Programming Techniques in GCLA II

Göran Falkman & Olof Torgersson

Department of Computer Sciences
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
{falkman,oloft}@cs.chalmers.se

Abstract

This paper presents work on methodologies for the programming tool GCLA II[1]. Two methods are investigated for reasoning about properties of regular expressions and their corresponding nondeterministic finite automata (NFA's). The methods described are combination of predicate and functional programming within the framework of GCLA II and an investigation of properties and possible usages of the duality of the "," (and) construct within the language.

1 Introduction

This paper is the first result of a project currently carried out at the Department of Computer Sciences under the supervision of Lars Hallnäs. Within this project we intend to study the problem of giving mathematically precise and implementation independent models of the notion of a knowledge base (KB). These models will then serve as a basis for computer aided development and testing of specialized knowledge based systems (KBS). We will especially concentrate on trying to characterize the logical interface of a KB, i.e. a user's basic conceptual model of a given KB. We will in particular look at applications in the field of diagnosis/ error detection.

An example of such an application is environments for program development, especially debugging. Our aim is to investigate how knowledge based systems (KBS) technology can be applied to such applications, using the programming tool GCLA II. The language GCLA II was originally designed for applications within KBS such as default and hypothetical reasoning, diagnosis and simulation.

The examples in this paper are taken from the area of lexical analysis, since this seems to be a necessary first step.

This paper contributes to the as yet poorly known domain of programming methodology for GCLA II. Two methods are investigated for reasoning about properties of regular expressions and their corresponding nondeterministic finite automata (NFA's). The methods described are combination of predicate and functional programming within the framework of GCLA II and an investigation of properties and possible usages of the duality of the "," (and) construct within the language.

2 GCLA II

2.1 Introduction to GCLA II

GCLA[2] II (*G*eneralized Horn *C*lause *LA*nguage) is perhaps best described as a logical programming language, with some properties usually found among functional languages.

Compared to Prolog, what has been added to GCLA II is the possibility to assume conditions. For example, the clause

```
a <= (b -> c).
```

1. This work was carried out as part of the work in the ESPRIT working group GENTZEN.
2. To be pronounced "gisela".

should be read as “a holds if c can be proved while assuming b”.

There is also a richer set of queries in GCLA II than in Prolog. An ordinary Prolog query is written

```
:- a.
```

and should be read as “Does a hold (in the definition *D*)?” In GCLA II one can also assume things in the query, for example

```
c \- a.
```

which should be read as “Assuming c, does a hold (in the definition *D*)?”, or “Is a derivable from c?”

To execute a program, a query *G* is posed to the system, asking whether there is a substitution σ such that $G\sigma$ holds according to the logic defined by the program. The goal *G* has the form $\Gamma \vdash$ c, where Γ is a list of assumptions, and c is the conclusion from the assumptions Γ. The system tries to construct a deduction showing that $G\sigma$ holds in the given logic.

For a more complete and comprehensive description of GCLA II’s theoretical properties see [1]. [2] demonstrates the program development methodology used in GCLA II. Various implementation techniques, including functional and object oriented programming, are also demonstrated.

2.2 Programs in GCLA II

A GCLA II program consists of two parts; One part is used to express the declarative content of the program, called the *definition* or the *object level*, and the other part is used to express rules and strategies acting on the declarative part, called the *rule definition* or the *meta level*.

2.2.1 The definition

The definition constitutes the formalization of a specific problem domain, and in general contains a minimum of control information. The intention is that the definition by itself gives a purely declarative description of the problem domain while a procedural interpretation of the definition is obtained only by putting it in the context of the rule definition.

2.2.2 The rule definition

The rule definition contains the procedural knowledge of the domain, i.e., the knowledge used for drawing conclusions based on the declarative knowledge in the definition. This procedural knowledge defines the possible inferences made from the declarative knowledge.

The rule definition contains *inference rule* definitions, which define how different inference rules should act, and *strategies* which control the search among the inference rules.

2.2.3 Example: Default reasoning

Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as well as all penguins, and finally we know that Pengo is a penguin. This knowledge is expressed in the following definition:

```
flies(X) <=
  bird(X),
  (penguin(X) -> false).

bird(tweety).
bird(polly).
bird(X) <=
  penguin(X).

penguin(pengo).
```

To be able to use this definition the way we want we have to write the following rule definition:

```
fs <=                                       % Never do axiom!
  right(fs),                                % First try standard right rules.
  left_if_false(fs).                        % else if consequent is false...
```

```
    left_if_false(PT) <=                     % Is the consequent false?
      (_ \- false).
    left_if_false(PT) <=                     % If so perform left rules.
      no_false_assump(PT),
      false_left(_).

    no_false_assump(PT) <=                   % No false assumption.
      not(member(false,A)) ->                % i.e. the term false is not a
      (A \- _).                              % member of the assumption list.
    no_false_assump(PT) <=
      left(PT).

    member(X,[X|_]).                         % Proviso definition.
    member(X,[_|R]):-
      member(X,R).
```

If we want to know which birds can fly, we pose the query

```
    fs \\- (\- flies(X)).
```

and the system will respond with `X = tweety` and `X = polly`.

If we want to know which birds cannot fly, we can pose the query

```
    fs \\- (flies(X) \- false).
```

and the system will respond with `X = pengo`.

3 Some background material

In the following subsections we will give a short introduction to *nondeterministic finite automata* and *Thompson's construction*. For a more complete description of these subjects we refer to [3].

3.1 Nondeterministic finite automata

Definition: A *nondeterministic finite automaton* (NFA) is a mathematical model that consists of:

1. a set of states S
2. a set of input symbols Σ
3. a transition function *move* that maps state-symbol pairs to sets of state
4. a state s_0 that is distinguished as the *start state*
5. a set of states F distinguished as *accepting states*

An NFA can be represented diagrammatically by a labelled directed graph, called a transition graph, in which the nodes are the states and the labelled edges represent the transition function.

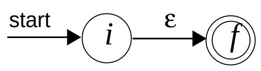
An NFA accepts an input string s if and only if there is some path in the transition graph corresponding to the NFA from the start state to some accepting state, such that the edge labels along this path spell out s. The *language defined* by an NFA is the set of input strings that it accepts.

3.2 Thompson's construction

Given a regular expression r, we first parse r into its constituent subexpressions. Then using rule (1) and (2) below, we construct NFA's for each of the basic symbols in r (those that are either ε or an alphabet symbol). Then, guided by the syntactic structure of the regular expression r, we combine these NFA's inductively using rule (3) below until we obtain the NFA for the entire expression.

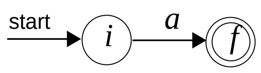
Each intermediate NFA produced during the course of the construction corresponds to a subexpression of r and has several important properties: it has exactly one final state, no edges enters the start state, and no edge leaves the final state.

1. For the regular expression ε, construct the NFA



Here *i* is a new start state and *f* a new accepting state.

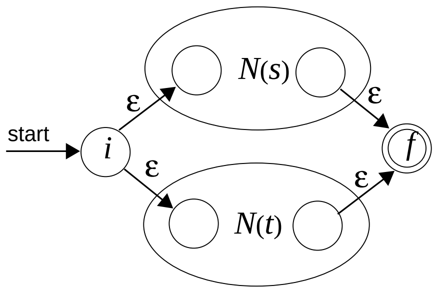
2. For the regular expression *a*, construct the NFA



Again *i* is a new start state and *f* a new accepting state.

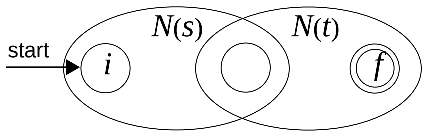
3. Suppose *N*(*s*) and *N*(*t*) are NFA's for the regular expressions *s* and *t* respectively.

   a) For the regular expression *s*|*t*, construct the composite NFA



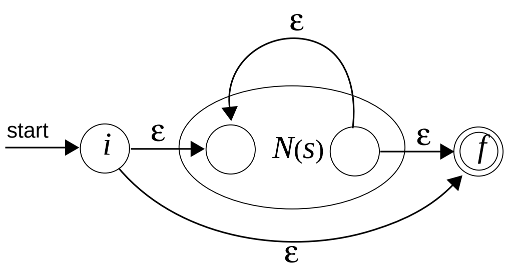
Here *i* is a new start state and *f* a new accepting state.

   b) For the regular expression *st*, construct the composite NFA



The start state of *N*(*s*) becomes the start state of the composite NFA and the accepting state of *N*(*t*) becomes the accepting state of the composite NFA. The accepting state of *N*(*s*) is merged with the start state of *N*(*t*).

   c) For the regular expression *s**, construct the composite NFA



Here *i* is a new start state and *f* a new accepting state.

   d) For the parenthesized regular expression (*s*), use *N*(*s*) itself as the NFA.

   e) For the regular expression s^+, use the NFA of the expression *ss**.

   f) For the regular expression *s*?, use the NFA of the expression ε|*s*.

4 Mixing functions and predicates in programs

This section describes how we simulate an NFA in GCLA II using the possibility to mix functions and predicates in a program.

4.1 A functional version of Thompson's construction

The algorithm described in Section 3.2 for constructing NFA's is easily changed into a function that takes a regular expression as its argument and returns the corresponding NFA generated by Thompson's construction. An NFA is represented by a 3-tuple consisting of a list, containing all the transitions of the NFA, its start state and its accepting state. When we use a function in a query we put the function call to the left of the turnstile, '`\-`', and get the result to the right. Using the strategy `thomp/0,` which assures that we will only get the answers we are interested in, we get the NFA corresponding to a regular expression like this:

```
thomp \\- thompson("a":"b") \- NFA.
NFA = nfa([m(0,"a",1),m(1,"b",2)],0,2)
```

The query above should be read: "What is the NFA corresponding to the regular expression *ab*?". We could also instantiate the result in which case the query should be read: "Is `nfa([m(0,"a"],1),m(1,"b",2)],0,2)` the NFA given by Thompson's construction for the regular expression *ab*.

The definition of `thompson/1` is simply a call to `thompson/2`, which has one clause for each kind of regular expression. We show how some of these should be read and then give the entire definition of `thompson/2.`

```
thompson(eps,S) <=
  (S+1 -> S1) -> nfa([m(S,eps,S1)],S,S1).
```

The clause above, which gives the NFA for the empty string, should be read: "If `S+1` evaluates to `S1` then `thompson(1,eps,S)` evaluates to `nfa([m(S,eps,S1)],S,S1)`."

```
thompson((A:B),S) <=
  (thompson(A,S) -> nfa(M1,S,T2)),
  (thompson(B,T2) -> nfa(M2,T2,T)),
  append(M1,M2,M3)
  -> nfa(M3,S,T).
```

This clause gives the NFA for the regular expressions *ab*. Here we have mixed functional and predicate programming, whereas `append/3` is a predicate that appends its first and second argument in the third. One way to read this clause is: "If `thompson(A,S)` evaluates to the NFA `nfa(M1,S,T2)`, and `thompson(B,T2)` evaluates to `nfa(M2,T2,T)`, and `M1` and `M2` are appended in `M3` then `thompson((A:B),S)` evaluates to the NFA `nfa(M3,S,T)`."

In the definition below, each clause corresponds to the different cases in the description of Thompson's construction given in Section 3.2.

```
thompson(eps,S) <=
  (S+1 -> S1) -> nfa([m(S,eps,S1)],S,S1).

thompson(A,S)
#{A\=(_|_),A\=(_:_),A\=eps,A\='?'(_),A\='+'(_),A\='*'(_),A\= reg(_,_)} <=
  (S+1 -> S1) -> nfa([m(S,A,S1)],S,S1).

thompson((A|B),S) <=
  (S+1 -> S1),(thompson(A,S1) -> nfa(M1,S1,T2)),
  (T2+1 -> ST2),(thompson(B,ST2) -> nfa(M2,ST2,T)
  (T+1 -> T1),
  append(M1,M2,M3)
  -> nfa([m(S,eps,S1),m(S,eps,ST2),m(T2,eps,T1),m(T,eps,T1)|M3],S,T1).

thompson((A:B),S) <=
  (thompson(A,S) -> nfa(M1,S,T2)),
  (thompson(B,T2) -> nfa(M2,T2,T)),
  append(M1,M2,M3)
  -> nfa(M3,S,T).

thompson((A)*,S) <=
  (S+1 -> S1),(thompson(A,S1) -> nfa(M1,S1,T)),
  (T+1 -> T1)
  -> nfa([m(S,eps,T1),m(S,eps,S1),m(T,eps,T1),m(T,eps,S1)|M1],S,T1).

thompson((A)+,S) <=
  thompson((A:A*),S).
thompson((A)?,S) <=
  thompson((A|eps),S).

thompson(reg(X,E),S) <=
  reg(X,E) -> thompson(E,S).
```

4.2 Running an NFA given by Thompson's construction

To be able to determinate whether a certain string is recognized by a given NFA or not, we need to simulate the NFA. For this purpose we use Algorithm 3.4 in [3].

An example of a typical query is:

```
sim \\- input("ab") \- simula(nfa([m(0,"a",1),m(1,"b",2)],0,2)).
```

This should be read: "Does the given NFA recognise the string `"ab"`?", and will give the answer `yes`. When writing a definition in GCLA II you have to decide whether you want to use the definition on the left or the right side of the sequent or on both. When we wrote the definition of `simula/1` we decided that it should be used to the right, and that we did not want to give the input string as a parameter to the simulation algorithm. Instead we use the left hand as a kind of storage and read characters from it when needed.

The simulation of an NFA uses two operations *move* and ε-*closure*. If *S* is a set of states then *move(S,a)* gives all states that can be reached from a state in *S* by a transition on *a*, the current input character. ε-*closure(S)* gives all the states that can be reached from a state in *S* by zero or more ε-transitions.

We describe Algorithm 3.4 by showing its definition in GCLA II:

```
simula(nfa(Transes,S0,Final)) <=
  simula(eclosure(stack([S0]),[S0],Transes),Transes,Final).
```

The first thing to do is to compute the ε-*closure* of the start state `S0`, which is done by a call to the function `eclosure/3`, execution then continues with a call to `simula/3:`

```
simula(S,Transes,Final) <=
  next_char(A,(eq(A,[]),
  member(Final,S))).
```

The above could be read: "If the next input character `A` is the empty string, then see if the final state, `Final`, is in `S`, the current set of states". If this is the case then the NFA accepts the given string.

The definition below tells us what to do if the next input character is not the empty string. First, all the states that can be reached by transitions on the current input character `A` are computed by `move/4` and then the ε-*closure* of these are computed and execution continues with a call to `simula/3`.

```
simula(S,Transes,Final)<=
  next_char(A,(neq(A,[]),
  move(S,[A],Transes,Moves),
  simula(eclosure(stack(Moves),Moves,Transes),Transes,Final))).
```

The entire definition of `simula/1` and `simula/3` is

```
simula(nfa(Transes,S0,Final)) <=
  simula(eclosure(stack([S0]),[S0],Transes),Transes,Final).

simula(E) #{E \= nfa(_,_,_)}<=
  (E -> NFA),
  simula(NFA).

simula(S,Transes,Final)#{S \= [], S \= [_|_]} <=
  (S -> S1),
  simula(S1,Transes,Final).

simula(S,Transes,Final) <=
  next_char(A,(eq(A,[]),
  member(Final,S))).

simula(S,Transes,Final)<=
  next_char(A,(neq(A,[]),
  move(S,[A],Transes,Moves),
  simula(eclosure(stack(Moves),Moves,Transes),Transes,Final))).
```

4.3 Some queries

Does the given NFA accept the input string?

```
sim \\- input("aaaaa") \-
  simula(nfa([m(0,eps,3),m(0,eps,1),m(2,eps,3),
  m(2,eps,1),m(1,"a"],2)],0,3)).
```

```
yes
```

The NFA in the example accepts *a**.

Another possibility is to give the regular expression and let the program construct the NFA thus:

```
sim \\- input("abaa") \- simula(thompson(("a":"b"):"a"*)).
```

```
yes
```

For regular expressions that are not on the form r_1:r_2, where r_1 is possibly infinite, it is also possible to generate strings generated by the regular expression:

```
sim \\- input(A) \- simula(thompson(("a":"b":"c")*)).
```

```
A = [];
A = [[]|_A];
A = "abc";
A = ["abc",[]|_A];
A = "abcabc";
...
```

Yet another way to use the knowledge of the system is to assume a number of regular expressions and check if any of these generates an NFA which accepts the input string.

```
sim \\- reg(_,"b"),reg(_,"a"),reg(_,("a")*),input("a") \-
  simula(thompson(reg(_,Exp))).
```

```
Exp = "a";
Exp = ("a"*);
no
```

One way to read this is: “Assuming that `reg(_,"a"), reg(_,"b")`and `reg(_,("a")*)` are regular expressions and that we have the input string `"a"`, is it then possible to construct an NFA that accepts the input string?”

We could also pose the query: “Is the input string `"a"` accepted by both the NFA’s generated by the regular expressions *aa** and *a*”.

```
sim \\- input("a") \- simula(thompson("a")),simula(thompson("a":("a")*)).
```

```
yes
```

Instead of giving the regular expressions as assumptions in the query we can put them in a definition, which will give us a lexical analyser for the defined regular expressions. With the definition

```
reg(digits,(reg(digit,_))*).
reg(digit,("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9")).
reg(relop,("="|"=":"<"|">")).
```

some possible queries are:

```
sim \\- input("23") \- simula(thompson(reg(N,_))).
```

```
N = digits;
no
```

```
sim \\- input("=")   \- simula(thompson(reg(N,_))).
```

```
N = relop;
no
```

4.4 About the rule definition

As mentioned before a GCLA II program consists of two parts the *definition* and the *rule definition.* To get the answers we want from the definitions shown above we have to write some new rules and strategies. For example we do not want to try the `d_right/2` rule on terms that we know have no definition such as numbers or lists. We therefore introduce a special class of terms, which we call *data*, and restrict the `d_right/2` rule so that it's not applicable to these.

```
data(X):- var(X).
data(X):- number(X).
data(X):- X == [].
data(X):- nonvar(X), functor(X,'.',2).
data(X):- functor(X,nfa,3).

d_right(C,PT) <=
  not(data(C)),
  atom(C),
  clause(C,B),
  (PT -> (A \- B)) -> (A \- C).
```

Another special construct is `nextchar/2`, which lacks definition since it's a so called *constructor* that has its own inference rule `next_right/1`

```
next_right(PT) <=
  hdt(String,H,T),
  (PT -> (I@[input(T)|R] \- C))
  -> (I@[input(String)|R] \- next_char(H,C)).
```

5 Using the duality of the comma constructor

In this section we will present another approach how to implement Thompson's construction in GCLA II. The idea behind this implementation is to use the duality of the comma-constructor in GCA II; to the left of the object level sequent the comma is read disjunctively and to the right of the object level sequent the comma is read conjunctively.

Suppose we want to pose queries like: “Find all regular expressions that contain the expression that corresponds to a given NFA as a subexpression” and “Find all subexpressions to the regular expression that corresponds to a given NFA”.

This can be accomplished by using just a small definition of Thompson's construction and the duality of the comma constructor. We will see that using the definition on the left side of the object level sequent corresponds to the first query above and using the definition on the right side of the object level sequent corresponds to the second query.

5.1 The definition

If we want to use the clauses of the definition on both sides, then we cannot use the definition of Thompson's construction given in the last section. The reason for this is that in the definition given in the last section many of the clauses were evaluated functionally, that is, they were using the arrow-construct. But the arrow-construct is not symmetric; the interpretation of the arrow-construct on the left side of the sequent differs from the interpretation of the arrow-construct on the right side of the sequent.

Instead we now let the different cases of Thompson's construction be defined by simply listing the transitions that the resulting NFA consists of. This means that we represent an NFA by just a comma-separated list of transitions. We do not bother listing the states, symbols, start state and accepting state of the NFA's since an NFA constructed by Thompson's construction is uniquely defined by its transition function. The states are denoted by successor-arithmetic and the numbering of the states always gives the start state the smallest number and the accepting state the largest number. For example, the NFA corresponding to the expression *a** is represented by

```
(m(0,eps,s(0)),m(s(s(0)),eps,s(s(s(0)))),m(s(0),a,s(s(0))),
  m(s(s(0)),eps,s(0)),m(0,eps,s(s(s(0)))))
```

Each clause in the definition corresponds directly to the step in Thompson's construction (see Section 3.2).
The complete definition is given by:

```
thompson(eps,S,s(S)) <=
  m(S,eps,s(S)).

thompson(A,S,s(S))
  #{A\=(_|_),A\=(_:_),A\='*'(_),A\='+'(_),A\='?'(_),A\=eps,A\=p(_)} <=
  m(S,A,s(S)).

thompson((A|B),S,s(T)) <=
  m(S,eps,s(S)),
  m(S,eps,s(T2)),
  m(T2,eps,s(T)),
  m(T,eps,s(T)),
  less_than(T2,s(T)),
  less_than(s(S),T2),
  thompson(A,s(S),T2),
  thompson(B,s(T2),T).

thompson((A:B),S,T) <=
  less_than(T2,T),
  less_than(S,T2),
  thompson(A,S,T2),
  thompson(B,T2,T).

thompson((A)*,S,s(T)) <=
  m(S,eps,s(S)),
  m(S,eps,s(T)),
  m(T,eps,s(T)),
  m(T,eps,s(S)),
  thompson(A,s(S),T).

thompson((A)+,S,T) <=
  thompson((A: A*),S,T).

thompson((A)?,S,T) <=
  thompson((A|eps),S,T).

thompson(p(E),S,T) <=
  thompson(E,S,T).

less_than(0,s(_)).
less_than(s(A),s(B)) <=
  less_than(A,B).
```

For example, the case when the regular expression is in the form of *a*|*b*, i.e., *a* or *b* is defined by

```
thompson((A|B),S,s(T)) <=
  m(S,eps,s(S)),
  m(S,eps,s(T2)),
  m(T2,eps,s(T)),
  m(T,eps,s(T)),
  less_than(T2,s(T)),
  less_than(s(S),T2),
  thompson(A,s(S),T2),
  thompson(B,s(T2),T).
```

If `m(`S_1, Sym, S_2`)` is read: "There is a transition from state S_1 to state S_2 on symbol Sym", then this definition can be read: "The NFA resulting when applying Thompson's construction to a regular expression in the form of *a*|*b* when starting in state `S` and ending in state `s(T)`, consists of an empty transition from state `S` to state `s(S)`, an empty transition from state `S` to state `s(T2)`, an empty transition from state `T2` to state `s(T)`, an empty transition from state `T` to state `s(T)`, the transitions of the NFA that results when applying Thompson's construction to *a*, starting in state `s(S)` and ending in state `T2` and the transitions of the NFA that results when applying Thompson's construction to *b*, starting in state `s(T2)` and ending in state `T`, where `T2` is a new state that lies between `S` and `s(T)`.

5.2 Some rules

We also need some rules and strategies to be able to use this definition the way we want. For example, we only want the `d_right/2` and `d_left/3` rules to be applicable to the `thompson/3` and `less_than/2` clauses. We also need a new `axiom/3` rule, since we only want the `axiom` to be applicable to variables and on `m/3` clauses. Since we do not want the `axiom/3` rule to be applicable when we are following a path in the search-tree that we know will not lead to a solution, we have to restrict the `axiom/3` rule to be applicable only to sequents where there are no non-evaluated `thompson/3` or `less_than/2` clauses and no `false/0` clauses. The new `axiom/3` rule becomes

```
axiom(A,C,I) <=
  \+functor(A,thompson,3),           % Not applicable to thompson/3.
  \+functor(C,thompson,3),
  \+functor(A,less_than,2),          % Not applicable to less_than/2.
  \+functor(C,less_than,2),
  \+functor(A,true,0),               % Not applicable to true/0.
  \+functor(C,true,0),
  \+member(thompson(_,_,_),I),       % No non-evaluated thompson/3
  \+member(thompson(_,_,_),R),       % in the sequent.
  \+member(less_than(_,_),I),        % No non-evaluated less_than/2
  \+member(less_than(_,_),R),        % in the sequent.
  \+member(false,I),                 % No false/0 in the sequent.
  \+member(false,R),
  term(C),
  term(A),
  unify(C,A) -> (I@[A|R] \- C).
```

5.3 Queries

There are two forms of queries that we can pose:

1. `thompson(`*E*`,`*S*`,`*T*`) \-` *m*$_1$`,`*m*$_2$`,...,`*m*$_n$`.`
   This query should be read: "Find all regular expressions *E* whose corresponding NFA (as given by Thompson's construction with start state *S* and accepting state *T*) contains *at least* the transitions *m*$_1$,*m*$_2$,...,*m*$_n$." If *m*$_1$,*m*$_2$,...,*m*$_n$ is the definition of an NFA and if this NFA corresponds to the regular expression *r*, this is equivalent to: "Find all regular expressions *E* which contain *r* as a subexpression."

2. *m*$_1$`,`*m*$_2$`,...,`*m*$_n$ `\- thompson(`*E*`,`*S*`,`*T*`).`
   This query should be read: "Find all regular expressions *E* whose corresponding NFA (as given by Thompson's construction with start state *S* and accepting state *T*) contains *at most* the transitions *m*$_1$,*m*$_2$,...,*m*$_n$." If *m*$_1$,*m*$_2$,...,*m*$_n$ is the definition of an NFA and if this NFA corresponds to the regular expression *r*, this is equivalent to: "Find all regular expressions *E* which are a subexpression of *r*."

5.3.1 Using the definition to the left

If we want to know which NFA that results when applying Thompson's construction to the regular expression *a*|*b* we can pose the query

```
gclar \\- thompson((a|b),0,_) \- X.
```

and the system will respond with

```
X = m(0,eps,s(0));
X = m(0,eps,s(s(0))));
X = m(s(s(0)),eps,s(s(s(s(0))))));
X = m(s(s(s(0)))),eps,s(s(s(s(0))))));
X = m(s(0),a,s(s(0));
X = m(s(s(s(0))),b,s(s(s(s(0))))));
no
```

that is, all transitions in the NFA corresponding to the expression *a*|*b*.

Another query is

```
gclar \\- thompson((a*|b:c),0,_) \- m(_,eps,T),m(T,a,_).
```

It should be read: “Is there an empty transition to state `T` from any state and a transition from state `T` to any state on symbol `a`, in the NFA corresponding to *a**|*bc*? The system will answer twice with `T = s(s(0))` since there are two such empty transitions, one from state 1 and one from state 3.

The NFA that corresponds to the expression *ab* is given by the transitions: `m(0,a,s(0))` and `m(s(0),b,s(s(0)))`. If we pose the query

```
gclar \\- thompson(E,0,s(s(0))) \- m(0,a,s(0)),m(s(0),b,s(s(0))).
```

the system will respond with the single answer `E = (a:b)` since that is the only expression whose corresponding NFA contains exactly those transitions. If we instead pose the query

```
gclar \\- thompson(E,0,s(s(0))) \- m(_,a,_),m(_,b,_).
```

we will get the answers `E = (a:b)` and `E = (b:a)` since these are the only expressions whose corresponding NFA’s start in state `0`, end in state `s(s(0))` and have transitions on *at least* symbol *a* and symbol *b*.

If we do not want to restrict ourselves to finding expressions whose corresponding NFA’s are of a certain length, we can pose the following query

```
less_than(T,s(s(s(s(0)))),thompson(E,0,T) \- m(_,a,_),m(_,b,_).
```

which should be read: “Find all expressions `E` whose corresponding NFA’s are of length less than 4 and contains transitions on *at least* symbol *a* and symbol *b*. The system will respond with

```
E = (a:b)
T = s(s(0));

E = (b:a)
T = s(s(0));

E = (eps:(a:b))
T = s(s(s(0)));

E = (eps:(b:a))
T = s(s(s(0)));

...
```

If we instead we instead pose the query

```
less_than(T,s(s(s(s(0)))),thompson(E,0,T) \- m(_,a,N),m(N,b,_).
```

which should be read: “Find all expressions `E` whose corresponding NFA’s are of length less than 4 and of which *ab* is a subexpression.” We will then get the answers

```
E = (a:b)
N = s(0)
T = s(s(0));

E = (eps:(a:b))
N = s(s(0))
T = s(s(s(0)));

...
```

5.3.2 Using the definition to the right

If we for some reason want to find all expressions `E` whose corresponding NFA’s consist only of one transition, we can pose the query

```
m(_,X,_) \- thompson(E,0,_).
```

and the system will respond with

```
X = eps
E = eps;

X = X
E = _X;

no
```

that is, `E` is either `eps` or a variable.

The query

```
m(_,a,_),m(_,b,_) \- less_than(T,s(s(s(0)))),thompson(E,0,T).
```

should be read: “Find all expressions `E` whose corresponding NFA’s are of length less than 3 and contains transitions on *at most* symbol *a* and symbol *b*. We will get the following answers

```
E = a
T = s(0);

E = b
T = s(0);

E = (a:b)
T = s(s(0));

E = (b:a)
T = s(s(0));

no
```

If we instead pose the query

```
m(_,a,N),m(N,b,_) \- less_than(T,s(s(s(0)))),thompson(E,0,T).
```

we will get the answers

```
E = a
N = s(0)
T = s(0);

E = b
N = 0
T = s(0);

E = (a:b)
N = s(0)
T = s(s(0));

no
```

that is, we get all subexpressions of *ab*.

References

[1] P. Kreuger, “GCLA II, A Definitional Approach to Control”, Extensions of Logic Programming: Proceedings of a workshop held at SICS, February 1991, Springer Lecture Notes in Artificial Intelligence.

[2] M. Aronsson, “Methodology and Programming Techniques in GCLA II”, Research Report SICS R9205, Swedish Institute of Computer Science, 1992.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, “Compilers: Principles, Techniques, Tools”, Addison-Wesley Publishing Company, Reading, Mass., 1986.