

A Collection of Functional Libraries for Theory of Computation

Guo-Qiang Zhang and Leon Smith
Department of EECS, Olin 610
Case Western Reserve University
10900 Euclid Avenue, Cleveland, OH 44106
email: gqz@eeecs.cwru.edu
url: <http://vorlon.ces.cwru.edu/~gqz>

Abstract. This paper reports the design and initial implementation of an extensive collection of libraries for automata theory and Turing machines in the functional programming languages SML and Haskell. Such an effort has two benefits: one is conceptual clarity, in the sense that computational concepts and construction algorithms match directly and faithfully to types and functions in the implementation, especially in terms of recursion; the other is conciseness and efficiency brought by the unusual features of these functional programming languages unimaginable in the traditional imperative setting. In SML, we provide a simulation of finite state machines and Turing machines by taking advantage of SML's structures, functors, and exceptional handling mechanism. The simplicity of this implementation makes it valuable for classroom use. In Haskell, we provide an implementation of deterministic and non-deterministic finite state machines, the conversion from the non-deterministic one to a deterministic one, state minimization, and constructions such as union, intersection, concatenation, and Kleene star on non-deterministic machines. Conversion from regular expressions to finite state machines can be achieved straightforwardly in terms of these constructions. The benefit of the Haskell implementation is the great freedom in the representation of actual automata to be used with the package. This freedom has been enhanced by using Multi-Parameter Type Class extensions of the Haskell type system to allow the various constructions to make very few assumptions about the data they use.

Key words: Automata theory, theory of computation, functional programming, SML, Haskell.

1 Introduction

The conceptual clarity and conciseness of functional programming languages such as SML [5] and Haskell [8] make them ideal languages in which to implement finite state machines and Turing machines, for both educational and research purposes. However, there has not been a systematic implementation of automata theory (DFA, NFA, Turing Machines, constructions) in these lan-

guages. The closest functional implementation is by Thompson [9] using Miranda, in a very limited way.

This paper is concerned with the simulation of automata theory (DFA, NFA, Turing machines, constructions, grammars, regular expressions) in the functional language SML and Haskell. Such an effort has two benefits: one is conceptual clarity, in the sense that computational concepts and construction algorithms match directly and faithfully to types and functions in the implementation, especially in terms of recursion; the other is conciseness and efficiency brought by the unusual features of these functional programming languages unimaginable in the traditional imperative setting. In SML, we provide a simulation of finite state machines and Turing machines by taking advantage of SML's structures, functors, and exceptional handling mechanism. The simplicity of this implementation makes it valuable for classroom use. In Haskell, we provide an implementation of deterministic and non-deterministic finite state machines, the conversion from the non-deterministic one to a deterministic one, state minimization, and constructions such as union, intersection, concatenation, and Kleene star on non-deterministic machines. The benefit of the Haskell implementation is the great freedom in the representation of actual automata to be used with the package. This freedom has been enhanced by using Multi-Parameter Type Class extensions of the Haskell type system to allow the various constructions to make very few assumptions about the data they use.

In the first part of the paper we describe the design and implementation of deterministic Turing machines and finite automata in SML. We then describe an implementation of an extensive set of constructions in Haskell. Comparison of the implementations is provided in the conclusion.

Remark. *This paper is not meant to be an introduction to functional programming in either SML or Haskell. Some knowledge of functional programming is necessary in order to understand and appreciate the contents of this paper.*

2 Turing machines in SML

Several ideas come to mind when one considers the implementation of Turing machines in a functional language. For example, a Turing machine can be represented as a list of tuples, with each tuple specifying the current state, the scanned symbol, the action, and the next state. This is probably the most likely implementation of Turing machines in an imperative language, or even a functional one (Thompson 1995 [9]). However, this representation does not seem to fully reflect the advantages offered by a functional language. To determine the next configuration, one has to match the current state with a tuple in the list, and then update the configuration accordingly. Moreover, the tuple list may not represent a function at all.

Another way is to represent both the state and the transition function as functions (see Exercise 11.3.30 of Mitchell 1996 [4]; to be fair, this example is for type inference), and the “next configuration” is determined by function application. This representation is sensitive to the number of

states a Turing machine has, for the arity of the functions depends on the total number of states. Thus, coding of states requires a bit of effort, and the implementation is not that intuitive.

We present an implementation of deterministic Turing machines in SML that attempts to be as intuitive and straightforward as possible. We use integers to represent states and use characters for tape symbols. The tape is two-way infinite. It can be naturally coded as a triple `(char list) * char * (char list)`. In our implementation, the transition function is represented as a (curried) function from states to tape symbols to state-action pairs. The advantage of a function, instead of a list of tuples, for the transition function is that the next-state can be found by a straightforward function application. Since we are interested in language acceptors here only, a computation can accept a string, reject a string, or be non-terminating.

2.1 Design

A Turing machine has a finite control which determines, for each step, whether it should move left, move right, or write something, and then enter a new state. This entails the use of some basic types in the structure for the implementation, such as `Gamma`, for tape symbols, and `State` for states. It is nature to let `Gamma` be `char` and `State` be `int`.

Tape. In the literature, two versions of Turing machines exist: one of them work on a one-way infinite tape, and the other work on a two-way infinite tape. However, it is mandatory to describe the tape in terms of three parts – the position of the read-write head, the tape content left of it, and the tape content right of it. A two-way infinite tape fits the description naturally without the restriction of a left end. The type for tape is

```
type Tape = (Gamma list) * Gamma * (Gamma list)
```

Transition function. The finite control of a Turing machine is specified by a transition function. Again, there are different versions of such a transition function in the literature, depending on what the machine is allowed to do at each step. Most definitions allow a Turing machine to both move the read-write head *and* change the content of the tape cell. To specify such an action, one needs to use a tuple indicating the direction of head-move as well as the what is been written. However, most of the time a machine makes a head-move without changing the tape content. Thus the tuple type will result in a lot of redundancy and this is not what we adopt.

Instead, we follow the idea of Barwise and Etchmenny (1993 [1]) in their design of Turing's World 3.0. At each step, a Turing machine does exactly one thing: it either moves left, or moves right, or makes a content change without a head-move. This entails the type for Turing machines' actions:

```
datatype Action = L | R | W of char
```

where **L** stands for a left-move, **R** for a right-move, and **W** of **char** for the writing of a character onto the current tape cell. The type of transition function is then

```
type Delta = State -> Gamma -> (State * Action)
```

Transition functions of this kind require less effort to code. Action and new state can be found by a direct application of the function. This matches perfectly with our conception of a Turing machine.

Machine. We implement the Turing machine package in a structure called **TM**. A machine consists of a triple indicating the initial state, the transition function, and the accepting state. A configuration is a state-tape tuple. Basic functions include **next**, for determining the next configuration; **delta_star**, for repeating **next** until the machine reaches the accepting state or else (see Section 2.3). In the case of acceptance, **delta_star** returns the string "accepted!". The function **runtm** receives an input to a Turing machine, sets up the initial configuration, and applies **delta_star** to run the machine. The function **next** can also be used to step-run the machine (by issuing **next it** in SML).

2.2 Implementation

We use Standard ML of New Jersey, Version 110.0.3 for the implementation. Since a tape is represented as a triple of type $(\text{Gamma list}) * \text{Gamma} * (\text{Gamma list})$, a special character must be singled out to denote a blank tape cell. We use the special character #, given in SML as **##**, for this purpose.

Tape contents change according to an action. The auxiliary function **change** is written for this purpose.

```
fun change (L, ([], cell, right))      = ([], blank, cell::right)
  | change (L, (ll::left, cell, right)) = (left, ll, cell::right)
  | change (R, (left, cell, []))        = (cell::left, blank, [])
  | change (R, (left, cell, rr::right)) = (cell::left, rr, right)
  | change ((W x), (left, cell, right)) = (left, x, right);
```

This function reflects the “two-stack” simulation of a Turing machine. A slight modification of the function can produce a more intuitive tape layout.

Function **next** determines the next configuration with respect to a Turing machine.

```
fun next (x,tm) =
  let val (curr, (left, cell, right)) = x
      val (_, delta, _) = tm
      val (newstate, act) = (delta curr cell)
```

```

    in ((newstate, change (act, (left, cell, right))), tm)
end;

```

The fourth line above, `val (newstate, act) = (delta curr cell)`, takes advantage of the fact that `delta` is a (curried) function. In a tuple list presentation of the transition function, more work is need to implement this line.

Another auxiliary function called `init` puts the input string on the tape appropriately so that the read-write head rests on the first symbol of the string.

```

fun init(x:string) =
  let val gs = explode(x)
      in ([], hd(gs), tl(gs))
  end;

```

Functions `delta_star` and `runtm` are routine to implement.

2.3 Exception handling

The implementation seems to be rather straightforward after the design phase, except for one thing. When we actually produce an example machine and test it, we either end successfully in an accepting state, or reach a point where the current configuration is out of the scope of the transition function's domain (uncaught exception). This is an ideal setting where SML's elegant exception handler can be put to use.

The outcome is a structure called `TM`, packaging all the types and functions together:

```

structure TM=
struct
  exception Abort;
  type Gamma      = char;
  type State      = int;
  datatype Action = L | R | W of Gamma;
  type Tape       = (Gamma list) * Gamma * (Gamma list);
  type Config     = State * Tape;
  type Delta      = State -> Gamma -> (State * Action);
  type Machine    = State * Delta * State;

  val blank= "#";

  fun change (L, ([], cell, right))      = ([], blank, cell::right)
    | change (L, (ll::left, cell, right)) = (left, ll, cell::right)

```

```

| change (R, (left, cell, []))          = (cell::left, blank, [])
| change (R, (left, cell, rr::right)) = (cell::left, rr, right)
| change ((W x), (left, cell, right)) = (left, x, right);

fun next (x,tm) =
  let val (curr, (left, cell, right)) = x
      val (_, delta, _) = tm
      val (newstate, act) = (delta curr cell)
  in ((newstate, change (act, (left, cell, right))), tm)
  end;

fun init(x:string) =
  let val gs = explode(x)
  in ([], hd(gs), tl(gs))
  end;

fun delta_star (x,tm)=
  let val (curr,_) = x
      val (_,_,accepts) = tm
  in if curr=accepts then "accepted!"
     else delta_star (next(x,tm))
  end
  handle Abort => "rejected!";

fun runtm(x, tm) =
  let val (initial,_,_) = tm
  in delta_star ((initial,init x),tm)
  end;
end;

```

2.4 Example

A six state Turing machine is coded for the language $\{a^n b^n c^n \mid n \geq 0\}$. The idea is for the machine to scan the tape from left to right and mark off a , b , c one by one until every symbol in the input string has been marked off.

```

open TM;
val initial =0:State;
val accept=1:State;

```

```

val delta:Delta = fn
0 => (fn #"x"=>(0,R) | #"#"=>(1,R) | #"a"=>(2,(W #"x")) | _=>raise Abort) |
2 => (fn #"a"=>(2,R) | #"x"=>(2,R) | #"b"=>(3,(W #"x")) | _=>raise Abort) |
3 => (fn #"b"=>(3,R) | #"x"=>(3,R) | #"c"=>(4,(W #"x")) | _=>raise Abort) |
4 => (fn #"c"=>(4,R) | #"x"=>(4,R) | #"#"=>(5,L)          | _=>raise Abort) |
5 => (fn #"#"=>(0,R) | _=>(5,L))      |
_ => raise Abort;

val tm:Machine = (initial,delta,accept);

```

Just as one would expect, `runtm("aaabbbccc",tm)` leads to "accepted!". Similarly, `runtm("aaabbbccc",tm)` leads to "rejected!".

One can see that implicit in the coding of the transition function is another advantage. We simply associate a function to a state, avoiding the repetition of the argument states, such as in

$$[(0, \text{"x"}, 0, R), (0, \text{"#"}, 1, R), (0, \text{"a"}, 2, W \text{"'x'})]$$

if one were to code the first line of `delta` as tuples. There is, however, one benefit of coding the transition function as a list of tuples: the `raise Abort` part can be completely avoided here but it reappears in the main body of a modified version for the function `next`.

3 Deterministic finite automata in SML

Although some versions of deterministic finite automata (DFA) only require the transition function to be partial (e.g. [1]), we follow the main stream tradition (e.g. [2]) of requiring the transition function to be a (total) function. (Other aspects of automata theory, such as the Myhill-Nerode Theorem, depends on the totality.)

We develop a simulation of DFA that takes advantage of ML's module system, especially *functors*. The module system allows user-defined alphabets and states. But once an alphabet `sigma` and a state set `state` is fixed, whether a function of a certain kind really is a *transition function* reduces to whether the function is defined everywhere on `sigma * state`. This is exactly where the ML interpreter provides useful feedback: the function is *well-defined* if the interpreter compiles the function without emitting `Warning: match nonexhaustive!`

3.1 Specification

The signatures `Sigma`, `State`, and `Automaton` define the alphabet, the state set, and functions for automaton-simulation. The function `next` determines the next configuration of a DFA by returning the new state with the remaining input symbols. Each application of `next` consumes exactly one symbol from the input list. The function `delta_star` recursively applies the function `next` from

a starting configuration until all the input symbols have been consumed. It returns the resulting state.

```
signature Sigma = sig eqtype sigma end;

signature State = sig eqtype state end;

signature Automaton =
  sig
    eqtype sigma;
    eqtype state;
    type dfa;
    val next: sigma list * state * dfa -> sigma list * state * dfa;
    val delta_star: sigma list * state * dfa -> state;
    val accept: sigma list -> dfa -> bool;
  end;
```

3.2 Implementation

The implementation is realized by a functor called `DFA`. The functor receives two structures, one of `Sigma` and the other of `State`, and returns a resulting structure of `Automaton`.

```
functor DFA(structure symbols:Sigma; structure states:State):Automaton =
  struct
    type sigma = symbols.sigma;
    type state = states.state;
    type dfa = state * (state->sigma->state) * state list;

    fun isin (x, []) = false
      | isin (x, y::ys) = if x=y then true else isin(x, ys);

    fun next([], curr, M) = ([],curr,M)
      | next(s::ss, curr, M as (initial, delta, final))
        = (ss, (delta curr s), M);

    fun delta_star(xs, curr, M) = if xs=[] then curr
      else delta_star (next(xs,curr,M));

    fun accept (w: sigma list) (M:dfa) =
      let val (initial, delta, final) = M
```

```

        val last = delta_star(w,initial,M)
    in
        isin(last,final)
    end;
end;

```

The function `isin` tests membership, which should not belong here. Functions `next`, `delta_star` and `accept` are straightforward, with `accept` returning a truth value.

3.3 Example

The pay-off of the careful design and the use of functors can be appreciated more concretely through an example. Here is a four states DFA over the alphabet $\{a, b\}$. The transition function, `delta`, is given using ML's anonymous function construct. The definitional uniformity lies in the fact that associated with each state is again a function, this time of type `two -> four`.

```

structure two = struct datatype sigma = a | b end;

structure four = struct datatype state = A | B | C | D end;

structure DFA1 = DFA (structure symbols = two; structure states = four);
open two;
open four;
open DFA1;

val delta = fn A=> (fn a => B | b => C)
            | B=> (fn a => D | b => A)
            | C=> (fn a => A | b => D)
            | D=> (fn a => C | b => B);
val example = (A, delta, [B,C]);

```

Of course, to fully specify a DFA we also need to indicate the initial state and the set of final states, as is given in the value `example` above. Test on the DFA shows that

```
accept [a,b,a,a,b,b] example;
```

returns `true`, and `accept [a,b,b] example;` returns `false`. This means that the string *abaabb* is accepted, while *abb* is not.

4 Finite state machines and constructions in Haskell

We now move to our Haskell implementation of the standard constructions in automata theory: DFA, NFA, union, intersection, concatenation, Kleene star, and minimization of DFA.

We represent automata as a triple consisting of a transition function, a start state, and a function from the set of states to a Boolean value representing the set of final states, similar to the SML implementation. However, in this Haskell implementation, the types for states and alphabets are more flexible. *This allows automata to make their own choices of implementing their transition functions.* Some automata may be able to exploit domain-specific properties to create functions that are more efficient than their table-based counterparts. As the functions that work with automata are more generic, hopefully the package is more useful.

```
newtype (Alphabet ab) => DFA st ab =
    MkDFA (st -> ab -> st, st, st -> Bool)

accepts :: (Alphabet ab) => DFA ab -> [ab] -> Bool
accepts (MkDFA (delta, start, isFinal)) string =
    isFinal (foldl delta start string)
```

To represent the `Alphabet`, we chose to introduce a type class with a variable that holds a list of all the letters in the alphabet. Using this along with the graph reachability algorithm, it is straightforward to compute the set of reachable states. This is essential for some constructions such as state minimization.

```
class Eq ab => Alphabet ab where
    alphabet :: [ab]
```

Two particular kinds of `Alphabet` are defined below. One is the alphabet of digits, and the other is the Boolean alphabet.

```
instance Alphabet Digit where
    alphabet = map mkDigit [0..9]

instance Alphabet Bool where
    alphabet = [False, True]
```

The function `mkDigit` takes an integer between 0 and 9 as the argument and returns a `Digit`. We also provide the function `mkDigits` which takes an integer and converts it to a list of digits.

One remarkable feature of this representation is its ability to represent automata with an infinite number of states. Many of our constructions will still work correctly with infinite automata, those that don't are usually inherently limited to finite automata, such as state minimization. Thus, our

package facilitates experimenting with infinite automata. Furthermore, we doubt that inadvertently creating infinite automata will be a problem in practice.

Now, we have everything necessary to build a few sample automata. The first one checks whether or not a string of boolean values has an odd number of trues and falses. *The first is a particularly good example of the transition function exploiting domain-specific properties in a way that would not be possible in an implementation based inherently around transition tables.*

```
odd_tfs :: DFA (Bool, Bool) Bool
odd_tfs = MkDFA ( d, s, f )
  where
    d (x,y) a = if a then (not x, y) else (x, not y)
    s          = (False,False),
    f x        = x == (True, True)
```

This DFA takes a string of Boolean values and returns true if the number of true values and the number of false values in the string are both odd.

Starting up Hugs (see [3] – a Haskell interpreter), we can interactively play with what we have defined. Our package implements the standard Haskell type class `Show`, so that when we type in an expression that has the type `DFA`, hugs will display a textual representation of the value. So this is what we get for the automata just created.

```
>> odd_tfs

((False,False), False) -> (False,True )
((False,False), True ) -> (True ,False)
((False,True ), False) -> (False,False)
((False,True ), True ) -> (True ,True )
((True ,False), False) -> (True ,True )
((True ,False), True ) -> (False,False)
((True ,True )*, False) -> (True ,False)
((True ,True )*, True ) -> (False,True )
```

In our textual representation, the start state is listed first, and the final states are marked by a star. Here is a test run of the DFA:

```
>> odd_tfs 'accepts' map ('1' ==) "001011001010"
True
```

The second DFA checks that a string of decimal digits is divisible by some number. It is actually a function for which we get a DFA for each integer n .

```

divisible :: Int -> DFA Int Digit
divisible n = MkDFA ( d, s, f )
  where
    d remainder digit = (10 * remainder + fromDigit digit) `mod` n
    s                  = 0
    f remainder       = remainder == 0

```

Here are some test runs:

```

>> divisible 7 'accepts' mkDigits 53784323
False

```

```

>> divisible 9 'accepts' mkDigits 53784324
True

```

We've provided 6 basic constructions for dealing with DFA: `crossDFA`, `unionDFA`, `dfa2nfa`, `reachableDFA`, `factorDFA`, and `minimizeDFA`. The standard constructions are easy and straightforward to define. For example, here is the code for the product (intersection) of two DFA:

```

crossDFA :: (Alphabet ab) => DFA st1 ab -> DFA st2 ab -> DFA (st1,st2) ab
crossDFA (MkDFA (d1, s1, f1)) (MkDFA (d2, s2, f2)) = MkDFA (d3, s3, f3)
  where
    d3 (st1, st2) a = (d1 st1 a, d2 st2 a)
    s3              = (s1, s2)
    f3 (st1, st2)  = f1 st1 && f2 st2

```

Likewise, `unionDFA` is the same, except that we use `(||)` (or) instead of `(&&)` (and) in the new final state function.

The last three constructions require the use of sets and finite maps. We have defined two multi-parameter type classes as interfaces to finite maps and sets, and provided an implementation of each based on the data structure of 2-3-4 trees. Here is a partial interface to each:

```

module Set ... where

class I set a where
  fromList :: [a] -> set a
  toList   :: set a -> [a]

  empty    :: set a

```

```

has      :: set a -> a -> Bool
add      :: set a -> a -> set a

module FiniteMap ... where

class I f a b where
  empty      :: f a b
  hasKey     :: f a b -> a -> Bool
  addKey     :: f a b -> (a,b) -> f a b
  changeKey  :: f a b -> a -> b -> (b -> b) -> f a b
  apply      :: f a b -> a -> b
  applyMaybe :: f a b -> a -> Maybe b
  applyDef   :: b -> f a b -> a -> b

```

The advantage of type classes is that we don't have to write our code using one specific implementation of a finite map or set. Multi-parameter type classes allow the implementation, not the interface, to define the range of appropriate types that the set contains. With single-parameter type classes (such as in our SML implementation), implementations are required to work with all appropriate types as set forth by the interface, and thus cannot make any further constraints on the type. Thus, with single-parameter type classes, we would probably have to resort to constraining the types that sets can hold to ordered types in the set interface in order to get reasonable implementations. This would rule out implementations that can work with unordered types, or more efficient implementations that work with more tightly constrained datatypes.

State reachability is fairly straightforward. We also provide a variant, `reachableDFA_list`, that returns the queue it used for the algorithm. The stack represents the states that are left to visit, and the set represents all the states that have been inserted into the stack.

```

reachableDFA :: (Alphabet ab, Set.I set st) => DFA st ab -> set st
reachableDFA (MkDFA(d,s,f)) = reachable' ([], Set.fromList [s])
  where
f (stack, set) state
  | set 'Set.has' state = (stack, set)
  | otherwise          = (state:stack, set 'Set.add' state)
reachable' ([], set) = set
reachable' (x:xs, set) = reachable' (foldl f (xs,set) (map (d x) alphabet))

```

We provide a special function, `factorDFA`, which takes a one-to-one function and a DFA and returns a automata equivalent to the first, except that states have been renamed using the function, the transition is represented as a finite map, and the final function is represented as set membership.

This is particularly useful after performing constructions on the automaton to compute a more efficient representation of the automaton.

Finally, `minimizeDFA` is implemented using the algorithm described in Hopcroft and Ullman [2]. We start with a collection of all the unordered pairs of the states of the DFA, and cross out all pairs where one state is a final state and the other is a nonfinal state. Then, for each unordered pair $\{x, y\}$ and each letter l , we calculate $p' = \{dxl, dyl\}$. If this pair p' is crossed out, then we cross out $\{x, y\}$, otherwise we mark p' such that if it is ever crossed out, we also cross out $\{x, y\}$. Once we've completed this task, all of the remaining pairs that have not been crossed out are states that are equivalent to each other. At this point, our algorithm takes each equivalence class, chooses a single representative state from each, and represents the new transition function as a map from each state to its representative state composed with the transition function. At this point, calling `factorDFA` would more than likely be worthwhile, as the resulting DFA will be less efficient than before.

To implement unordered pairs, we assume that the set of states of the DFA have an ordering defined on them, and then assure that the first element in the pair is less than the second element. A empty finite map from pairs of states to an optional list of pairs of states is used to hold the necessary information about each pair. The value `Nothing` means that pair has been crossed out, and the value `Just xs` is used to hold all the pairs that need to be crossed out if that pair gets crossed out.

Then, to choose each representative element, we start with an empty finite map from states to states. We then step through each pair of states $\{x, y\}$, where $x \leq y$, in lexicographic order. If y is not associated with anything in our finite map, we associate y with x . Here is the complete code in Haskell for minimization:

```

minimizeDFA_using
  :: (Ord st, Alphabet ab, FM.I fm (st,st) (Maybe [(st,st)]), FM.I fm' st st)
  => fm (st,st) (Maybe [(st,st)]) -> fm' st st -> DFA st ab -> DFA st ab
minimizeDFA_using emptymap emptymap' m@(MkDFA (d,s,f))
  | nonfinal == [] = MkDFA (const, s, const True)
  | final     == [] = MkDFA (const, s, const False)
  | otherwise      = MkDFA (d',s',f)
where
  d' st ab = FM.apply repElems (d st ab)
  s'       = FM.apply repElems s

states = Set.toList (reachable m)
where
  reachable = reachableDFA :: (Ord a, Alphabet b) => DFA a b -> Set.T234 a

```

```

(final, nonfinal) = List.partition f states

repElems = foldl (\fm (y,x) -> FM.changeKey fm y x id) emptymap'
  [ (y,x)
  | (x:xs) <- tails states, y <- (x:xs),
  x == y || FM.applyDef (Just []) minsts (x,y) /= Nothing ]

minsts = foldl (trans alphabet) minsts_0 ( pairs states )
  where
    minsts_0 = foldl FM.addKey emptymap
      [ (sortPair (x,y),Nothing) | x <- final, y <- nonfinal ]

trans []      fm pair = fm
trans (ab:abs) fm pair@(x,y)
  | x' == y'  = trans abs fm pair
  | otherwise = case (FM.applyDef (Just []) fm pair') of
    Nothing -> crossout fm pair
    Just xs  -> trans abs ( FM.addKey fm (pair', Just (pair:xs)) ) pair
  where pair'@(x',y') = sortPair (d x ab, d y ab)

crossout fm pair = case (FM.applyDef (Just []) fm pair) of
  Nothing -> fm
  Just xs  -> foldl crossout fm' xs
  where fm' = FM.changeKey fm pair (Nothing) (const Nothing)

tails :: [a] -> [[a]]
tails [] = []
tails xs = xs : tails (tail xs)

pairs :: [a] -> [(a,a)]
pairs list = [ (x,y) | (x:xs) <- tails list, y <- xs ]

sortPair (x,y)
  | x <= y    = (x,y)
  | otherwise = (y,x)

```

4.1 Nondeterministic Finite Automata

The kind of NFA we have implemented is NFA without ϵ -moves, that is, we do not allow our NFA to switch states without consuming a letter from the input. If we were to introduce ϵ -moves,

we would have to be careful of our simulation algorithms behavior on cycles of ϵ -moves, which is important as they could potentially be introduced using our various constructions. Eliminating them ensures our algorithm to simulate the NFA will terminate. Also, it leads to a simpler, more efficient representation, and it did not pose any difficulty in implementing NFA constructions, as we can simply take the ϵ -closure whenever we want to introduce an ϵ -move. Thus, when we say we introduced an ϵ -move from state x to state y , we really mean that we added all the transitions of y to the transitions of x .

Our representation of NFA is very similar to the representation of a DFA, except that the transition relation is represented by a function from a state to a letter to a list of states. We simulate the NFA using a simple backtracking algorithm.

```
newtype (Alphabet ab) => NFA st ab =
  MkNFA (st -> ab -> [st], st, st -> Bool)

instance Automaton (NFA b) where
  accepts (MkNFA (delta, start, isFinal)) string = accepts' start string
  where
    accepts' state []      = isFinal state
    accepts' state (x:xs) = case (delta state x) of
      [] -> False
      [s] -> accepts' s xs
      ss -> foldr (\s a -> accepts' s xs || a) False ss
```

One could rewrite the second equation of `accepts'` by replacing the case statement by the last alternative in the case statement. The case statement is so that when the transition function has only one alternative, `accepts'` can be tail-recursive. This often is a common case for NFA.

We provide seven basic kinds of constructions on NFA: `crossNFA`, `unionNFA`, `concatNFA`, `kleeneNFA`, `nfa2dfa`, `reachableNFA`, and `factorNFA`. The function `crossNFA` is implemented in an analogous way as `crossDFA`, but `unionNFA` was implemented by introducing a new start state, and added ϵ -moves from the new start state the start state of x and the start state of y . To represent the new set of states, we used `Nothing` as the new start state, `Just (Left x)` as the set of states for the left argument, and `Just (Right x)` as the set of states for the right argument.

```
unionNFA :: (Alphabet ab) =>
  NFA st1 ab -> NFA st2 ab -> NFA (Maybe (Either st1 st2)) ab

unionNFA (MkNFA (d1, s1, f1)) (MkNFA (d2, s2, f2)) = MkNFA (d3, s3, f3)
  where
    d3 Nothing      x = map (Just . Left) (d1 s1 x)
      ++ map (Just . Right) (d2 s2 x)
```

```

d3 (Just (Left a)) x = map (Just . Left) (d1 a x)
d3 (Just (Right a)) x = map (Just . Right) (d2 a x)
s3                          = Nothing
f3 (Nothing)                = (f1 s1 || f2 s2)
f3 (Just (Left a))         = f1 a
f3 (Just (Right a))        = f2 a

```

The function `concatNFA` was implemented by introducing an ϵ -move from each final state of x to the start state of y . Finally, the `kleeneNFA` introduces ϵ -moves from each final state of x to the start state of x . We use `(Left x)` to represent the states of the left argument, and `(Right x)` to represent the states of the right argument. `kleeneDFA` was implemented in a very analogous way, except that an ϵ -move was added from each final state to the start state.

```

concatNFA :: (Alphabet ab) =>
  NFA st1 ab -> NFA st2 ab -> NFA (Either st1 st2) ab

```

```

concatNFA (MkNFA (d1, s1, f1)) (MkNFA (d2, s2, f2))
  = MkNFA (d3, s3, f3)

```

where

```

d3 (Left x) a = map Left (d1 x a) ++ if (f1 x) then
  (map Right (d2 s2 a)) else []
d3 (Right y) a = map Right (d2 y a)
s3 = Left s1
f3 (Left _) = False
f3 (Right y) = f2 y

```

```

kleeneNFA :: (Alphabet ab) => NFA st ab -> NFA st ab
kleeneNFA (MkNFA (d, s, f)) = MkNFA (d', s, f)

```

where

```

d' st a = d st a ++ if (f st) then (d s a) else []

```

5 Conclusion

We believe that in some sense, *the code is the documentation* for functional programming. Functional expressions make precise mathematical sense due mainly to referential transparency, or lack of side-effects. The mathematical concepts and algorithms used in automata theory can be coded fairly directly in SML and Haskell, as our implementation shows. Our main contribution is in the careful selection of type structures used in representing automata-theoretic concepts, allowing both flexibility and simplicity.

We have not yet reached a definite choice between SML and Haskell: as with any implementation, there are trade-offs and we intend to keep both versions.

There are two main advantages with SML. One is the the exception handling mechanism, and the other is SML's interactive user interface, which allows for more efficient interaction with the interpreter. For example, in the SML implementation of DFA, since we are using enumeration types for the alphabet and the states, the use of functors allows the ML interpreter to check whether a transition function of type `state->sigma->state` is total or not: the function is total exactly when the interpreter does not give the “match non-exhaustive” warning. If it were not for this feature, we would have to use a tuple list to represent a function, and to write an auxiliary function to check if a tuple list indeed defines a total function.

The main advantage of Haskell is in the multi-parameter type class extensions. This allows a flexible representation of states and alphabets in a canonical way, offering a good balance of type constraint and flexibility.

As can be seen, our package can be extended with various other features. Regular expressions and grammars come next – their implementation will follow the same style. In the implementation of grammars and parsing, we may be able to take advantage of Haskell's built-in monad structures. Although our implementation is not yet complete, we have achieved in a short time what may have taken much longer to accomplish in the imperative setting – in a more reliable, reusable, and concise way.

6 The packages

The SML and Haskell packages are available for downloading at

<http://vorlon.cwru.edu/~gqz/automata.html>

You need to install SML and Hugs (a Haskell interpreter) in order to run them, though. These can be obtained at [7] for SML and [3] for Haskell.

References

- [1] J. Barwise and J. Etchemendy. *Turing's World 3.0* for the Macintosh. *Lecture Notes No. 35*, CSLI Publications, Stanford, California, 1993.
- [2] J.E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [3] <http://haskell.cs.yale.edu/hugs/>
- [4] J. Mitchell. *Foundations of Programming Languages*. The MIT Press, 1996.
- [5] L. Paulson. *ML for the Working Programmer*, 2nd edition. Cambridge University Press, 1996.

- [6] D. Raymond and D. Wood. Grail: A C++ Library for Automata and Expressions *Journal of Symbolic Computation* 17(4), p. 341-350, 1994.
- [7] <http://cm.bell-labs.com/cm/cs/what/smlnj/smlnj.html>
- [8] S. Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, 1996.
- [9] S. Thompson. *Regular Expressions and Automata using Miranda*, http://stork.ukc.ac.uk/computer_science/Miranda_craft/regExp.