒ y=x) ys  
    fun subset xs ys = List.all (elem ys) xs  
  end
```

- Schreiben Sie eine Prozedur `member : int → ISet.set → bool`, die testet, ob eine Zahl in einer Menge enthalten ist.
- Schreiben Sie eine Prozedur `empty : ISet.set → bool`, die testet, ob eine Menge leer ist.
- Schreiben Sie eine Prozedur `equal : ISet.set → ISet.set → bool`, die testet, ob zwei Mengen gleich sind.

Lösungsvorschlag 14.1:

```
fun elem x ys = ISet.subset (ISet.set [x]) ys  
  
fun empty xs = ISet.subset xs (ISet.set [])  
  
fun equal xs ys = ISet.subset xs ys andalso ISet.subset ys xs
```

Aufgabe 14.2

Warum ist die folgende Deklaration unzulässig?

```
structure S :> sig eqtype t end = struct  
  type t = int → int  
end
```

Lösungsvorschlag 14.2: Because the type `t` is not an equality type.

Aufgabe 14.3

Die Struktur `ISet` aus Abbildung 14.1 auf S. 281 implementiert die abstrakte Datenstruktur `ISet` korrekt aber ineffizient. Zum einen kann die eine Menge Liste Elemente mehrfach enthalten (z.B. bei `ISet.set [1,1]`). Zum anderen hat die Operation `subset` die quadratische Laufzeit (in der Summe der Länge der darstellenden Listen).

Schreiben Sie eine Struktur `ISSet`, die Mengen mit strikt sortierten Listen implementiert (§5.4 und 5.5). Achten Sie darauf, dass Sie die Operation `set` mit linear-logarithmischer und die Operationen `union` und `subset` mit linearer Laufzeit realisieren.

Lösungsvorschlag 14.3:

```

structure ISSet :> ISET =
  struct
    type set = int list

    fun merge (nil, ys ) = ys
      | merge (xs, nil ) = xs
      | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                              else y::merge(x::xr,yr)
    fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys)) (nil, nil) xs
    fun msort [] = []
      | msort [x] = [x]
      | msort xs =
        let val (ys,zs) = split xs
        in merge(msort ys, msort zs) end

    fun normalize NONE (x :: xs) = x :: normalize (SOME x) xs
      | normalize _ [] = []
      | normalize (SOME y) (x :: xs) = if y = x then normalize (SOME y) xs
        else x :: normalize (SOME x) xs

    fun set xs = normalize NONE (msort xs)
    fun union xs ys = normalize NONE (merge(xs, ys))
    fun subset [] _ = true
      | subset _ [] = false
      | subset (x :: xs) (y :: ys) =
        if x > y then subset (x :: xs) ys
        else if x < y then false
        else subset xs ys
  end
end

```

Aufgabe 14.5 (Umgebungen)

Unter einer Umgebung wollen wir wie in §6.4.2 eine Funktion verstehen, die durch Strings realisierte Bezeichner auf bestimmte Werte abbildet. Deklarieren Sie eine Struktur `Env`, die Umgebungen wie folgt implementiert:

```

type 'a env
exception Unbound
val env : (string * 'a) list -> 'a env
val lookup : 'a env -> string -> 'a
val update : 'a env -> string -> 'a -> 'a env

```

- `env` liefert zu einer Liste von Paaren die entsprechende Umgebung. Wenn die Liste zu einem String mehr als ein Paar enthält, soll nur eines der Paare verwendet werden.
- `lookup` liefert zu einer Umgebun `f` und einem String `s` den Wert `f s`. Falls `f` auf `s` nicht definiert ist, wirft `lookup` die Ausnahme `Unbound`.
- `update` liefert zu einer Umgebung `f`, einem String `s` und einem Wert `x` die Umgebung `f [s:=x]`.

Lösungsvorschlag 14.5:

```

signature ENV =
sig
  type 'a env
  exception Unbound
  val env : (string * 'a) list → 'a env
  val lookup : 'a env → string → 'a
  val update : 'a env → string → 'a → 'a env
end

structure Env :> ENV =
struct
  type 'a env = (string * 'a) list
  exception Unbound
  fun env u = u
  fun lookup u s =
    case List.find (fn (s', _) ⇒ s' = s) u of
      SOME (_, v) ⇒ v
    | NONE ⇒ raise Unbound
  fun update u s v =
    case List.find (fn (s', _) ⇒ s' = s) u of
      SOME _ ⇒ List.map (fn (s', v') ⇒ if s' = s then (s, v) else (s', v')) u
    | NONE ⇒ (s, v) :: u
end

```

Aufgabe 14.6 (Listen)

Deklarieren Sie eine Struktur `MyList`, die Listen mit der folgenden Signatur bereitstellt:

```

eqtype 'a list
val empty : 'a list
val cons : 'a → 'a list → 'a list
val null : 'a list → bool
val hd : 'a list → 'a
val tl : 'a list → 'a list

```

Achten Sie darauf, dass die Struktur alle Operationen mit konstanter Laufzeit implementiert. Machen Sie sich klar, dass alle anderen Listenoperationen mit den Operationen der Struktur programmiert werden können. Zeigen Sie das am Beispiel von `foldl`.

Lösungsvorschlag 14.6:

```

signature LIST =
sig
  eqtype 'a list
  val empty : 'a list
  val cons : 'a → 'a list → 'a list
  val null : 'a list → bool
  val hd : 'a list → 'a
  val tl : 'a list → 'a list
end

structure MyList =
struct
  type 'a list = 'a list
  val empty = []
  fun cons x xs = x :: xs
  fun null [] = true
    | null _ = false
  fun hd (x :: xs) = x
    | hd [] = raise Empty
  fun tl (x :: xs) = xs
    | tl [] = raise Empty
end

```

Aufgabe 14.7 * (Puzzle)

Für eine abstrakte Datenstruktur stellt sich oft die Frage, ob eine gewünschte Operation mit den Operationen der Struktur programmiert werden kann. Dazu wollen wir eine abstrakte Datenstruktur `Puzzle` betrachten, die endliche Mengen ganzer Zahlen gemäß der folgenden Signatur bereitstellt:

```
type set
val set : int list → set
val find : (int → bool) → set → int option
```

Die Operation `set` soll zu einer Liste die entsprechende Menge liefern, und die Operation `find` soll zu einer Eigenschaft und einer Menge ein Element der Menge liefern, das die Eigenschaft erfüllt. Die Operation `find` ruft den Eigenschafts-Test nur auf Elementen der Menge auf.

- Deklarieren Sie eine Struktur `Puzzle`, die die abstrakte Datenstruktur `Puzzle` realisiert.
- Schreiben Sie mithilfe der Operationen von `Puzzle` eine Prozedur `set → int list`, die die Elemente einer Menge als Liste liefert.

Lösungsvorschlag 14.7 *:

```
signature PUZZLE =
sig
  type set
  val set : int list → set
  val find : (int → bool) → set → int option
end

structure Puzzle :> PUZZLE =
struct
  type set = int list

  fun merge (nil, ys ) = ys
    | merge (xs, nil ) = xs
    | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                          else y::merge(x::xr,yr)
  fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys)) (nil, nil) xs
  fun msort [] = []
    | msort [x] = [x]
    | msort xs =
      let val (ys,zs) = split xs
      in merge(msort ys, msort zs) end

  fun normalize NONE (x :: xs) = x :: normalize (SOME x) xs
    | normalize _ [] = []
    | normalize (SOME y) (x :: xs) = if y = x then normalize (SOME y) xs
    else x :: normalize (SOME x) xs

  fun set xs = normalize NONE (msort xs)
  fun find p xs = List.find p xs
end

(* Version 1 *)

exception int_list of int list

fun puzzlex xs set =
  let
    val _ = Puzzle.find (fn x =>
      if List.exists (fn y => x = y) xs then false
      else raise int_list (x :: xs)) set
  in
```

```

    xs
  end handle int_list xs' ⇒ puzzlex xs' set

fun puzzle1 set = puzzlex [] set

(* Version 2 *)

fun puzzle2 set =
  let val xs = ref [] in
    Puzzle.find (fn x ⇒ (xs := x :: !xs; false)) set;
    !xs
  end

```

Aufgabe 14.8

Schreiben Sie eine Prozedur `vector2list : 'a vector → 'a list`, die zu einem Vektor die entsprechende Liste liefert.

Lösungsvorschlag 14.8:

```
fun vector2list v = Vector.foldr (op::) [] v
```

Aufgabe 14.9

Schreiben Sie eine Prozedur `cons : 'a → 'a vector → 'a vector`, die zu `x` und $[x_1, \dots, x_n]$ den Vektor $[x, x_1, \dots, x_n]$ liefert, Verwenden Sie dabei `Vector.concat`.

Lösungsvorschlag 14.9:

```
fun cons x xs = Vector.concat [(Vector.fromList [x]), xs]
```

Aufgabe 14.15 (Optional)

Schreiben Sie polymorphe Prozeduren `ssort`, `smerge` und `ssublist`, wie sie für die Deklaration des Funktors `Set` in Abbildung 14.6 erforderlich sind.

Lösungsvorschlag 14.15:

```

fun smerge compare xs ys =
  case (xs, ys) of
  (nil, ys) ⇒ ys
| (xs, nil) ⇒ xs
| (x::xr, y::yr) ⇒
  case compare (x, y) of
  EQUAL ⇒ x::smerge compare xr yr
| LESS ⇒ x::smerge compare xr (y::yr)
| GREATER ⇒ y::smerge compare (x::xr) yr
fun split xs = foldl (fn (x, (ys,zs)) ⇒ (zs, x::ys)) (nil, nil) xs
fun ssort compare xs =
  case xs of
  [] ⇒ []
| [x] ⇒ [x]
| xs ⇒
  let val (ys,zs) = split xs
  in smerge compare (ssort compare ys) (ssort compare zs) end

fun ssubset compare xs ys =
  case (xs, ys) of
  ([], _) ⇒ true
| (_, []) ⇒ false
| (x :: xs, y :: ys) ⇒

```

```
case compare (x, y) of
  GREATER ⇒ ssubset compare (x :: xs) ys
| LESS ⇒ false
| EQUAL ⇒ ssubset compare xs ys
```