Informatik II für Verkehrsingenieure
Haskell at Work

Janis Voigtländer

Technische Universität Dresden

Sommersemester 2007

Wiederholung — Towers of Hanoi

Regeln:
- drei Plätze: A, B und C
- zu Beginn n Scheiben unterschiedlicher Größe auf Platz A
- niemals eine größere auf einer kleineren Scheibe

Ziel: alle Scheiben auf Platz B

Strategie:

$$\begin{aligned} towers(n+1, i, j, k) &= towers(n, i, k, j) \;\; move(i, j) \;\; towers(n, k, j, i) \\ towers(0, i, j, k) &= \varepsilon \end{aligned}$$

In Haskell:

```
towers(n+1,i,j,k) = towers(n,i,k,j) ++ [(i,j)] ++ towers(n,k,j,i)
towers(0,i,j,k)   = []
```

Towers of Hanoi — Gesamtprogramm (I)

```
module Main where

towers(n+1,i,j,k) = towers(n,i,k,j) ++ [(i,j)]
                    ++ towers(n,k,j,i)
towers(0,i,j,k)   = []

data Place = A | B | C  deriving (Show,Read)

step (A,B) (a:as,bs,cs) = (as,a:bs,cs)
step (A,C) (a:as,bs,cs) = (as,bs,a:cs)
step (B,A) (as,b:bs,cs) = (b:as,bs,cs)
step (B,C) (as,b:bs,cs) = (as,bs,b:cs)
step (C,A) (as,bs,c:cs) = (c:as,bs,cs)
step (C,B) (as,bs,c:cs) = (as,c:bs,cs)
```


Towers of Hanoi — Gesamtprogramm (II)

```
run (move:rest) conf = conf:run rest (step move conf)
run []          conf = [conf]

disk 0 n = replicate (2*n-1) ' '
disk i n = replicate (n-i) ' ' ++ replicate (2*i-1) '*'
           ++ replicate (n-i) ' '

output (a:as,b:bs,c:cs) n = do putStr (disk a n)
                               putStr (disk b n)
                               putStrLn (disk c n)
                               output (as,bs,cs) n
output ([],[],[])       n = return 0

output' (as,bs,cs) n = output
                        (replicate (n-length as) 0 ++ as,
                         replicate (n-length bs) 0 ++ bs,
                         replicate (n-length cs) 0 ++ cs) n
```

Towers of Hanoi — Gesamtprogramm (III)

```
animate (conf:rest) n = do output' conf n
                           putStrLn (replicate (6*n-3) '-')
                           getLine
                           animate rest n
animate []          n = return 0

main = do n <- readLn
          animate (run (towers (n,A,B,C)) ([1..n],[],[])) n
```


Towers of Hanoi — Test

```
> main
2
 *
***
---------


***    *
---------


   *** *
---------

    *
   ***
---------
```

Wiederholung — AM_0

$\mathsf{AM}_0 = \mathsf{BZ} \times \mathsf{DK} \times \mathsf{HS} \times \underline{\mathsf{Inp}} \times \underline{\mathsf{Out}}$ mit:

BZ	$= \mathbb{N}$	Befehlszähler
DK	$= \mathbb{Z}^*$	Datenkeller
HS	$= \{h \mid h : \mathbb{N} \dashrightarrow \mathbb{Z}\}$	Hauptspeicher
$\underline{\mathsf{Inp}}$	$= \mathbb{Z}^*$	Eingabeband
$\underline{\mathsf{Out}}$	$= \mathbb{Z}^*$	Ausgabeband

- READ n: Lesen von Eingabeband in Hauptspeicher
- WRITE n: Ausgabe aus Hauptspeicher auf Ausgabeband
- LOAD n: Ablage aus Hauptspeicher auf Datenkeller
- STORE n: Entnahme aus Datenkeller in Hauptspeicher
- LIT z: Ablage einer Konstante auf Datenkeller
- ADD, MUL, SUB, DIV, MOD, LT, EQ, NE, GT, LE, GE: Berechnungen und Vergleiche (auf Datenkeller)
- JMP n: Sprung
- JMC n: Sprung abhängig von Datenkeller

Wiederholung — Befehlssemantik (I)

$\mathcal{C}[\![.]\!]: \Gamma \longrightarrow (\mathsf{AM}_0 \dashrightarrow \mathsf{AM}_0)$

$\mathcal{C}[\![\mathsf{READ}\ n]\!](m, d, h, inp, out) =$
wenn $inp = first(inp).rest(inp)$ mit $first(inp) \in \mathbb{Z}, rest(inp) \in \mathbb{Z}^*$,
dann $(m+1, d, h[n/first(inp)], rest(inp), out)$

$\mathcal{C}[\![\mathsf{WRITE}\ n]\!](m, d, h, inp, out) =$
wenn $h(n) \in \mathbb{Z}$, dann $(m+1, d, h, inp, out.h(n))$

$\mathcal{C}[\![\mathsf{LOAD}\ n]\!](m, d, h, inp, out) =$
wenn $h(n) \in \mathbb{Z}$, dann $(m+1, h(n) : d, h, inp, out)$

$\mathcal{C}[\![\mathsf{STORE}\ n]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d'$, dann $(m+1, d', h[n/d.1], inp, out)$

$\mathcal{C}[\![\mathsf{LIT}\ z]\!](m, d, h, inp, out) = (m+1, z : d, h, inp, out)$

Wiederholung — Befehlssemantik (II)

$\mathcal{C}[\![\text{ADD}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, (d.2 + d.1) : d', h, inp, out)$

für MUL, SUB, DIV und MOD analog

$\mathcal{C}[\![\text{LT}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, b : d', h, inp, out)$,
wobei $b = 1$, falls $d.2 < d.1$, sonst $b = 0$

für EQ, NE, GT, LE und GE analog

$\mathcal{C}[\![\text{JMP } e]\!](m, d, h, inp, out) = (e, d, h, inp, out)$

$\mathcal{C}[\![\text{JMC } e]\!](m, d, h, inp, out) =$
wenn $d = 0 : d'$, dann (e, d', h, inp, out);
wenn $d = 1 : d'$, dann $(m + 1, d', h, inp, out)$

Beispiel

1: LIT 0;	5: NE;	9: JMP 2;
2: READ 1;	6: JMC 10;	10: STORE 1;
3: LOAD 1;	7: LOAD 1;	11: WRITE 1;
4: LIT 0;	8: ADD;	

(1 , ε , [] , 5.2.0 , ε)

Beispiel

1: LIT 0;	5: NE;	9: JMP 2;
2: READ 1;	6: JMC 10;	10: STORE 1;
3: LOAD 1;	7: LOAD 1;	11: WRITE 1;
4: LIT 0;	8: ADD;	

(2 , 7 , [1/2] , 0 , ε)
(3 , 7 , [1/0] , ε , ε)
(4 , 0:7 , [1/0] , ε , ε)
(5 , 0:0:7 , [1/0] , ε , ε)
(6 , 0:7 , [1/0] , ε , ε)
(10 , 7 , [1/0] , ε , ε)
(11 , ε , [1/7] , ε , ε)
(12 , ε , [1/7] , ε , 7)

Implementierung der AM_0 in Haskell — Modellierung

$AM_0 = BZ \times DK \times HS \times \underline{Inp} \times \underline{Out}$ mit:

BZ	$= \mathbb{N}$	Befehlszähler
DK	$= \mathbb{Z}^*$	Datenkeller
HS	$= \{h \mid h : \mathbb{N} \dashrightarrow \mathbb{Z}\}$	Hauptspeicher
$\underline{Inp}$	$= \mathbb{Z}^*$	Eingabeband
$\underline{Out}$	$= \mathbb{Z}^*$	Ausgabeband

$\rightsquigarrow$ `(Int,[Int],[(Int,Int)],[Int],[Int])`

zum Beispiel:

(5 , 0:5:0 , [1/5] , 2.0 , ε) $\rightsquigarrow$ `(5,[0,5,0],[(1,5)],[2,0],[])`

Implementierung der AM_0 in Haskell — Befehle

```
data Command = READ Int | WRITE Int | LOAD Int
             | STORE Int | LIT Int | ADD | MUL
             | SUB | DIV | MOD | LT | EQ | NE
             | GT | LE | GE | JMP Int | JMC Int

program :: [Command]
program = [LIT 0, READ 1, LOAD 1, LIT 0, NE, JMC 10,
           LOAD 1, ADD, JMP 2, STORE 1, WRITE 1]
```


Implementierung der AM_0 in Haskell — Befehlssemantik

```
step :: Command -> (Int,[Int],[(Int,Int)],[Int],[Int])
        -> (Int,[Int],[(Int,Int)],[Int],[Int])
```

$\mathcal{C}[\![\text{READ } n]\!](m, d, h, inp, out) =$

wenn $inp = \mathit{first}(inp).\mathit{rest}(inp)$ mit $\mathit{first}(inp) \in \mathbb{Z}, \mathit{rest}(inp) \in \mathbb{Z}^*$,
dann $(m + 1, d, h[n/\mathit{first}(inp)], \mathit{rest}(inp), out)$

$\rightsquigarrow$

```
step (READ n) (m,d,h,first:rest,out) =
   (m+1,d,update h n first,rest,out)
```

$\mathcal{C}[\![\text{WRITE } n]\!](m, d, h, inp, out) =$

wenn $h(n) \in \mathbb{Z}$, dann $(m + 1, d, h, inp, out.h(n))$

$\rightsquigarrow$

```
step (WRITE n) (m,d,h,inp,out) =
   (m+1,d,h,inp,out ++ [get h n])
```

Implementierung der AM_0 in Haskell — Befehlssemantik

$\mathcal{C}[\![\text{LOAD } n]\!](m, d, h, inp, out) =$
wenn $h(n) \in \mathbb{Z}$, dann $(m + 1, h(n) : d, h, inp, out)$

$\rightsquigarrow$

```
step (LOAD n) (m,d,h,inp,out) =
   (m+1,(get h n):d,h,inp,out)
```

$\mathcal{C}[\![\text{STORE } n]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d'$, dann $(m + 1, d', h[n/d.1], inp, out)$

$\rightsquigarrow$

```
step (STORE n) (m,d1:d',h,inp,out) =
   (m+1,d',update h n d1,inp,out)
```

$\mathcal{C}[\![\text{LIT } z]\!](m, d, h, inp, out) = (m + 1, z : d, h, inp, out)$

$\rightsquigarrow$

```
step (LIT z) (m,d,h,inp,out) = (m+1,z:d,h,inp,out)
```


Implementierung der AM_0 in Haskell — Befehlssemantik

$\mathcal{C}[\![\text{ADD}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, (d.2 + d.1) : d', h, inp, out)$

$\rightsquigarrow$

```
step ADD (m,d1:d2:d',h,inp,out) =
   (m+1,(d2+d1):d',h,inp,out)
```

für MUL, SUB, DIV und MOD analog:

```
step MUL (m,d1:d2:d',h,inp,out) =
   (m+1,(d2*d1):d',h,inp,out)
step SUB (m,d1:d2:d',h,inp,out) =
   (m+1,(d2-d1):d',h,inp,out)
step DIV (m,d1:d2:d',h,inp,out) =
   (m+1,(div d2 d1):d',h,inp,out)
step MOD (m,d1:d2:d',h,inp,out) =
   (m+1,(mod d2 d1):d',h,inp,out)
```

Implementierung der AM_0 in Haskell — Befehlssemantik

$\mathcal{C}[\![\text{LT}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, b : d', h, inp, out)$,
wobei $b = 1$, falls $d.2 < d.1$, sonst $b = 0$

$\rightsquigarrow$

```
step LT (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2<d1 then 1 else 0):d',h,inp,out)

step EQ (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2==d1 then 1 else 0):d',h,inp,out)
step NE (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2/=d1 then 1 else 0):d',h,inp,out)
step GT (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2>d1 then 1 else 0):d',h,inp,out)
step LE (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2<=d1 then 1 else 0):d',h,inp,out)
step GE (m,d1:d2:d',h,inp,out) =
   (m+1,(if d2>=d1 then 1 else 0):d',h,inp,out)
```


Implementierung der AM_0 in Haskell — Befehlssemantik

$\mathcal{C}[\![\text{JMP } e]\!](m, d, h, inp, out) = (e, d, h, inp, out)$

$\rightsquigarrow$

```
step (JMP e) (m,d,h,inp,out) = (e,d,h,inp,out)
```

$\mathcal{C}[\![\text{JMC } e]\!](m, d, h, inp, out) =$
wenn $d = 0 : d'$, dann (e, d', h, inp, out);
wenn $d = 1 : d'$, dann $(m + 1, d', h, inp, out)$

$\rightsquigarrow$

```
step (JMC e) (m,0:d',h,inp,out) = (e,d',h,inp,out)
step (JMC e) (m,1:d',h,inp,out) = (m+1,d',h,inp,out)
```

Implementierung der AM_0 in Haskell — Hilfsfunktionen

```
get :: [(Int,Int)] -> Int -> Int
get ((a,b):h) n = if a==n then b else get h n

> get [(1,3),(4,2),(5,7)] 4
2


update :: [(Int,Int)] -> Int -> Int -> [(Int,Int)]
update ((a,b):h) n c | a==n = (a,c):h
                     | a<n  = (a,b):(update h n c)
                     | a>n  = (n,c):(a,b):h
update []        n c = [(n,c)]

> update [(1,3),(4,2),(5,7)] 3 1
[(1,3),(3,1),(4,2),(5,7)]
```


Implementierung der AM_0 in Haskell — Programmsemantik

```
run :: (Int,[Int],[(Int,Int)],[Int],[Int])
       -> [(Int,[Int],[(Int,Int)],[Int],[Int])]
run conf = if valid conf then conf:(run (next conf))
                         else [conf]

valid :: (Int,[Int],[(Int,Int)],[Int],[Int]) -> Bool
valid (m,d,h,inp,out) = 1<=m && m<=(length program)

next :: (Int,[Int],[(Int,Int)],[Int],[Int])
        -> (Int,[Int],[(Int,Int)],[Int],[Int])
next (m,d,h,inp,out) = step (program !! (m-1))
                            (m,d,h,inp,out)

initial :: (Int,[Int],[(Int,Int)],[Int],[Int])
initial = (1,[],[],[5,2,0],[])
```

Implementierung der AM_0 in Haskell — Gesamtprogramm

```
module Main where

import Prelude hiding (Ordering(..))

...

program :: [Command]
program = [LIT 0, READ 1, LOAD 1, LIT 0, NE, JMC 10,
           LOAD 1, ADD, JMP 2, STORE 1, WRITE 1]

...

initial :: (Int,[Int],[(Int,Int)],[Int],[Int])
initial = (1,[],[],[5,2,0],[])

main = mapM print (run initial)
```


Implementierung der AM_0 in Haskell — Test

```
> main
(1,[],[],[5,2,0],[])
(2,[0],[],[5,2,0],[])
(3,[0],[(1,5)],[2,0],[])
(4,[5,0],[(1,5)],[2,0],[])
(5,[0,5,0],[(1,5)],[2,0],[])
(6,[1,0],[(1,5)],[2,0],[])
(7,[0],[(1,5)],[2,0],[])
(8,[5,0],[(1,5)],[2,0],[])
(9,[5],[(1,5)],[2,0],[])
(2,[5],[(1,5)],[2,0],[])
(3,[5],[(1,2)],[0],[])
(4,[2,5],[(1,2)],[0],[])
(5,[0,2,5],[(1,2)],[0],[])
(6,[1,5],[(1,2)],[0],[])
(7,[5],[(1,2)],[0],[])
...
```

Zum Vergleich:

1: LIT 0;	5: NE;	9: JMP 2;
2: READ 1;	6: JMC 10;	10: STORE 1;
3: LOAD 1;	7: LOAD 1;	11: WRITE 1;
4: LIT 0;	8: ADD;	

(1 , ε , [] , 5.2.0 , ε)
(2 , 0 , [] , 5.2.0 , ε)
(3 , 0 , [1/5] , 2.0 , ε)
(4 , 5:0 , [1/5] , 2.0 , ε)
(5 , 0:5:0 , [1/5] , 2.0 , ε)
(6 , 1:0 , [1/5] , 2.0 , ε)
(7 , 0 , [1/5] , 2.0 , ε)
(8 , 5:0 , [1/5] , 2.0 , ε)

$\mathcal{C}[\![\text{LOAD } n]\!](m, d, h, inp, out) =$
wenn $h(n) \in \mathbb{Z}$, dann $(m+1, h(n) : d, h, inp, out)$

Wiederholung — Floyd-Warshall-Algorithmus

Gegeben: Distanzgraph (V, E, c) mit:

- $V = \{1, \ldots, n\}$ für ein $n \geq 1$
- $E \subseteq V \times V$
- $c : E \longrightarrow \mathbb{R}^+$

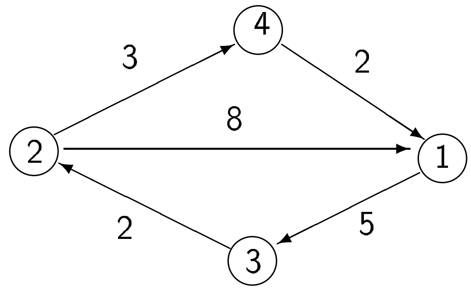
Gesucht: minimaler Aufwand, um auf einem Weg von i nach j zu gelangen, für beliebig vorgegebene $i, j \in V$

Ansatz: $W_G = (W(i,j) \mid 1 \leq i, j \leq n)$, wobei

$$W(i,j) = \begin{cases} c(i,j) & \text{wenn } i \neq j, (i,j) \in E \\ 0 & \text{wenn } i = j \\ \infty & \text{wenn } i \neq j, (i,j) \notin E \end{cases}$$

$$\begin{aligned} W^0(i,j) &= W(i,j) \\ W^{k+1}(i,j) &= min\,\{W^k(i,j)\ ,\ W^k(i,k+1) + W^k(k+1,j)\} \end{aligned}$$

Beispiel:



$$\rightsquigarrow \; W_G = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & \infty & 3 \\ \infty & 2 & 0 & \infty \\ 2 & \infty & \infty & 0 \end{pmatrix}$$

$$W^1 = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & 13 & 3 \\ \infty & 2 & 0 & \infty \\ 2 & \infty & 7 & 0 \end{pmatrix}, W^2 = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & 13 & 3 \\ 10 & 2 & 0 & 5 \\ 2 & \infty & 7 & 0 \end{pmatrix}$$

$$W^3 = \begin{pmatrix} 0 & 7 & 5 & 10 \\ 8 & 0 & 13 & 3 \\ 10 & 2 & 0 & 5 \\ 2 & 9 & 7 & 0 \end{pmatrix}, W^4 = \begin{pmatrix} 0 & 7 & 5 & 10 \\ 5 & 0 & 10 & 3 \\ 7 & 2 & 0 & 5 \\ 2 & 9 & 7 & 0 \end{pmatrix}$$

Floyd-Warshall-Algorithmus in Haskell (I)

```
data Entry = Inf | F Int

w_g :: [[Entry]]
w_g = [[F 0,Inf,F 5,Inf],
       [F 8,F 0,Inf,F 3],
       [Inf,F 2,F 0,Inf],
       [F 2,Inf,Inf,F 0]]

n :: Int
n = length w_g

ws :: [[[Entry]]]
ws = [wm k | k <- [0..n]]
```

Floyd-Warshall-Algorithmus in Haskell (II)

```
wm :: Int -> [[Entry]]
wm 0     = w_g
wm (k+1) = [[min (wk i j)
                 (wk i (k+1) `plus` wk (k+1) j)
             | j <- [1..n]]
            | i <- [1..n]]
   where wk i j = ws !! k !! (i-1) !! (j-1)

min,plus :: Entry -> Entry -> Entry

min Inf b = b
min a Inf = a
min (F a) (F b) = F (if a<b then a else b)

Inf `plus` b = Inf
a `plus` Inf = Inf
(F a) `plus` (F b) = F (a+b)
```


Floyd-Warshall-Algorithmus in Haskell — Gesamtprogramm

```
module Main where

...

w_g :: [[Entry]]
w_g = [[F 0,Inf,F 5,Inf],
       [F 8,F 0,Inf,F 3],
       [Inf,F 2,F 0,Inf],
       [F 2,Inf,Inf,F 0]]

...

main = ...
```

Floyd-Warshall-Algorithmus in Haskell — Test

```
> main
[0,*,5,*]
[8,0,*,3]
[*,2,0,*]
[2,*,*,0]

[0,*,5,*]
[8,0,13,3]
[*,2,0,*]
[2,*,7,0]

[0,*,5,*]
[8,0,13,3]
[10,2,0,5]
[2,*,7,0]

...
```