Abgabe: 23.12.2008 (vor der Vorlesung)

Aufgabe 10.1 (H) Module und Funktoren

a) Es sei folgende Signatur gegeben:

```
module type Zahl = sig
  type z
  val zero : z
  val ( +. ) : z -> z -> z
  val ( *. ) : z -> z -> z
  val string_of_zahl : z -> string
end
```

i) Definieren Sie ein Modul Boolean, das von dieser Signatur ist. Der $+.$-Operator bzw. der $*.$-Operator dieses Moduls soll dem logischen Oder- bzw. dem logischen Und-Operator entsprechen. Das Element zero soll das neutrale Element der Addition sein, d.h. in diesem Fall `false`.

ii) Definieren Sie ein Modul MinPlusNat, das von dieser Signatur ist. Die Elemente sind alle natürlichen Zahlen inklusive der 0 erweitert um ∞. Der $+.$-Operator dieses Moduls soll das Minimum zweier Zahlen berechnen. Der $*.$-Operator soll der gewöhnlichen Addition entsprechen. Das Element zero soll das neutrale Element der Addition sein, d.h. in diesem Fall ∞.

b) Matrizen sind durch folgende Signatur definiert:

```
module type Matrix = sig
  type e
  type m
  val matrix_zero : m
  val ( **. ) : m -> m -> m
  val set_entry : m -> int * int -> e -> m
  val string_of_matrix : m -> string
end
```

Dabei ist e der Typ für die Einträge der Matrix und m der Typ für die Matrix selbst. Der Wert matrix_zero ist eine Matrix, die nur aus Null-Einträgen besteht. Der Operator $**.$ ist die Matrix-Multiplikation. Der Aufruf `set_entry m (i,j) e` liefert eine Matrix zurück, die der Matrix m entspricht, bis darauf, dass der Eintrag in der i-ten Zeile und der j-ten Spalte e ist.

i) Definieren Sie einen Funktor MakeMatrix, der eine Struktur der Signatur Zahl als Argument erhält und eine Struktur der Signatur Matrix zurückliefert. Die Matrix-Multiplikation $**.$ soll eine Verallgemeinerung der Matrixmultiplikation aus Aufgabe 7.1 sein, d.h., es sollen die Multiplikation $*.$ und die Addition +. der übergebenen Zahl-Struktur verwendet werden.

ii) Testen Sie Ihre Implementierung anhand folgender Zeilen:

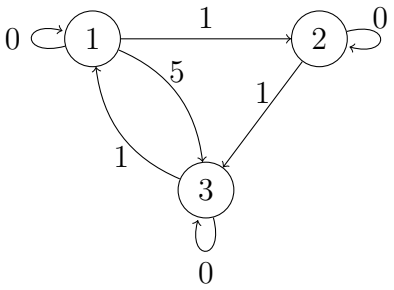
```
module BooleanMatrix = MakeMatrix(Boolean)
open BooleanMatrix
let a = set_entry matrix_zero (1,1) true
let a = set_entry a      (2,2) true
let a = set_entry a      (3,3) true
let a = set_entry a      (1,2) true
let a = set_entry a      (2,3) true
let a = set_entry a      (3,1) true

module MinPlusNatMatrix = MakeMatrix(MinPlusNat)
open MinPlusNat
open MinPlusNatMatrix
let b = set_entry matrix_zero (1,1) (Value 0)
let b = set_entry b      (2,2) (Value 0)
let b = set_entry b      (3,3) (Value 0)
let b = set_entry b      (1,2) (Value 1)
let b = set_entry b      (2,3) (Value 1)
let b = set_entry b      (3,1) (Value 1)
let b = set_entry b      (1,3) (Value 5)
```

c) Definieren Sie einen Funktor Fix, der eine Struktur M der Signatur Matrix als Argument erhält und eine Struktur zurückliefert, in der eine Funktion fix : M.m -> M.m definiert ist, die eine Matrix solange mit sich selbst multipliziert, bis keine Veränderung mehr auftritt. Die so berechnete Matrix A, für die also $A^2 = A$ gilt, soll zurückgeliefert werden.

d) Test Sie ihre Implementierung anhand der Matrizen a und b.

e) Was haben Sie mithilfe Ihrer Implementierung eigentlich berechnet, wenn Sie davon ausgehen, dass eine Matrix einen, unter Umständen kantenbewerteten, gerichteten Graphen, repräsentiert. Beispielsweise repräsentiert die oben definierte Matrix b den folgenden kantenbewerteten, gerichteten Graphen:



Lösungsvorschlag 10.1

```
(* Teil a *)
module type Zahl = sig
  type z
  val zero : z
  val ( +. ) : z -> z -> z
  val ( *. ) : z -> z -> z
  val string_of_zahl : z -> string
```

```
end

(* Teil a i *)
module Boolean = struct
  type z = bool
  let zero = false
  let ( +. ) = ( || )
  let ( *. ) = ( && )
  let string_of_zahl = function true -> "1" | false -> "0"
end

(* Teil a ii *)
module MinPlusNat = struct
  type z = Value of int | Inf
  let zero = Inf
  let ( +. ) x y =
    match (x,y) with
      Value x, Value y             -> Value(min x y)
    | Value x, Inf | Inf, Value x -> Value x
    | Inf,Inf                       -> Inf
  let ( *. ) x y =
    match (x,y) with
      Value x, Value y -> Value(x + y)
    | _                -> Inf
  let string_of_zahl = function Value x -> string_of_int x | _ -> "inf"
end

(* Teil b *)
module type Matrix = sig
  type e
  type m
  val matrix_zero : m
  val ( **. ) : m -> m -> m
  val set_entry : m -> int * int -> e -> m
  val string_of_matrix : m -> string
end

exception NoImpl

(* Teil b i und f
   fuer b i reicht als erste Zeile
module MakeMatrix(Z:Zahl) = struct
*)
module MakeMatrix(Z:Zahl) : Matrix with type e = Z.z = struct
  open Z
  type e = Z.z
  type m = (int * (int * z) list) list

  (* Vektor-Vektor-Multiplikation *)
  let rec v_mult p x y =
    match (x,y) with
      ((px,vx)::x',(py,vy)::y') ->
        if px = py then
```

```
        v_mult (p +. vx *. vy) x' y'
      else if px < py then
        v_mult p x' y
      else
        v_mult p x y'
  | _ -> p

let v_mult x y = v_mult zero x y

(* Matrix * Vektor *)
let m_v_mult a v =
  let l = List.map (fun (j,aj) -> (j, v_mult aj v)) a in
    (* Multipliziert Zeilenvektor aj mit dem Vektor v *)
    List.filter (fun (j,aj) -> (aj <> zero)) l (* Nullen entfernen *)

let rec v_to_list i = function
  (j,x)::v -> (i,j,x)::v_to_list i v
| _ -> []

let rec m_to_list = function
  (i,v)::a -> v_to_list i v @ m_to_list a
| _ -> []

let m_from_list l =
  let l = List.sort compare l in
  List.fold_right
    (
    fun (i,j,x) ->
      function ((i',v)::vs) ->
        if i = i' then
          ((i,(j,x)::v)::vs)
        else
          (i,[(j,x)])::((i',v)::vs)
      | _ -> [(i,[(j,x)])]
    )
    l
    []

let transpose a =
  let l = m_to_list a in
  let l = List.map (fun (i,j,x) -> (j,i,x)) l in
  m_from_list l

let ( **. ) a b =
  let b = transpose b in
  let handle_row_a (i,ai) = (i, m_v_mult b ai) in
  let r = List.map handle_row_a a in
  List.filter (fun (_,ai) -> ai <> []) r

let rec vector_set_entry v i z =
  match v with
    []                          -> [(i,z)]
  | (i',z')::v' when i < i' -> (i,z)::v
```

```
    | (i',z')::v' when i = i' -> (i,z)::v'
    | (i',z')::v'              -> (i',z')::vector_set_entry v' i z

  let rec set_entry m (i,j) z =
    match m with
      []                       -> [(i,[(j,z)])]
    | (i',v)::m' when i < i' -> (i,[(j,z)])::m
    | (i',v)::m' when i = i' -> (i,vector_set_entry v j z)::m'
    | (i',v)::m'             -> (i',v)::set_entry m' (i,j) z
  let matrix_zero = []
  let string_of_vector v =
    String.concat
      ";"
      (List.map (fun (i,z) -> "(" ^ string_of_int i ^ ","
                              ^ string_of_zahl z ^ ")") v)
  let string_of_matrix m =
    String.concat
      "\n"
      (List.map (fun (i,v) -> string_of_int i ^ "␣->␣"
                              ^ string_of_vector v) m)
end

(* Teil c *)
module Fix(M : Matrix) = struct
  open M
  let rec fix m =
    let m' = m **. m in
    if m' = m then m else fix m'
end

(* Teil b ii *)
module BooleanMatrix = MakeMatrix(Boolean)
open BooleanMatrix
let a = set_entry matrix_zero (1,1) true
let a = set_entry a    (2,2) true
let a = set_entry a    (3,3) true
let a = set_entry a    (1,2) true
let a = set_entry a    (2,3) true
let a = set_entry a    (3,1) true

module MinPlusNatMatrix = MakeMatrix(MinPlusNat)
open MinPlusNat
open MinPlusNatMatrix
let b = set_entry matrix_zero (1,1) (Value 0)
let b = set_entry b    (2,2) (Value 0)
let b = set_entry b    (3,3) (Value 0)
let b = set_entry b    (1,2) (Value 1)
let b = set_entry b    (2,3) (Value 1)
let b = set_entry b    (3,1) (Value 1)
let b = set_entry b    (1,3) (Value 5)

(* Teil d *)
module BF = Fix(BooleanMatrix)
```

```
let fix_a = BF.fix a
let str_fix_a = BooleanMatrix.string_of_matrix fix_a
let _ = print_string str_fix_a

module NF = Fix(MinPlusNatMatrix)
let fix_b = NF.fix b
let str_fix_b = MinPlusNatMatrix.string_of_matrix fix_b
let _ = print_string str_fix_b
```

e) a: Versteht man die Matrix a als Adjazenzmatrix eines gerichteten Graphen, ohne Kantengewichte, wurde in Teil d) die Erreichbarkeit zwischen zwei Knoten berechnet. Damit dies funktioniert, muss beim Aufbau der Matrix darauf geachtet werden, dass jeder Knoten mit sich selbst verbunden ist.

b: Versteht man die Matrix b als Adjazenzmatrix eines gerichteten, kantenbewerteten Graphen wurden in Teil d) die Längen der kürzesten Wege zwischen zwei Knoten berechnet. Damit dies funktioniert, muss die Matrix mit den Kantenwerten 0 für die Einträge (i, i) für alle i initialisiert werden.

Aufgabe 10.2 (P) In großen Schritten zum Ziel

Gegeben seien folgende MiniOCaml-Definitionen:

```
let f = (fun x -> (fun y -> 2 * x + y))

let g = f 7

let rec fact =
  fun n ->
    match n with
        0 -> 1
      | x -> x * (fact (x-1))
```

Konstruieren Sie die Beweise für folgende Aussagen:

a) f 3 4 $\Rightarrow$ 10

b) g 2 $\Rightarrow$ 16

c) fact 2 $\Rightarrow$ 2

Lösungsvorschlag 10.2

a)

$$\dfrac{\dfrac{\dfrac{f = fun\ x \rightarrow (fun\ y \rightarrow 2 * x + y)}{f \Rightarrow fun\ x \rightarrow (fun\ y \rightarrow 2 * x + y)}\ (GD)}{f\ 3 \Rightarrow fun\ y \rightarrow 2 * 3 + y}\ (App) \quad \dfrac{2 * 3 \Rightarrow 6 \quad 6 + 4 \Rightarrow 10}{2 * 3 + 4 \Rightarrow 10}\ (Op)}{f\ 3\ 4 \Rightarrow 10}\ (App)$$

b)

$$\dfrac{g = f\ 7 \quad \dfrac{\dfrac{\dfrac{f = fun\ x \rightarrow (fun\ y \rightarrow 2 * x + y)}{f \Rightarrow fun\ x \rightarrow (fun\ y \rightarrow 2 * x + y)}\ (GD)}{f\ 7 \Rightarrow fun\ y \rightarrow 2 * 7 + y}\ (App)}{g \Rightarrow fun\ y \rightarrow 2 * 7 + y}\ (GD) \quad \dfrac{2 * 7 \Rightarrow 14 \quad 14 + 2 \Rightarrow 16}{2 * 7 + 2 \Rightarrow 16}\ (Op)}{g\ 2 \Rightarrow 16}\ (App)$$

c) Zur Vereinfachung nehmen wir folgende Setzungen vor:

$$e_{match} = match\ n\ with\ 0 \rightarrow 1\ \mid\ x \rightarrow x * (fact\ (x - 1))$$

$$\pi := \dfrac{fact = fun\ n \rightarrow e_{match}}{fact \Rightarrow fun\ n \rightarrow e_{match}}\ (GD)$$

$$\dfrac{\pi \quad \dfrac{2 \Rightarrow 2 \equiv x[2/x] \qquad \dfrac{2 - 1 \Rightarrow 1 \qquad \dfrac{\pi \quad \dfrac{1 \Rightarrow 1 \equiv x[1/x] \qquad \dfrac{1 - 1 \Rightarrow 0 \qquad \dfrac{\pi \quad \dfrac{0 \Rightarrow 0 \equiv 0}{e_{match}[0/n] \Rightarrow 1}\,(PM)}{\textbf{fact 0} \Rightarrow \textbf{1}}\,(App)}{fact\ (1-1) \Rightarrow 1}\,(App) \qquad 1 * 1 \Rightarrow 1}{1 * (fact\ (1-1)) \Rightarrow 1}\,(Op)}{e_{match}[1/n] \Rightarrow 1}\,(PM)}{\textbf{fact 1} \Rightarrow \textbf{1}}\,(App)}{fact\ (2-1) \Rightarrow 1} \qquad 2 * 1 \Rightarrow 2}{2 * (fact\ (2-1)) \Rightarrow 2}\,(Op)}{e_{match}[2/n] \Rightarrow 2}\,(PM)}{\textbf{fact 2} \Rightarrow \textbf{2}}\,(App)$$

Aufgabe 10.3 (P) Abgeleitete Regeln

Zeigen Sie die Gültigkeit der folgenden abgeleiteten Regeln:

a)

$$\frac{\mathtt{e} = []}{(\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2) = \mathtt{e}_1}$$

b)

$$\frac{\mathtt{e}\ \text{terminiert} \qquad \mathtt{e} = \mathtt{e}' :: \mathtt{e}''}{(\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2) = \mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}]}$$

Lösungsvorschlag 10.3

Es sei angenommen, dass $\mathtt{e} = []$ gilt.

a) **Fall 1:** Die Auswertung des Ausdrucks $\mathtt{e}_1$ terminiert. Dann existiert ein Wert $\mathtt{v}_1$ so dass $\mathtt{e}_1 \Rightarrow \mathtt{v}_1$ gilt. Da $\mathtt{e} = []$ gilt, gilt $\mathtt{e} \Rightarrow []$. Es folgt

$$\frac{\mathtt{e} \Rightarrow [] \qquad \mathtt{e}_1 \Rightarrow \mathtt{v}_1}{(\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2) \Rightarrow \mathtt{v}_1}\ (PM)$$

Daraus folgt $\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2 = \mathtt{e}_1$.

Fall 2: Die Auswertung des Ausdrucks $\mathtt{e}_1$ terminiert nicht. Zur Herleitung eines Widerspruchs sei angenommen, dass die Auswertung des Ausdrucks $\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2$ terminiert. Es existiert also ein Beweis der Form

$$\frac{\mathtt{e} \Rightarrow [] \qquad \mathtt{e}_1 \Rightarrow \mathtt{v}_1}{(\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2) \Rightarrow \mathtt{v}_1}\ (PM)$$

Daraus folgt, dass die Auswertung des Ausdruck e_1 terminiert — Widerspruch.

b) Es sei angenommen, dass

$$\mathtt{e}\ \text{terminiert} \quad \text{und} \quad \mathtt{e} = \mathtt{e}' :: \mathtt{e}''$$

gelten. Daraus folgt, dass ein Wert $\mathtt{v}$ exisitiert, so dass $\mathtt{e} \Rightarrow \mathtt{v}$ und $\mathtt{e}' :: \mathtt{e}'' \Rightarrow \mathtt{v}$ gelten. Notwendigerweise muss $\mathtt{v} = \mathtt{v}' :: \mathtt{v}''$ mit

$$\mathtt{e}' \Rightarrow \mathtt{v}' \quad \text{und} \quad \mathtt{e}'' \Rightarrow \mathtt{v}''$$

gelten. Daraus folgt, dass

$$\mathtt{e}' = \mathtt{v}' \quad \text{und} \quad \mathtt{e}'' = \mathtt{v}''$$

gelten. Mit dem Substitutionslemma folgt

$$\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}] = \mathtt{e}_2[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}]. \tag{1}$$

Fall 1: Die Auswertung des Ausdrucks $\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}]$ terminiert. Es existiert also ein Wert $\mathtt{v}_2$ mit $\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}] \Rightarrow \mathtt{v}_2$ und wegen (1) gilt $\mathtt{e}_2[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}] \Rightarrow \mathtt{v}_2$. Es folgt

$$\frac{\mathtt{e} \Rightarrow \mathtt{v}' :: \mathtt{v}'' \equiv \mathtt{x} :: \mathtt{xs}[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}] \quad \mathtt{e}_2[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}] \Rightarrow \mathtt{v}_2}{\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2 \Rightarrow \mathtt{v}_2}\ (PM)$$

Beide Seiten werten sich also zu $\mathtt{v}_2$ aus. Daher gilt

$$\mathtt{match\ e\ with}\ [] \to \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \to \mathtt{e}_2 = \mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}].$$

Fall 2: Die Auswertung des Ausdrucks $\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}]$ terminiert nicht. Zur Herleitung eines Widerspruchs sei angenommen, dass die Auswertung des Ausdrucks

$$\mathtt{match\ e\ with}\ [] \rightarrow \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \rightarrow \mathtt{e}_2$$

terminiert. Es existiert also ein Wert $\mathtt{v}_2$ mit

$$\mathtt{match\ e\ with}\ [] \rightarrow \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \rightarrow \mathtt{e}_2 \Rightarrow \mathtt{v}_2.$$

Es existiert also ein Beweis der Form

$$\frac{\mathtt{e} \Rightarrow \mathtt{v}' :: \mathtt{v}'' \equiv \mathtt{x} :: \mathtt{xs}[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}] \quad \mathtt{e}_2[\mathtt{v}'/\mathtt{x}, \mathtt{v}''/\mathtt{xs}] \Rightarrow \mathtt{v}_2}{\mathtt{match\ e\ with}\ [] \rightarrow \mathtt{e}_1 \mid \mathtt{x} :: \mathtt{xs} \rightarrow \mathtt{e}_2 \Rightarrow \mathtt{v}_2}\ (PM)$$

Mit (1) folgt, dass

$$\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}] \Rightarrow \mathtt{v}_2$$

gilt. Dies ist ein Widerspruch zu der Annahme, dass die Auswertung des Ausdrucks $\mathtt{e}_2[\mathtt{e}'/\mathtt{x}, \mathtt{e}''/\mathtt{xs}]$ nicht terminiert. □

Big-Step Operationelle Semantik

Axiome: $v \Rightarrow v$ für jeden Wert v

Tupel:
$$\frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)} \; (T)$$

Listen:
$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{::} e_2 \Rightarrow v_1 \texttt{::} v_2} \; (L)$$

Globale Definitionen:
$$\frac{f = e \quad e \Rightarrow v}{f \Rightarrow v} \; (GD)$$

Lokale Definitionen:
$$\frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\texttt{let}\ x = e_1\ \texttt{in}\ e_0 \Rightarrow v_0} \; (LD)$$

Funktionsaufrufe:
$$\frac{e_1 \Rightarrow \texttt{fun}\ x \texttt{->} e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1\ e_2 \Rightarrow v_0} \; (App)$$

Pattern Matching:
$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \ldots, v_k/x_k] \quad e_i[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1 \texttt{->} e_1 \ \texttt{|}\ \ldots \ \texttt{|}\ p_m \texttt{->} e_m \Rightarrow v} \; (PM)$$

— sofern v' auf keines der Muster $p_1, \ldots, p_{i-1}$ passt

Eingebaute Operatoren:
$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 \,\mathsf{op}\, v_2 \Rightarrow v}{e_1 \,\mathsf{op}\, e_2 \Rightarrow v} \; (Op)$$

— Unäre Operatoren werden analog behandelt.

Substitutionslemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$