DFAs

Greg Kobele

November 19, 2018

A `DFA` is a 5-tuple $\langle Q, \Sigma, q_0, F, \delta\rangle$ consisting of

1. a set Q of states
2. a set Σ of symbols
3. a start state q_0
4. a set F of final states
5. and a (partial) function δ mapping state-symbol pairs to states

We can simply write this directly in Haskell.

```
type DFA state symb =
  ([state],[symb],state,[state],(state,symb) -> state)
```

Note that this is a definition of a *type synonym*; we have taken an already existing type, and given it a new name. Note further that we use *lists* in the Haskell definition to implement *sets*.

1 Some machines

Now that we have a *type* for (deterministic) finite state machines, we can define objects which have that type. We begin with a machine which rejects all words over the alphabet $\Sigma = \{a, b, c\}$ containing a b; in linguistic terms, it is a *constraint* against b.

```
notB :: DFA Int Char
notB = ([0,1],['a','b','c'],0,[0],deltaB)
  where
    deltaB (0,'b') = 1
    deltaB (0,_) = 0
    deltaB (1,_) = 1
```

Our next machine recognizes strings over the alphabet $\Sigma = \{0, 1\}$ containing both an *even* number of the letter *0* and an *odd* number of the letter *1*. However, instead of an alphabet of $\Sigma = \{0, 1\}$, we will use `Bool`, with `False` standing for 0 and `True` for 1. That is, we will *represent* the input `[1,0,0]` as the list `[True,False,False]`.

```
evenFoddT :: DFA String Bool
evenFoddT = (["ee","eo","oe","oo"],[False,True],"ee",["eo"],deltaEO)
  where
    delta1 ("ee",False) = "oe"
    delta1 ("eo",False) = "oo"
    delta1 ("oe",False) = "ee"
    delta1 ("oo",False) = "eo"
    delta1 ("ee",True) = "eo"
    delta1 ("eo",True) = "ee"
    delta1 ("oe",True) = "oo"
    delta1 ("oo",True) = "oe"
```

2 Recognition

Recognition requires an implementation of the following mathematical condition:

$$A \text{ recognizes } w \text{ iff } \delta^*(q_0, w) \in F$$

This can be rendered almost verbatim in Haskell.

```
a@(qs,sigma,q0,fs,delta) `recognizes` input =
  deltaStar (q0,input) `elem` fs
  where
    deltaStar (q,[]) = q
    deltaStar (q,(b:bs)) = deltaStar ((delta (q,b)),bs)
```

In the above code, we used an *as-pattern* in the first argument of `recognize`. An *as*-pattern has the following syntax: `var @ pattern`, where `var` is a variable name, and `pattern` is any pattern. This allows us to give a name (`var`) to the entire argument, and at the same time break it down into subparts using pattern matching. In the code above, the name for the argument `a` was just used to enhance readability. In addition, the function `deltaStar` was defined *locally* to the function `recognize` in a *where* clause. This means that `deltaStar` is visible only inside of `recognize`, and is not defined in the global name space.

Somewhat unsurprisingly, the action of `deltaStar` is a common one, when dealing with lists: given a starting value (here q), walk through a list element by element, each time updating the current value somehow (here *delta*). This transforms a list of the form `a:b:c:d:[]` is to a sequence of function applications `f (f (f (f i a) b) c) d` where f is the *update rule* and i the *initial value*. Going through a list one element at a time, updating the results, is called *folding* through a list, and depending on whether the initial value is combined first with the initial (leftmost) or final (rightmost) element. The behaviour of `deltaStar` is a *left fold*, and is implemented in Haskell as `foldl`. We can thus define `deltaStar` (internally to `recognizes`) using `foldl` in the following way.

```
deltaStar (q,input) = foldl (curry delta) q input
```

This definition is complicated by the fact that `delta` is defined as taking both its arguments simultaneously (as a *pair*), instead of taking them one after the other. The function `curry` turns a function that wants its next two arguments simultaneously into one that expects them one after the other. Using the inverse of `curry`, called `uncurry`, we can simplify the definition of `deltaStar` somewhat.

```
deltaStar = uncurry $ foldl (curry delta)
```

Note that `delta` needs to be curried, because `foldl` wants a function that takes arguments one after the other, and then the expression `foldl (curry delta)` is uncurried because `deltaStar` wants the next two arguments of this expression to be given simultaneously. The operator `($)` is simply an explicit representation of function application (so `f a` is the same thing as `f $ a`), however, it has a very low *precedence*, which means roughly that Haskell tries to interpret the entirety of what comes to its right as the argument, and the entirety of what comes to its left as the function. This can allow us to save on parentheses, as the sequence of function applications `f (g a)` can be given as `f $ g a`.

3 Revisiting partiality

We have been using an *implicit* representation of partiality; *delta* was just left undefined whenever it was, well, undefined! This can cause problems (run-time errors) in our programs, making their behaviour unpredictable. Here we reimplement machines so as to make the partiality of the transition function explicitly represented. We use for this a `Maybe` data type.

```
type DFA state symb =
  ([state],[symb],state,[state],(state,symb) -> Maybe state)
```

The type of `DFA` now says explicitly that the transition function will only *maybe* return a next state. This forces us to be more careful in functions that use DFAs, but guarantees that they will not crash our programs!

```
recognizes :: Eq q => DFA q s -> [s] -> Bool
a@(qs,sigma,q0,fs,delta) `recognizes` input =
  case deltaStar (q0,input) of
    Nothing -> False
    Just q -> q `elem` fs
  where
    deltaStar (q,[]) = Just q
    deltaStar (q,(b:bs)) =
      do
        q' <- delta (q,b)
        deltaStar (q',bs)
```

To deal with the possibility that `delta` returns nothing, `deltaStar` has been rewritten using *do*-notation. Inside a *do*-block, we extract the contents of a `Maybe` value, if any, by writing `q' <- delta (q,b)`. If `delta (q,b)` is `Just x`, this extracts the state `x` and binds it to `q'`. If `delta (q,b)` is `Nothing`, the entire *do*-block returns `Nothing`.[1] Now that `deltaStar` only *maybe* returns a state, we must deal with this possibility in the code. The code is written with an explicit case analysis (using the keywords `case ... of`).

We can redefine the previous machine accepting words with an even number of the letter *0*, and an odd number of *1* (represented as truth values). We take advantage of the regularities in the transitions, and encode the states as pairs of boolean values, where a state `(b,c)` says whether we have seen an odd number of *0* (if `b == True`), and whether we have seen an odd number of *1* (`c == True`).

```
evenFoddT :: DFA (Bool,Bool) Bool
evenFoddT = (qEO,[False,True],(False,False),[(False,True)],deltaEO)
  where
    qEO = [(b,c) | b <- [False,True], c <- [False,True]]
    deltaEO ((b,c),False) = Just (not b, c)
    deltaEO ((b,c),True) = Just (b, not c)
```

[1] The *do*-block is syntactic sugar (i.e. a pretty syntactic abbreviation) for the code: `delta (q,b) >>= \q' -> deltaStar (q',bs)`, at which the entire thing might better be written in terms of `foldl`: `deltaStar = uncurry $ foldl (\accu i -> accu >>= flip (curry delta) i)`.

4 Testing

We would like to be sure that our code does what we think it should (i.e., that I did not make a mistake while typing). One way to do this is to have a test suite! We can create a list of examples, and check whether our code performs correctly on these examples.

```
test1 = []
test2 = [True]
test3 = [False,True]
test4 = [False,False,False,True,False]
test5 = [False,False,False,True,False,False,True]
```

Haskell reports the following results:

```
*Main> evenFoddT `recognizes` test1
False
*Main> evenFoddT `recognizes` test2
True
*Main> evenFoddT `recognizes` test3
False
*Main> evenFoddT `recognizes` test4
True
*Main> evenFoddT `recognizes` test5
False
```

Developing a good test suite is hard work! We need to make sure we pick good examples, which are diverse enough to instantiate all the difficulties of the problem. Of course, each program we attempt to write may have different bugs, and it is not clear that a static test suite will find all of them.

Another, 'fairer' way of testing our code is to generate random examples. We can leverage the functionality of the `Test.QuickCheck` module for this. First we create a property (a function from objects of interest to `True` or `False`).

```
prop_soundEO :: [Bool] -> Property
prop_soundEO s =
  acceptedEO s ==> (evenFalse s && oddTrue s)
  where
    acceptedEO = recognizes evenFoddT
    evenFalse = even . length . filter (not . id)
    oddTrue = odd . length . filter id
```

This property is `True` of a string accepted by `evenFoddT` if the string has an even number of the letter `False` and an odd number of `True`. We can define a related property of a string, which is had whenever a string with an even number of `False` and an odd number of `True` is accepted by `evenFoddT`

```
prop_completeEO :: [Bool] -> Property
prop_completeEO s =
  (evenFalse s && oddTrue s) ==> acceptedEO s
  where
    acceptedEO = recognizes evenFoddT
    evenFalse = even . length . filter (not . id)
    oddTrue = odd . length . filter id
```

Then we can pass these properties to the `quickCheck` function, to see whether 100 randomly generated inputs satisfy it.

```
*Main> quickCheck prop_soundEO
+++ OK, passed 100 tests.
*Main> quickCheck prop_completeEO
+++ OK, passed 100 tests.
```