TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen WS 2008/09
Einführung in die Informatik 2 **Übungsblatt 10**
Prof. Dr. Helmut Seidl, T. M. Gawlitza, S. Pott, M. Schwarz 16.12.2008

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:

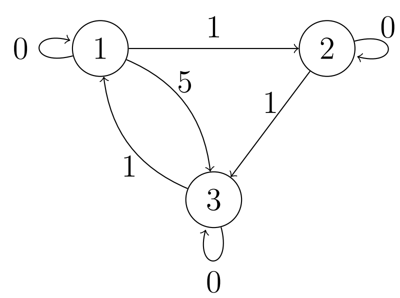
```
module BooleanMatrix = MakeMatrix(Boolean)
open BooleanMatrix
let a = set_entry matrix_zero (1,1) true
let a = set_entry a    (2,2) true
let a = set_entry a    (3,3) true
let a = set_entry a    (1,2) true
let a = set_entry a    (2,3) true
let a = set_entry a    (3,1) true

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:



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) $\texttt{f 3 4} \Rightarrow 10$

b) $\texttt{g 2} \Rightarrow 16$

c) $\texttt{fact 2} \Rightarrow 2$

Aufgabe 10.3 (P) Abgeleitete Regeln

Zeigen Sie die Gültigkeit der folgenden abgeleiteten Regeln:

a)

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

b)

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

Big-Step Operationelle Semantik

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

Tupel:

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

Listen:

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

Globale Definitionen:

$$\frac{f = e \qquad e \;\Rightarrow\; v}{f \;\Rightarrow\; v} \;(GD)$$

Lokale Definitionen:

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

Funktionsaufrufe:

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

Pattern Matching:

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

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

Eingebaute Operatoren:

$$\frac{e_1 \;\Rightarrow\; v_1 \qquad e_2 \;\Rightarrow\; v_2 \qquad v_1 \,\mathsf{op}\, v_2 \Rightarrow v}{e_1 \,\mathsf{op}\, e_2 \;\Rightarrow\; v} \;(Op)$$

— Unäre Operatoren werden analog behandelt.