Christoph Lüth
Marcus Ermler
Christian Maeder
Martin Ring
Diedrich Wolter

7. Übungsblatt

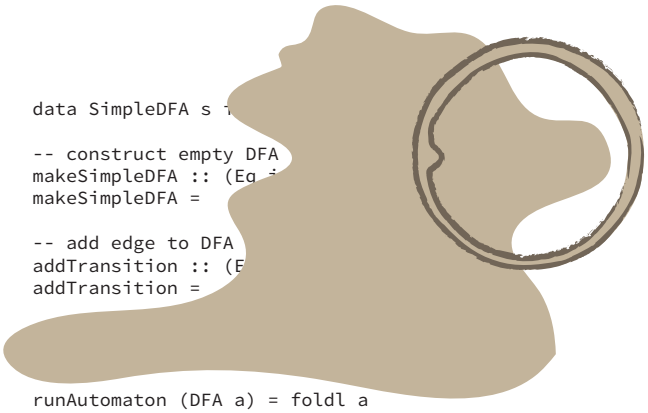
Ausgabe: 05.12.12
Abgabe: 14.12.12

In diesem Jahr feiert die Informatik den 100. Geburtstag von Alan Turing, einem der einflussreichsten Theoretiker auf dem Gebiet der Berechenbarkeit. Deshalb soll hier ein Geburtstagskanon von Aufgaben zur Automatentheorie und -praxis angestimmt werden. Automaten finden zum Beispiel Anwendung in der Abbildung von Prozessen.

Ein Kernbegriff der durch Turing geprägten Automatentheorie ist der des Zustandes als Abstraktion eines Momentes in einer komplexen Berechnung. Funktionale Programmierung kommt hingegen bewusst zustandsfrei daher. Deshalb sollen diese Aufgaben Ihnen auch aufzeigen, dass dies kein Widerspruch sein muss, sondern lediglich eine Frage der Betrachtung ist.

7.1 *Zustände sind das!* — *8 Punkte*

Funktionale Programme können elegant knapp sein—im vorliegenden Fall wurde dies jedoch zum Verhängnis ihres Stellen-Vorinhabers. Ein einzelner Kaffeefleck auf dem letzten Ausdruck der als verschollen geglaubten Software zur Ampelsteuerung hat sowohl der glücklichen Wiederentdeckung als auch der Stellen-Inhaberei ihres Stellen-Vorinhabers den garaus gemacht (siehe Abb. 1). Nehmen Sie die Herausforderung an, den Quellcode für endliche Automaten samt Ampelsteuerung zu rekonstruieren?



```
data SimpleDFA s

-- construct empty DFA
makeSimpleDFA :: (Eq
makeSimpleDFA =

-- add edge to DFA
addTransition :: (E
addTransition =

runAutomaton (DFA a) = foldl a

-- simple test case: cat automaton
data CatStates = Sleepy | Hungry deriving (Eq,Show)
data CatAction = Sleep| SeeMouse | GetFood deriving (Eq,Show)

catDFA = simpleDFAFromList [(Sleepy, Sleep,   Hungry), (Sleepy, SeeMouse, Hungry),
                            (Hungry, GetFood, Sleepy), (Hungry, SeeMouse, Hungry)]

-- test:"runAutomaton catDFA (Just Sleepy) [Sleep,SeeMouse,GetFood]"->"Just Sleepy"
```

Abbildung 1: Quel malheur! Das verschüttete Programm.

1. Ein Praktikant hat bereits die sichtbaren Teile des Programmes abgetippt[1]. Rekonstruieren Sie das „verschüttete“ Programm, indem Sie die Typdefinition für endliche Automaten (engl.: **D**eterministic **F**inite **A**utomaton) sowie die Definition der Funktionen wiederherstellen.
2. Erstellen Sie einen Automaten für eine „bedarfsgesteuerte Fußgängerfurt“ nach dem Muster in Abb. 2.

[1] Auf der Webseite zu finden.

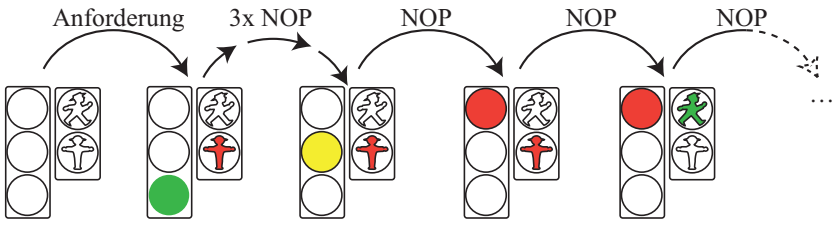


Abbildung 2: Bedarfsgesteuerte Fußgängerfurt.

7.2 *Sichere Automaten* — *4 Punkte*

Nach üblicher Definition gelangen Automaten bei einer nicht behandelten Eingabe in einen nicht definierten Zustand aus dem sie nicht wieder herauskommen. Man sagt auch, ein Automat blockiert. Um sicherzustellen, dass dies nicht aufgrund eines Modellierungsfehlers geschieht, soll eine Testfunktion entwickelt werden, die feststellt, ob ein Blockieren von einem (Start-)zustand aus möglich ist und ggf. eine entsprechende Folge von Eingaben zurückgibt.

1. Wie lässt sich Blockieren umsetzen? Erweitern Sie ihren Automaten gegebenfalls!
2. Nehmen wir zunächst an, dass sowohl Zustandstyp s als auch Eingabetyp i endlich, bekannt und vergleichbar sind. Entwickeln Sie einen Algorithmus für die Testfunktion ausgehend von einem Startzustand unter Verwendung der endlichen Liste von allen Zuständen und möglichen Eingaben!

   ```
   canBlock :: (Eq s, Eq i) ⇒ SimpleDFA s i → s → [s] → [i] → Maybe [i]
   ```

7.3 *Von automatischen Automaten zu fleißigen Bibern* — *8 Punkte*

Um beliebig komplexe Berechnungen zu meistern sind mächtigere Automatenmodelle notwendig, wie zum Beispiel die Turingmaschine (TM). Eine TM verfügt über einen Bandspeicher, der, zusammen mit dem Zustand, die sogenannte Konfiguration einer TM darstellt. Eine Eingabe ist nicht notwendig, da der Bandspeicher diese Rolle übernehmen kann. Fleißige Biber sind besondere Turingmaschinen, deren Aufgabe es ist, möglichst viele Einsen auf das Arbeitsband zu schreiben und dann zu terminieren (erreichen eines Endzustandes). Das Bandalphabet eines Bibers ist gegeben durch $\{0, 1\}$. Ein fleißiger Biber wird auf einem mit Nullen initialisiertem Band gestartet und nach Terminierung werden die Anzahl der zusammenhängenden Einsen gezählt. Das Bemerkenswerte an fleißigen Bibern ist, dass es algorithmisch unlösbar ist (d.h., sie ist unentscheidbar), den fleißigsten Biber mit n Zuständen zu bestimmen. In Abb. 3 sind zwei Biber dargestellt, der linke ist der fleißigste 1-Zustandsbiber (Haltezustände werden nicht mitgezählt)!

1. Erstellen Sie einen Datentyp zur Darstellung einer TM für fleißige Biber! (Tipp: betrachten Sie den Biber als 2-Kellerautomat)
2. Erstellen Sie eine Funktion `runBeaver :: Beaver -> Integer -> Maybe (Integer,Integer)` die fleißige Biber simuliert und die Anzahl der Zustandsübergänge und erzeugten Einsen bestimmt!

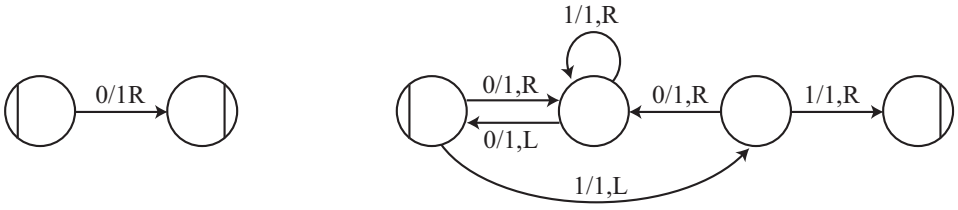


Abbildung 3: Zwei fleißige Biber. Eine Kante $r/w, M$ beschreibt eine Transition die ausgeführt wird, falls der Kopf den Wert r liest; es wird w geschrieben und der Kopf bewegt sich danach einen Schritt in Richtung $M \in \{L, R\}$.

Um die Terminierung sicherzustellen wird eine Maximalzahl der Zustandsübergängen angegeben, bei deren Überschreitung wird die Simulation abgebrochen und `Nothing` zurückgeliefert wird.

3. Setzten Sie die in Abb. 3 dargestellten Biber um und entwerfen Sie eigene Biber mit 3 und 4 Zuständen!
4. Finden Sei einen fleißigeren Biber mit 3 Zuständen als den abgebildeten? (*1 Bonuspunkt*)
5. Finden Sie einen fleißigeren Biber mit 4 Zuständen als die anderen Gruppen? (*2 Bonuspunkte*)

? *Verständnisfragen*

1. Was ist ein abstrakter Datentyp (ADT)?
2. Was sind Unterschiede und Gemeinsamkeiten zwischen ADTs und Objekten, wie wir sie aus Sprachen wie Java kennen?
3. Wozu dienen Module in Haskell?