Finite State Machines

A First Example for Turning Mathematics into Haskell Programs

Wolfram Kahl

3 January 2005

We develop a simple implementation of some finite-state machine concepts in Haskell for demonstrating the ease with which mathematical definitions can be turned into functional programs.

For the sake of simplicity, everything will be part of a single Haskell module `FSM` that exports the interface to all its contents — such a module declaration has to be the first code line of a Haskell file.

```
module FSM where
```

The usual definition of finite-state machines with input alphabet Σ presents a finite-state machine as a quadruple

$$(\mathcal{S}, s_0, \mathcal{F}, \delta)$$

consisting of the following components:

- a set $\mathcal{S}$ of arbitrary elements — the elements of $\mathcal{S}$ are now called "*states*"
- a *start state* $s_0 : \mathcal{S}$
- a set $\mathcal{F} : \mathbb{P}\,\mathcal{S}$ of *final states*, and
- a *transition relation* $\delta : \mathcal{S} \times \Sigma \leftrightarrow \mathcal{S}$

Such a finite-state machine is called *deterministic* if δ is deterministic, i.e., a (partial) function. (Any finite-state machine can easily be converted into an equivalent *total* finite-state machine by adding a "catch-all" state to $\mathcal{S}$ and adding transitions to this catch-all state for all pairs outside $\operatorname{dom}\delta$.)

If this definition of finite-state machines is considered in a typed setting, the set of states is typically a subset of some set (or type) of "potential states" — for the sake of convenience we fix this type of potential states and introduce a type synonym for it:

```
type State = Int
```

This convention makes the state set $\mathcal{S}$ superfluous as part of the finite-state machines for most purposes — more specifically, for all those purposes where only states occurring in the transition relation are relevant.
We also fix the symbol type and introduce a corresponding type synonym:

```
type Symbol = Char
```

We have to find an appropriate type for the implementation of the transition relation δ. There is no primitive relation type constructor in Haskell, so first of all we use the isomorphism between relations and set-valued functions ($A \leftrightarrow B \;\widehat{=}\; A \to \mathbb{P}\,B$) to move to

$$\mathcal{S} \times \Sigma \to \mathbb{P}\,\mathcal{S} \;.$$

Next we use *currying* to be able to supply the arguments one at a time — the general isomorphism is $(A \times B) \to C \;\widehat{=}\; A \to (B \to C)$ — so we now have:

$$\mathcal{S} \to (\Sigma \to \mathbb{P}\,\mathcal{S}) \ .$$

Looking at this, we notice that a different choice of argument order would yield as result type of the outer function type constructor a state transition relation, which is a useful concept when working with automate, so we decide to switch the argument order ($A \times B \;\widehat{=}\; B \times A$) and obtain:

$$\Sigma \to (\mathcal{S} \to \mathbb{P}\,\mathcal{S}) \ .$$

Finally we decide to implement sets as lists (or sequences) — an inefficient, but easy-to-use choice:

$$\Sigma \to (\mathcal{S} \to \mathrm{seq}\,\mathcal{S})$$

We define a type synonym for this type of transition relations:

```
type TransRel = Symbol -> State -> [State]
```

A simple example of a transition relation of this type:

```
tr1 :: TransRel
tr1 'a' 0 = [1]
tr1 'a' 2 = [1]
tr1 'b' 1 = [0,2]
tr1 _ _   = []       -- ``_'' denotes a nameless ``wildcard'' variable
```

Even without considering the other components of a finite-state machine, we can now "lift" the transition relation from single symbols as input to whole words as input:

```
wordTrans :: TransRel -> [Symbol] -> State -> [State]
wordTrans tr []         s = [s]
wordTrans tr (sy : sys) s = tr sy s >>= wordTrans tr sys
```

The infix operator `>>=` is used here at type

```
[State] -> (State -> [State]) -> [State]
```

and collects into its own result the results of applying its second argument (which is a function expecting a `State`) to each element of its first argument (which is a list of `State`s).
As an experiment, we can check the states reachable in `tr1` via some word:

```
*FSM> wordTrans tr1 "ababab" 0
[0,2,0,2,0,2,0,2]
```

According to the discussion above, we can now implement a finite-state machine as a triple:

```
type FSM = (TransRel, State, [State])
```

Acceptance of a word by a non-deterministic finite-state machine is defined as membership of any final state in the states reachable via that word from the start state — we use prelude functions to express this directly:

```
accepts :: FSM -> [Symbol] -> Bool
accepts (tr, s0, fin) sys = any (`elem` fin) (wordTrans tr sys s0)
```