Adaptive DFA – the development of adaptable methods

Dan Popa
Univ. "Vasile Alecsandri", Calea Mărăşeşti 157, Bacău, Romania
danvpopa@ub.ro, popavdan@yahoo.com

Abstract: This paper is a part of a study concerning the development of adaptable methods and their use in the field of Formal Languages and Automata Theory. The author is introducing the Adaptive Determinist Finite Automata (ADFA) using mathematics inspired by the use of the very high level functional language Haskell 98. The training method, the way of use, examples and the history of the Adaptive Determinist Finite Automata together with the applications already built on that concept are also presented.

1.Introduction and overview

The long-term goals of this research were:
To increase productivity and efficiency in a world where the time is short and projects need to be flexible.
To implement the idea from [Arm01]: "Do not shoot balloons with cannons, shoot the fighter-planes with thermal missiles!", in software building methodology and especially in language and interpreters building methodology.
To develop the theory of adaptive/adaptable languages, tools and products. (specific goal)
Five years ago, in [Pop 04] and [Pop 05] the Adaptive DFA was described and presented. Checking the classic references like [Aho07] (even the updated volumes), leads us to the conclusion that ADFA are a different model / machine for languages recognition.
Various implementations was made during last years using Oberon-2, using C and C++ by the original author and his students. [Bal 08], [Sme 09] .
Now we are coming back with a mathematical point of view concerning Adaptive DFA suggested by the use of the very high level functional language Haskell [Pey 02].

2.Preliminaries

The mathematics of Adaptive DFA is here described using a notation high related to Haskell.

Functions will be denoted by long names: *f(x)* will be used together with, for ex: *funct(x)* and even *funct x*. Multiple parameters functions will be written nor as *function (a, b)* but as *function a b* , which is a common practice for Haskell programmers.

The sets we use here will be, actually, ordered sets with eventually duplicated elements (lists). They are written using right parenthesis [].
Examples :
x= [1,4,5]
[x | x <- a , x >3]

Every program is usually having some auxiliary functions, so here they are ours:

Intersection of two lists, using this notation

```
intersect a b =
    [ c1 | c1 <- a, c2 <-b , c1 == c2]
```

Adding spaces at the end of the string s is an other function we needed:

```
addspace s = ' ':s++" "
```

Haskell programmers may note that it can be written as (not so fast version):

```
addspace s = " "++s++" "
```

3.Classes of characters

According to the paper [Pop 05] where Adaptive DFA was mathematically presented for the first time, the characters processed by an Adaptive DFA are, first of all classified in :
-Letters,
-Digits,
-Spaces etc.
The process is similarly with a part of the lexical analysis.

We have used a simple Haskell function to compute the class of symbol, the classes being identified by characters:

-Letters, the 'l' class
-Digits, the 'c' class
-Spaces , the '_' class
-Others , the '?' class

```
clasa a =
    if (a >= 'a' && a <= 'z') ||
       (a >= 'A' && a <= 'Z')
        then 'l'
        else if (a >= '0' && a<= '9')
            then 'c'
            else if a == '\t' || a =='\n' || a ==' ' then '_'
                else '?'

-- 'l' = alphabetic,  'c' = digits,  '_' = spaces
```

4.Preparing the words for storage

In our approach, classifying the characters from a new word
means two successive processing: first of all adding spaces around the word then classify the resulted string character by character. What we get will be called a "scheme" of a word. For example: "_ccc_" is a scheme produced by a three digits number.
The process can be implemented as a Haskell function:

```
clasifica = (map clasa). addspace
```

Where:
-- *map* is the usual *map* of the functional languages which applies a function to all the elements of a list. For example : map *f* [x,y,z] = [*f* x, *f* y, *f* z]
-- 'dot' is the product of functions (reverse composition) as it is defined in the standard library of the Haskell language.

5.Simulating the storage in the matrix

The original paper [Pop 05] had used a bi-dimensional matrix. Here we are using an other data structure which in fact simulate the storage in that rare matrix. Here we are explaining the process of storage of the schemes in the datastructure:
For every triple (x,y,z) (x,y,z being classes of successive symbols of the word) we will preserve the schemes of those words in a list which is associated with the triple, becoming the 4th element.
This list which is associated with a triple can be easily found by filtering the dictionary itself using a sort of substring function:

```
filter (substr (x,y,z))  dict
```

where the substring filter is defined by the next two equations:

```
substr (x,y,z) (c1:c2:c3:t) =
    if c1==x && c2==y && c3==z
    then True
    else substr (x,y,z) (c2:c3:t)
substr (x,y,z) (c1:c2:[]) = False
```

How it works: if the sequence of classes "xyz" is found somewhere in the scheme of a word, this fact triggers the placement of that scheme in that list which is associated with the triple.

6.The trained Adaptive DFA

So, he result which is produced by processing the whole dictionary can be computed by the next function:

```
automat dict =
 [ (x,y,z, filter (substr (x,y,z))  dict )   |  x <- n,
   y <- n ,
   z <- n  ]
   where
     n = "lc_"     -- n = map clasificare "D2 "
```

-- Note: The dictionary which is used here should be the list of schemes of the words serving as training examples.

7.Rebuilding examples from previous papers

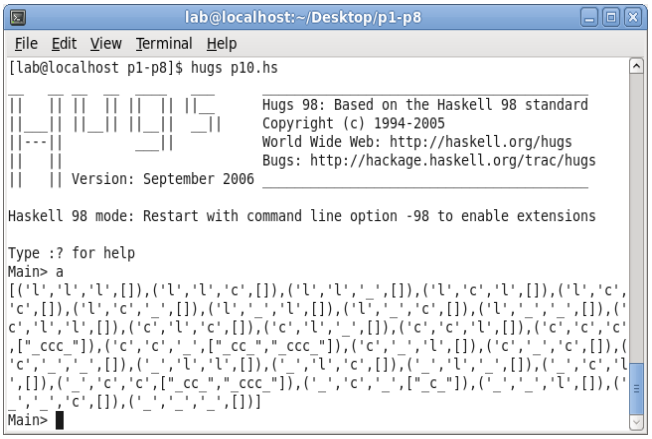
Now, the adaptive DFA from [Popa05] which was trained to accept numbers can be simply defined as:

```
a = automat ["_c_", "_cc_", "_ccc_"]
```

Or, using examples and the classification function:

```
a = automat [ clasifica "0", clasifica "21",  clasifica "196"]
```

Now, the adaptive DFA from [Popa05] can be computed by simply asking Hugs or GHCi to produce an explicit value:



Here is the value:
[('l','l','l',[]),('l','l','c',[]),('l','l','_',[]),('l','c','l',[]),('l','c','c',[]),('l','c','_',[]),('l','_','l',[]),('l','_','c',[]),('l','_','_',[]),('c','l','l',[]),('c','l','c',[]),('c','l','_',[]),('c','c','l',[]),('c','c','c',["_ccc_"]),('c','c','_',["_cc_","_ccc_"]),('c','_','l',[]),('c','_','c',[]),('c','_','_',[]),('_','l','l',[]),('_','l','c',[]),('_','l','_',[]),('_','c','l',[]),('_','c','c',["_cc_","_ccc_"]),('_','c','_',["_c_"]),('_','_','l',[]),('_','_','c',[]),('_','_','_',[])]

We may want to simplify the list, by ignoring the (x,y,z,[]) quadruples. A function which is able to eliminate such quadruples is:

```
simplu a = [ x | x <- a , lastp x /= [] ]
           where  lastp (_,_,_,y)= y
```

By using this function, we are able to filter the important quadruples:

Main> simplu a
[('c','c','c',["_ccc_"]),('c','c','_',["_cc_","_ccc_"]),('_','c','c',["_cc_","_ccc_"]),('_','c','_',["_c_"])]

We will denote this simple form by: *a' = simplu a*
But we will not study it here.

8. Using a trained ADFA

A scheme of a word being given, the analysis is implemented by the folowing function:

```
analiza cuvant automat=
  [ m | (x,y,z,m) <- automat  , (x,y,z)  `elem` triplete cuvant]
```

The parameters are the scheme of the word and the ADFA.
And, if you want to trace the computation you may use:

```
trace cuvant automat=
  [ (x,y,z,m)| (x,y,z,m) <- automat ,
               (x,y,z)  `elem` triplete cuvant]
```

...where the scheme of the input text is broken in "triples" computed by the next function:

```
triplete ::[Char] -> [(Char,Char,Char)]
triplete (a:b:c:d) = (a,b,c) : (triplete (b:c:d ))
triplete (b:c:_)   = []
```

Note: in the previous paragraph, x `elem` m denotes the test "if the element x belongs to the list m " and is provided as a standard Haskell operator.

9. Back to the analyzer's engine

Taking a closer look to the function implementing the processing of the (scheme of a) word using an ADFA

```
analiza cuvant automat=
  [ m | (x,y,z,m) <- automat
      , (x,y,z)  `elem` triplete cuvant]
```

we can remark: The produced list may contains more sets of "schemes". If one "scheme" appears in all this sets -> the word is accepted. See the next paragraph.

10. Accepting a word

Now we can define in which situations the word is accepted. We are defining a sort of *acceptance by intersection,* because
when the ADFA is processing a token, it can identify more than one set of schemes partially matching that token. So we have just compute the intersection:

```
acceptare cuvant automat
  = foldl intersect (head a ) a
    where
      a =  analiza cuvant automat
```

Remark: the above function can also be written as:

```
acceptare cuvant automat
  = foldl intersect (head a ) (tail a)
    where
      a =  analiza cuvant automat
```

11. Acceptance criteria

The intersection contains **one or more** schemes . This is the case when the input is **accepted**. It means there exist one or more schemes (and correspondingly there are some examples in the set which was used for training) matching the given word (actually, it's scheme!).

The intersection did not contain a common scheme, so it is the empty list [ ]. In this case the input is **not accepted**. None of the words from the set which was used for training could provide a whole set of "triples".

12. The trained ADFA is working now !

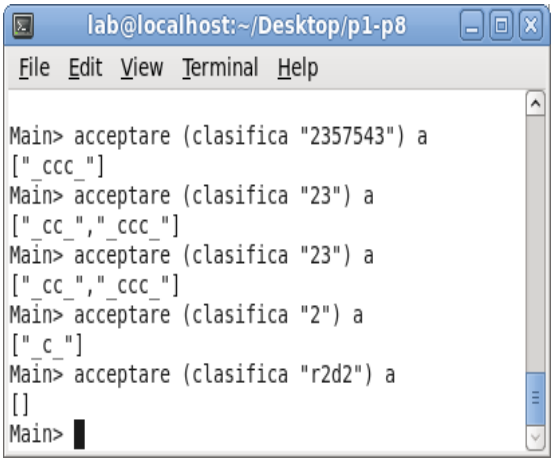
Here are some words which are given to the automata defined in the 7th paragraph. Below each one, there is the corresponding answer of the trained system and a comment:

> acceptare (clasifica "2357543") a
["_ccc_"]
Interpretation: this is a number composed by three or more digits.

> acceptare (clasifica "23") a
["_cc_", "_ccc_"]
Interpretation: this is a number composed by two or three or more digits.

> acceptare (clasifica "2") a
["_c_"]
Interpretation: this is a number composed by just one digit.

> acceptare (clasifica "r2d2") a
[]
Interpretation: The empty list [] being returned, this word is not accepted. This is not a number in the sense of examples provided during the training of the ADFA.



13. Advantages

Even with a good textbook in hand concerning the use of Flex / Lex - a common scanner generator - actually the main problem concerning scanner's production is to find a person which know al least 3 “languages”: C, regular expressions, Lex/Flex specifications language or format.

Working with ADFA, the problem is simply disappearing. We do not need to know all that languages and we do not need to specify how the DFA will be built. There is no need to learn regular expressions or other formalism. All we have to do is to train an ADFA using the examples we need.

The code of the ADFA is just one, it can be (pre)compiled and shipped as object code, if needed.

14.Conclusions

The adaptive automata can be implemented using various languages both imperative or functional. We have tried: Oberon-2, C++, Haskell.
The theory and technology may have multiple appliances: video alarm systems [Sme 09], anti-virus products, automatic observers, music synthesis and recognition, voice identification systems...and maybe more.

15.Present and the next step

The main iresearch is actually focus on testing the limits of adaptive automata and decid how kind of applications are suitable for them. There are some things well established:

ADFA may be used as part of compilers and interpreters, also for DSL's production. In fact this was the starting point of our research: to avoid the usual specifications needed by a lexer generator.

There is also a dissertation by Smeu Florin [Sme 09], one of our students, which had implemented ADFA using the C++ language on the Linux platform and used them in order to built alarms triggered by image changes. The images was provided by a web-cam via the video4linux interface . The experiment was a success and the student won a prize for that work. The ADFA was trained using a static image of the guarded area. An other student, Bălăiță Constantin [Bal 08] had tested the ability of ADFA as being used as engine for anti-spam filters. He also succeeds. Anti spam filters are also a good application of ADFA.

Other applications: some of our students was requested to try to apply the ADFAs in other projects, included here being the music recognition and music synthesis. We do not have their results available here, now.

16. References

[Aab-96] Aaby, A. Anthony; *Haskell Tutorial* http://www.cs.wwc.edu/~cs_dept/KU/PR/Haskell.html

[Aab-05] Aaby, Anthony; Popa, Dan; *Construcţia compilatoarelor folosind Flex şi Bison*, Edusoft, Bacău, 2005

[Arm01] Armour Philip: *The business and software: Zeppelins and jet planes: a methaphor for modern software projects*. Comm. Of ACM, 44(10):13-15 Oct.2001

[Aho07] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, *Compilers Principles, Techniques, & Tools*, sec.ed.2007, Pearson Education (chap 3, pp 109-189)

[Bal 08] Bălăiță, Constantin; *Aplicații ale automatelor adaptive la realizarea sistemelor anti-spam (Applications of the ADFA at the anti-spam filters building),* disertation, Univ. din Bacău, 2008

[Pey-02] Peyton Jones, Simon (editor); *Haskell 98 Languages and Libraries – The revised Report*, Cambridge Univ., Sept.2002

[Pop04] Popa Dan; *Adaptable Tokenizer for Programming Languages* , Simpozionul International al Tinerilor Cercetatori, ASEM, Chisinau 2004, pg 55-57, ISBN 9975-75-239-x

[Pop05] Popa Dan ; *Adaptive DFA based on array of sets*, Studii si Cercetari Ştiinţifice, Seria Matematica, Nr 15 (2005) p 113-121, ISSN 1224 - 2519

[Sme 09] Smeu Florin: *Sistem de supraveghere video bazat pe automat adaptiv. (Video surveillancee system based on adaptive automata)*, dissertation, Univ. of Bacău , 2009 http://stiinte.ub.ro/cercetare/c-conferinte/106/327