Haskell Live

[10] Software Transactional Memory in Haskell, Tortenwurf und Aufgabenblatt 7

Bong Min Kim
e0327177@student.tuwien.ac.at

Christoph Spörk
christoph.spoerk@inode.at

Florian Hassanen
florian.hassanen@googlemail.com

Bernhard Urban
lewurm@gmail.com

10. Dezember 2010

Software Transactional Memory

Siehe `10stm.zip`. Entpacken und mit `make` builden. Man benötigt dafür die Pakete `stm`, `network` `regex-base` und `regex-compat` die man auf `www.haskell.org` finden kann (falls sie nicht schon installiert sind). Danach kann man den server mit `./server` in der Shell starten und sich mit `telnet` <`hostaddr`> `8888` verbinden. Viel spass `:-)`

Ein äußerst lesenswertes Paper über STM von Simon Peyton-Jones (Erfinder von Haskell und GHC Entwickler): `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/#beautiful`

Tortenwurf

```haskell
import Data.List

type Person = Int; type Reihe = [Person]

permutation :: Reihe → [Reihe]
permutation [] = [[]]
permutation xs = [x : ys | x ← xs, ys ← permutation (delete x xs)]

  -- torten werfer
  -- 1.Variante
cakeThrowerVisible1 :: Reihe → [Person]
cakeThrowerVisible1 list = [p | i ← [0 .. l], let p = list !! i, (i ≡ 0 ∨ p > maximum (take i list))]
  where l = (length list) − 1

  -- 2.Variante
cakeThrowerVisible2 :: Reihe → [Person]
cakeThrowerVisible2 list = cakeThrowerVisible2_ 0 list

cakeThrowerVisible2_ _ [] = []
cakeThrowerVisible2_ maxSoFar (p : ps)
   | p > maxSoFar = p : (cakeThrowerVisible2_ p ps)
   | otherwise = cakeThrowerVisible2_ maxSoFar ps

  -- torten opfer
cakeVictimVisible1 :: Reihe → [Person]
cakeVictimVisible1 list = cakeThrowerVisible1 (reverse list)

cakeVictimVisible2 :: Reihe → [Person]
cakeVictimVisible2 list = cakeThrowerVisible2 (reverse list)

  -- brute force solution
solveCakeCrime :: Int → Int → Int → [Reihe]
solveCakeCrime n p r = [per | per ← (permutation [1 .. n]),
  length (cakeThrowerVisible2 per) ≡ p,
  length (cakeVictimVisible2 per) ≡ r]
```

Aufgabenblatt 7

isPostfix

$$\begin{array}{l}
\mathbf{type}\ \mathit{State} = \mathit{Integer}; \mathbf{type}\ \mathit{StartState} = \mathit{State}\\
\mathbf{type}\ \mathit{AcceptingStates} = [\mathit{State}]\\
\mathbf{type}\ \mathit{Word}\ a = [a]; \mathbf{type}\ \mathit{Row}\ a = [[a]]\\
\mathbf{data}\ \mathit{AMgraph}\ a = \mathit{AMg}\ [(\mathit{Row}\ a)]\ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})\\
\mathbf{type}\ \mathit{Automaton}\ a = \mathit{AMgraph}\ a\\
\mathbf{type}\ \mathit{Postfix}\ a = \mathit{Word}\ a\\
\mathit{isPostfix} :: \mathit{Eq}\ a \Rightarrow (\mathit{Automaton}\ a) \rightarrow \mathit{StartState} \rightarrow \mathit{AcceptingStates} \rightarrow (\mathit{Postfix}\ a) \rightarrow \mathit{Bool}\\
\mathit{isPostfix}\ \mathit{ma}\ s\ \mathit{end}\ \mathit{post} =\\
\quad \mathbf{case}\ \mathit{givePrefix}\ \mathit{ma}\ s\ \mathit{end}\ \mathit{post}\ \mathbf{of}\\
\qquad \mathit{Nothing} \rightarrow \mathit{False}\\
\qquad \mathit{Just}\ _ \rightarrow \mathit{True}
\end{array}$$

givePrefix

$$\begin{array}{l}
\mathbf{type}\ \mathit{Prefix}\ a = \mathit{Word}\ a\\
\mathit{getTransitions} :: \mathit{Eq}\ a \Rightarrow (\mathit{Automaton}\ a) \rightarrow \mathit{State} \rightarrow (\mathit{Row}\ a)\\
\mathit{getTransitions}\ (\mathit{AMg}\ \mathit{rows})\ s = \mathit{rows}\ !!\ (\mathit{fromIntegral}\ s)\\
\mathit{getStates} :: \mathit{Eq}\ a \Rightarrow (\mathit{Automaton}\ a) \rightarrow [\mathit{Integer}]\\
\mathit{getStates}\ (\mathit{AMg}\ \mathit{rows}) = [0\,..\,(\mathit{fromIntegral}\ \$\ \mathit{length}\ \mathit{rows} - 1)]\\
\mathit{givePrefix} :: \mathit{Eq}\ a \Rightarrow (\mathit{Automaton}\ a) \rightarrow \mathit{StartState} \rightarrow \mathit{AcceptingStates} \rightarrow (\mathit{Postfix}\ a) \rightarrow (\mathit{Maybe}\ (\mathit{Prefix}\ a))\\
\mathit{givePrefix}\ \mathit{ma}\ s\ \mathit{end}\ \mathit{post}\\
\quad \mid \mathit{ret} \equiv [\,] = \mathit{Nothing}\\
\quad \mid \mathit{otherwise} = \mathit{Just}\ \$\ \mathit{head}\ \mathit{ret}\\
\quad \mathbf{where}\ \mathit{ret} = \mathit{givePrefix'}\ \mathit{ma}\ s\ \mathit{end}\ ((\mathit{getStates}\ \mathit{ma}) \setminus\!\setminus [s])\ s\ [\,]\ \mathit{post}
\end{array}$$

```
givePrefix' :: Eq a ⇒ (Automaton a) → StartState → AcceptingStates → [State] → State → (Word a) → (Postfix a) → [(Prefix a)]
givePrefix' ma s end tovisit akt word post
  | accept ma s end (word ++ post) = [word]
  | otherwise = concat
    [concat
      [givePrefix' ma s end newtv ii (word ++ [e]) post
       | e ← (transitions !! i)   -- jede mögliche Transition durchprobieren
      ]
    | i ← [0 .. (length transitions) − 1]
    , let ii = fromIntegral i
    , let newtv = tovisit \\ [ii]   -- den zu bearbeiteten Knoten im nächsten rekursiven Aufruf ausschließen
    , ¬ $ null (transitions !! i)   -- nur wenn es mind. einen Übergang gibt
    , ii ∈ tovisit   -- nur wenn dieser Knoten in der “ToDo-Liste” steht
    ]
  where transitions = getTransitions ma akt
accept :: Eq a ⇒ (Automaton a) → StartState → AcceptingStates → (Word a) → Bool
accept ma src sinks [] = src ∈ sinks
accept ma@(AMg matrix) src sinks (kante : xs) =
  or [accept ma (fromIntegral y) sinks xs
     | y ← [0 .. ((length row) − 1)]
     , kante ∈ (row !! y)
    ]
  where row = matrix !! (fromIntegral src)
```

traverse

```
type Vertex = Integer; type Origin = Vertex; type Destination = Vertex
data ALbgraph a = ALbg [(Origin, a, [Destination])] deriving (Eq, Show)
traverse :: Eq a ⇒ (a → a) → (a → Bool) → (ALbgraph a) → (ALbgraph a)
traverse _ _ (ALbg []) = ALbg []
```

$traverse\ f\ p\ (ALbg\ ((k, a, dest) : xs))$
$\quad \mid p\ a = ALbg\ ((k, (f\ a), dest) : rek)$
$\quad \mid otherwise = ALbg\ ((k, a, dest) : rek)$
$\quad$ **where** $(ALbg\ rek) = traverse\ f\ p\ (ALbg\ xs)$

isWellColored

data $Color = Red \mid Blue \mid Green \mid Yellow$ **deriving** $(Eq, Show)$

data $Ugraph = Ug\ [(Origin, Color, [Destination])]$ **deriving** $(Eq, Show)$

$getOrigin :: Ugraph \rightarrow Origin \rightarrow (Origin, Color, [Destination])$
$getOrigin\ (Ug\ [\,])\ \_ = error$ `"Leerer Graph"`
$getOrigin\ (Ug\ ((o, c, d) : xs))\ oo$
$\quad \mid o \equiv oo = (o, c, d)$
$\quad \mid otherwise = getOrigin\ (Ug\ xs)\ oo$

$getColor :: Ugraph \rightarrow Origin \rightarrow Color$
$getColor\ ug\ o = s2$
$\quad$ **where** $(\_, s2, \_) = getOrigin\ ug\ o$

$isWellColored :: Ugraph \rightarrow Bool$
$isWellColored\ ug@(Ug\ list) = isWellColored'\ ug\ allorg$
$\quad$ **where** $allorg = [\,o \mid (o, \_, \_) \leftarrow list\,]$

$isWellColored' :: Ugraph \rightarrow [\,Origin\,] \rightarrow Bool$
$isWellColored'\ \_\ [\,] = True$
$isWellColored'\ ug\ (o : orgs)$
$\quad \mid or\ [\,c \equiv (getColor\ ug\ n) \mid n \leftarrow dests\,] = False$
$\quad \mid otherwise = isWellColored'\ ug\ orgs$
$\quad$ **where** $(\_, c, dests) = getOrigin\ ug\ o$