ELSEVIER

Theoretical Computer Science 266 (2001) 59–79

Theoretical Computer Science

www.elsevier.com/locate/tcs

The accepting power of unary string logic programs

Tatsuru Matsushita[a,*], Colin Runciman[b]

[a] *Department of Applied Statistics, The University of Reading, MPS Research Unit, P.O. Box 240, Early Gate, Reading RG6 6FN, UK*

[b] *Department of Computer Science, University of York, York, Y01 5DD, UK*

Received April 1999; revised November 1999

Communicated by M. Nivat

Abstract

The set of programs written in a small subset of pure Prolog called US is shown to accept exactly the class of regular languages. The language US contains only unary predicates and unary function symbols. Also, a subset of US called RUS is shown to be equivalent to US in its ability in accepting the class of regular languages. Every clause in RUS contains at most one function symbol in the head and at most one literal with no function symbol in the body. The result is very close to a theorem of Matos (TCS April 1997) but our proof is quite different. Though US and RUS have the same accepting power, their conciseness of expression is dramatically different: if we try to write an RUS program equivalent to a US program, the number of predicates in the RUS program could be $O(2^{2^N})$ where N is the sum of the number of predicates and the number of functors in the US program. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Expressive power; Prolog; Program transformation; Regular language; Finite-state acceptor

1. Introduction

Consider a logic programming language containing only a single constant, unary functors, and unary predicates. We call such a language "Unary-String" logic language (US) because we can represent arbitrary finite strings on an alphabet by using the set of functors as our alphabet and the single constant as the terminator of the strings.

In this paper we examine how powerful US programs are as acceptors of formal languages. Since there are only unary functors in a US program P, a Herbrand model [9] of P is a set of ground terms, each of which contains a string of functors (terminated

* Corresponding author.
E-mail addresses: t.matsushita@reading.ac.uk (T. Matsushita), colin@cs.york.ac.uk (C. Runciman).

by a constant). Then any predicate q in P defines a set of strings (a language), for each word $f_1f_2\dots f_k$ of which the term $q(f_1f_2\dots f_k(0)))$ is in the model. Operationally, the word $f_1f_2\dots f_k$ belongs to the language defined by q if the goal $\leftarrow q(f_1f_2\dots f_k(0))$ has a refutation w.r.t. P. So a US program works as a machine recognising languages.

For a predicate q defined by several clauses, the set of ground terms satisfying q in the model can be thought of as a union of sets of ground terms each of which is associated with one of the defining clauses. Conjunctive bodies indicate at least the power to express intersection of languages. Recursively defined predicates suggest at least the power of Kleene closure of concatenation. Together these observations suggest that at least the class of regular languages can be recognised by US programs. However, we can introduce extra function symbols in a clause body like an input stack, and we can make recursion at any position in a clause body. These observations suggest that US might accept something more complex than regular sets, but the main result of this paper is a proof that:

Proposition. *US programs accept exactly the class of regular languages.*

In other words, the expressive power of US in accepting languages coincides with the class of regular languages. A similar result has been published by Matos [6] but our approach here is quite different. Further comparisons with Matos' result will be given in Section 5.

Since any regular language can be specified by a finite automaton, in order to prove the conjecture, we show a one-to-one mapping between the set of finite automata and the set of US programs. An automaton A is *equivalent* to a logic program P if refutation of a ground goal w.r.t. P implies that the path associated with functors in the goal leads A to a final state, and vice versa.

The proof consists of two parts. Firstly, in Section 3 we show that any deterministic finite automaton (DFA) can be converted into an equivalent US program. In fact, we convert DFA's into programs written in a subset of US called the "Regular Unary String" language (RUS). Because of a strong similarity between transition rules of DFA's and RUS clauses, this part is relatively easy. Secondly, in Section 4 we prove that any US program can be converted to an equivalent non-deterministic finite automaton (NFA) with ε-moves. Since we have to cope with prefixes in clause bodies and conjunction of body literals, the second part is much harder than the first one. From the two translation results, we conclude that US programs accept exactly the class of regular languages.

In Section 2 the small logic languages US and RUS are explained. Also brief definitions of finite automata are given, partly for introducing some notational conventions. Section 3 defines a conversion scheme from an arbitrary DFA to an RUS program and proves the equivalence. The conversion scheme from US programs to NFA's is presented and proved in Section 4. Finally, Section 5 discusses related work, then concludes.

$$
\begin{array}{lll}
P^{\mathrm{us}} & = & \mathit{clause} \\
 & \mid & \mathit{clause}\ P^{\mathrm{us}} \\
\mathit{clause} & = & \mathit{head}. \\
 & \mid & \mathit{head} \leftarrow \mathit{body}. \\
\mathit{head} & = & \mathit{atom} \\
\mathit{body} & = & \mathit{atom} \\
 & \mid & \mathit{atom}, \mathit{body} \\
\mathit{atom} & = & \mathit{name}\ (\mathit{term}) \\
\mathit{term} & = & 0 \\
 & \mid & X \\
 & \mid & \mathit{name}\ (\mathit{term}) \\
\mathit{goal} & = & \leftarrow \mathit{atom}
\end{array}
$$

Fig. 1. Syntax of US.

2. Preliminaries

This section defines the two logic languages US and RUS, and introduces notations for derivations and refutations of logic programs, used in the later sections. Also the section includes brief definitions of DFA's and NFA's with ε-moves, and a few notations for graph representations of the automata.

In many cases we use the terminology in [5] for logic programming. For automata theory, we adopt notations in [4].

2.1. US

Unary String logic language (US) is a language of *definite logic programs* [8] containing only one constant 0 which is the terminator of strings of the object language, only one variable X which represents suffixes of strings, countably many unary functors, and countably many unary predicates. The syntax of US is given in Fig. 1.

The language does not contain negations nor any extra-logical predicates. Extra-logical predicates such as *read*, *write*, *assert*, *retract* impede logical reading of programs, on which our conjecture relies. Negation might be understood in connection with complementation of regular sets, but we have left it for future work.

Furthermore, we assume that the constant 0 may appear only in unit clauses. Ground body literals might work as guards before computing the intersection of the other non-ground goals, but do not seem to provide especially interesting consequences, and we exclude them.

Thus, examples of legal clauses of US include

$$p(f(0)).$$
$$p(f(g(X))).$$
$$p(f(g(X))) \leftarrow q(g(X)), r(h(k(X))).$$

$$
\begin{array}{lcl}
P^{\text{rus}} & = & \textit{clause} \\
 & | & \textit{clause } P^{\text{rus}} \\
\textit{clause} & = & \textit{head}. \\
 & | & \textit{head} \leftarrow \textit{body}. \\
\textit{head} & = & \textit{name } (\textit{term}) \\
\textit{term} & = & 0 \\
 & | & X \\
 & | & \textit{name } (X) \\
\textit{body} & = & \textit{name } (X)
\end{array}
$$

Fig. 2. Syntax of RUS.

but not

$$p(f(g(X))) \leftarrow q(g(0)), r(h(k(X))).$$

An atom of US is denoted $p(f_1 \ldots f_n(\omega)\ldots)$, or $p(\bar{f}(\omega))$, where ω is either 0 or X. Clauses are denoted by symbols $C_0, C_1, \ldots$, and symbols $G_0, G_1, \ldots$ denote goals. A derivation of a goal G_0 w.r.t. a program P is a finite or infinite sequence of goals

$$G_0 \xrightarrow{C_1} G_1 \xrightarrow{C_2} G_2 \rightarrow \cdots,$$

where C_i's are clauses in P and G_i is derived from G_{i-1} and C_i by one step resolution.

A refutation is a finite derivation of which the last goal is empty:

$$G_0 \xrightarrow{C_1} G_1 \xrightarrow{C_2} G_2 \rightarrow \cdots \xrightarrow{C_n} \square$$

2.2. RUS

Regular unary string (RUS) logic language is a subset of US in which

- every clause has at most one atom in its body,
- every unit clause has no functor symbol,
- every non-unit clause has exactly one functor symbol in its head, and
- there is no functor symbol in any clause body.

The syntax of RUS is given in Fig. 2. The language is quite restrictive. Every RUS clause must have one of the following forms:

$$
\begin{array}{l}
p(0). \\
p(X). \\
p(g(X)) \leftarrow q(X).
\end{array}
$$

So

$$
\begin{array}{l}
p(f(g(0))). \\
p(X) \leftarrow q(f(X)).
\end{array}
$$

are not allowed in RUS.

2.3. Finite-state automata

A deterministic finite automaton A is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is a transition function of type $Q \times \Sigma \rightarrow Q$.

Each finite automaton can be depicted as a graph, each node of which corresponds to a state and each edge from a state p to a state q annotated by a symbol f corresponds to a transition $(p, f) \rightarrow q$. We use graph notation

$$p \xrightarrow{f} q$$

to represent the transition $(p, f) \rightarrow q$. In order to represent a sequence of transitions (or a path) from p to q associated with a string $\bar{f} = f_1 \ldots f_n$, we write

$$\begin{aligned} & p \xrightarrow{f_1} \circ \rightarrow \cdots \rightarrow \circ \xrightarrow{f_n} q \\ \text{or } & p \xrightarrow{f_1 \ldots f_n} q \\ \text{or } & p \xrightarrow{\bar{f}} q. \end{aligned}$$

A finite automaton accepts a string $f_1 \ldots f_n$ if the sequence of transitions from the initial state q_0 associated with the string leads to a final state of the automaton. We use a star superscript to indicate a final state:

$$q_0 \xrightarrow{f_1} \circ \rightarrow \cdots \rightarrow \circ \xrightarrow{f_n} q^*$$

or if the name of the final state is not relevant we use $\odot$:

$$q_0 \xrightarrow{f_1} \circ \rightarrow \cdots \rightarrow \circ \xrightarrow{f_n} \odot$$

NFA with ε-moves: An automaton is *non-deterministic* when some of its states have more than one transition for the same input symbol. The type of transition function δ becomes $Q \times \Sigma \rightarrow 2^Q$, where 2^Q denoted the power set of Q.

An *empty transition* is a transition which does not consume any input symbol, denoted as $p \xrightarrow{\varepsilon} q$. The transition function δ of an NFA with ε-moves has type: $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$.

It is well known that any non-deterministic finite automaton with empty transitions can be converted into an equivalent deterministic finite automaton [4].

State transition system (STS): In order to make a clear matching between logic programs and recognising machines, occasionally we use a term: *state transition system* (STS). An STS is an automaton without initial state: i.e. for any FA (deterministic or non-deterministic) $A = (Q, \Sigma, \delta, q_0, F)$, we define an STS $S = (Q, \Sigma, \delta, F)$. We can define many different automata based on S by choosing a specific state in Q as an initial state.

In the subsequent arguments, strictly speaking, logic programs correspond to STS and a pair of a logic program and a predicate name corresponds to a pair of STS and an initial state, namely an automaton.

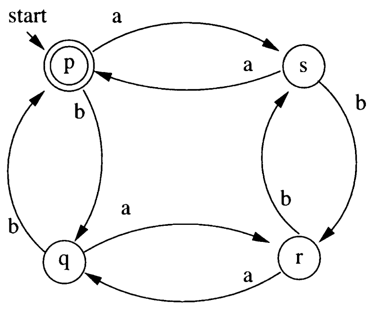

Fig. 3. A DFA A accepting strings containing an even number of a's and b's.

3. Deriving an RUS program from an FSA

This section considers translation of finite automata into Regular Unary String logic programs. The Section 3.1 describes how to construct an RUS program from an arbitrary finite automaton. Then in Section 3.2 we prove the equivalence between the original DFA and the derived RUS program.

3.1. Derivation of RUS program

From a finite automaton $A=(Q,\Sigma,\delta,q_0,F)$ we construct an RUS program P^{rus} as follows:

1. For each state $q \in Q$, introduce a unary predicate q.
2. For each symbol in alphabet $f \in \Sigma$, introduce a unary functor f.
3. For each transition rule $(p,f) \to q$, we add a clause $p(f(X)) \leftarrow q(X)$ to P^{rus}.
4. For each final state $q_j \in F$, we add a unit clause $q_j(0)$ to P^{rus}.
5. The initial state q_0 specifies the predicate of initial goals. An input string $f_1 \ldots f_n$ given to A is represented by an initial goal $\leftarrow q_0(f_1 \ldots f_n(0)\ldots)$.

Example 1. The graph in Fig. 3 represents a DFA accepting strings on $\{a,b\}$ containing an even number of a's and an even number of b's, taken from [4]. The arrow pointing to the node p denotes the initial state. The state p is also the only final state of A. The program in Fig. 4 is the RUS program derived from A through the above procedure.

We claim that the derived program P^{rus} is *equivalent* to the finite automaton A, in the sense of the following theorem.

Theorem 1. *A goal* $\leftarrow q_0(f_1 \ldots f_n(0)\ldots)$ $(n \geqslant 0)$ *has a refutation w.r.t.* P^{rus} *if and only if A accepts the string* $f_1 \ldots f_n$ *from the initial state* q_0.

```
p(0).
p(a(X)) ← s(X).
p(b(X)) ← q(X).
q(a(X)) ← r(X).
q(b(X)) ← p(X).
r(a(X)) ← q(X).
r(b(X)) ← s(X).
s(a(X)) ← p(X).
s(b(X)) ← r(X).
```

Fig. 4. RUS program of A.

3.2. Verification of the derivation

Because there is a direct mapping from transition rules into clauses, the proof is straightforward.

Proof. *If*: Suppose the automaton A accepts a string $f_1 \dots f_n$ from the initial state q_0, i.e. there is a path: $q_0 \xrightarrow{f_1} \cdots \xrightarrow{f_n} q_n^*$ in the graph. Then the transition function δ contains n transitions $(q_j, f_{j+1}) \to q_{j+1}$ $(0 \leqslant j \leqslant n-1)$.

From Rules 3 and 4 in Section 3.1, the derived program must contain clauses:

$$C_j = q_j(f_{j+1}(X)) \leftarrow q_{j+1}(X) \quad (0 \leqslant j \leqslant n-1),$$
$$C_n = q_n(0)$$

and from Rule 5, the accepted string is represented by an initial goal:

$$G_0 = \leftarrow q_0(f_1 \dots f_n(0) \dots).$$

By applying clauses C_j $(0 \leqslant j \leqslant n)$ successively to the initial goal, we obtain a derivation

$$G_0 \xrightarrow{C_0} G_1 \xrightarrow{C_1} \cdots \xrightarrow{C_{n-1}} G_n \xrightarrow{C_n} \square$$

where

$$G_j = \leftarrow q_j(f_{j+1} \dots f_n(0)) \quad (0 \leqslant j \leqslant n-1)$$
$$G_n = \leftarrow q_n(0).$$

So the derivation is a refutation of the initial goal.

Only if: Suppose a goal $G_0 = \leftarrow q_0(f_1 \dots f_n(0) \dots)$ has a refutation w.r.t. the derived program, i.e. there is a refutation of G_0:

$$G_0 \xrightarrow{C_0} G_1 \xrightarrow{C_1} G_2 \xrightarrow{C_2} \cdots \to \square$$

We have to find an accepting path of $f_1 \dots f_n$ from q_0.

If $n = 0$ the initial goal is $\leftarrow q_0(0)$. In order to derive an empty goal, the program must contain either $q_0(X)$ or $q_0(0)$. However, since every unit clause in the derived program is constructed from Rule 4, the clause must be $q_0(0)$. Then q_0 is a final state of A and there is a path of length 0 (q_0^*) accepting the empty string.

If $n > 0$, because of the form of the initial goal and the general form of non-unit clauses in the derived program, the first clause in the refutation must be $C_0 = q_0(f_1(X)) \leftarrow q_1(X)$ for some predicate q_1. Thus, the automaton contains the states q_0, q_1 and a transition rule $(q_0, f_1) \to q_1$.

From G_0 and C_0 we derive the next goal $G_1 = \leftarrow q_1(f_2 \ldots f_n(0) \ldots)$. The same argument as for G_0 holds for G_1, and the automaton contains another state q_2 and another transition rule $(q_1 f_2) \to q_2$, and so on, inductively. In general the automaton contains transition rules $(q_j, f_{j+1} \to q_{j+1}$ for all $0 \leqslant j \leqslant n-1$. In order to refute the second last goal $\leftarrow q_n(0)$ the program contains a unit clause $q_n(0)$, and hence, q_n is a final state.

If we combine the transitions above, we can construct a path

$$q_0 \xrightarrow{f_1} q_1 \xrightarrow{f_2} q_2 \xrightarrow{f_3} \cdots \xrightarrow{f_n} q_n^*$$

So, the string $f_1 \ldots f_n$ from q_0 is accepted by the finite automaton. $\square$

4. Deriving an FSA from a US program

In the previous section we have shown that any DFA is equivalent to an RUS program which is also a US program since RUS is a proper subset of US. In this section we prove that any US program is equivalent to an NFA with ε-moves. Here we frequently use the term STS defined in Section 2.3.

In order to construct the finite automaton A equivalent to (a pair of a predicate name and) a US program P^{us}, Section 4.1 describes construction of a *basis* STS A^0 representing all clause heads in P^{us}. The STS A^0 contains all possible segments of paths containing no ε-move. In other words, any paths in the final automaton but not in A^0 contain at least one ε-move. Section 4.2 considers how to interpret body literals in non-unit clauses. In Section 4.3 the procedure to construct the automaton A is defined. Non-unit clauses are iteratively augmented into A^0. In order to accommodate conjunctive bodies, we calculate intersection of automata, each of which corresponds to a conjunct of the body. The construction is proved correct in Section 4.4.

4.1. Construction of A^0

Given a US program we construct the equivalent finite automaton A in two steps. In this subsection we focus attention on clause heads in the program, and construct a basis A^0. The subsequent two subsections extend A^0 to A.

The STS A^0 contains every ‘essential’ transition in A, in the sense that although we ignore all body literals in the program, every possible move in the final automaton A is a sequence of paths in A^0 glued together by ε-moves. Also, every additional state in A is a ‘collective shadow’ of states in A^0, as we shall see in Section 4.3.

An STS $A^0 = (Q, \Sigma, \delta, F)$ is constructed from P^{us} as follows:

- The input alphabet Σ is the set of all functors in P^{us}.
- For each predicate q defined in P^{us}, Q contains a unique state $\{[q]\}$. We call such a state with a single element of unit length a *predicate state*.
- For each clause head $q(h_1 \dots h_n(\omega)\dots)$ $(n \geqslant 1)$, Q contains n *derived* states $\{[q, h_1]\}$, $\{[q, h_1, h_2]\}, \dots, \{[q, h_1, h_2, \dots, h_n]\}$. We call the last state $\{[q, h_1, h_2, \dots, h_n]\}$ including the full sequence of functor symbols, the *bottom* state of the clause head.

  Different clause heads introduce different sequences of states, even if they have the same predicate names and sequences of functors. If two clauses bring about the same name, they are distinguished from each other by superscripts if necessary. For example, $\{[q^1, h_1, h_2]\}$ and $\{[q^2, h_1, h_2]\}$. We abbreviate $\{[q, h_1, h_2, \dots, h_k]\}$ to $\{[q, \bar{h}]\}$ when details of the h_i are not relevant.
- For each unit clause $q(h_1 \dots h_n(\omega)\dots)$. $n \geqslant 0$, the bottom state $\{[q, h_1, \dots, h_n]\}^*$ is a final state. We use a star superscript to indicate a final state.
- For each unit clause with a variable $q(h_1 \dots h_n(X)\dots)$. $n \geqslant 0$, the bottom state $\{[q, h_1, \dots, h_n]\}^*$ has a set of looping arcs, one for each symbol in Σ, i.e.

  $$\forall f_j \in \Sigma, \quad \{[q, h_1, h_2, \dots, h_n]\}^* \xrightarrow{f_j} \{[q, h_1, h_2, \dots, h_n]\}^*$$

  are transitions in δ.
- Finally, for each clause head $q(h_1 \dots h_n(X)\dots)$ $(n \geqslant 1)$,

  $$\begin{array}{lcl} \{[q]\} & \xrightarrow{h_1} & \{[q, h_1]\} \\ \{[q, h_1]\} & \xrightarrow{h_2} & \{[q, h_1, h_2]\} \\ & \vdots & \\ \{[q, h_1, \dots, h_{n-1}]\} & \xrightarrow{h_n} & \{[q, h_1, \dots, h_{n-1}, h_n]\} \end{array}$$

  are transitions in δ.

The STS A^0 is a set of linear sequences of states. Every derived state has exactly one in-coming move. Every derived state, except bottom states of unit clauses with variables, has at most one out-going move. The number of states is equal to the sum of the number of predicates defined in the program and the number of occurrences of functors in all clause heads of the program.

Example 2. Consider the program $P1$ in Fig. 5. The STS A^0 derived from $P1$ contains two predicate states, $\{[p]\}$ and $\{[q]\}$, and seven further derived states:

$$\{[p^1, a]\}^*, \{[p^2, a]\}, \{[p^2, a, b]\}, \{[p^3, a]\}, \{[q, b]\}, \{[q, b, c]\} \quad \text{and} \quad \{[q, b, c, d]\}.$$

There is just one final state $\{[p^1, a]\}^*$. The graph in Fig. 6 is A^0 of $P1$.

4.2. Interpretation of body literals

Now we consider subgoals in non-unit clauses. A clause $p(\bar{f}(X)) \leftarrow q(\bar{h}(X))$ can be read declaratively as: "If for some term t, $q(\bar{h}(t))$ is true, then $p(\bar{f}(t))$ is true as

$$\begin{array}{lcl}
p(a(0)). & & \\
p(a(b(X))) & \leftarrow & p(a(X)). \\
p(a(X)) & \leftarrow & q(b(c(X))). \\
q(b(c(d(X)))) & \leftarrow & p(a(b(X))).
\end{array}$$

Fig. 5. Program $P1$.

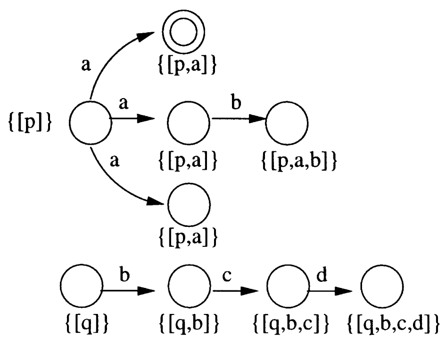

Fig. 6. A^0 of $P1$.

well." In order to convert the clause into a set of state transitions, we paraphrase "$p(t)$ is true" is "t leads the machine from p to a final state". Thus we re-write the above sentence as

> If a path $\bar{t}$ preceded by a path $\bar{h}$ leads the automaton from $\{[q]\}$ to a final state, then the same path $\bar{t}$ preceded by another path $\bar{f}$ leads the machine from $\{[p]\}$ to a final state.

The variable X in the body literal $q(\bar{h}(X))$ denotes a set of paths each of which begins from one of the *goal states* of the literal, defined below.

Definition 1 (*Goal states*). The *goal states* of a body literal $q(h_1 \ldots h_n(X)\ldots)$, denoted as $\mathscr{GS}\{q(h_1 \ldots h_n(X)\ldots)\}$, is a set of states *reachable* from the predicate state $\{[q]\}$ via successive moves associated with functors $h_1, \ldots, h_n$ in this order, possibly interspersed with ε-moves.

Thus, we connect the bottom state $\{[p, \bar{f}]\}$ of the clause head with each goal state in $\mathscr{GS}\{q(\bar{h}(X))\}$ by an ε-move. By this connection the string $\bar{f}$ is connected to the strings represented by X starting from each goal state in $\mathscr{GS}\{q(\bar{h}(X))\}$. Hence, if there is a path from any state of $\mathscr{GS}\{q(\bar{h}(X))\}$ to a final state, we obtain an accepting path from $\{[p]\}$.

Example 3. Consider the three clauses and a goal

$$p(a(X)) \leftarrow q(b(X)).$$
$$q(b(c(X))) \leftarrow r(d(X)).$$
$$r(d(e(0))).$$
$$\leftarrow p(X).$$

Given the program and the query, Prolog systems give the answer $X = a(c(e(0)))$.

In the derived automaton we connect the bottom state $\{[p,a]\}$ of the first clause by an ε-move to the state $\{[q,b]\}$. We similarly connect $\{[q,b,c]\}$ by an ε-move to $\{[r,d]\}$. We obtain

$$\begin{tikzcd}
\{[p]\} \arrow[r, "a"] & \{[p,a]\} \arrow[d, "\varepsilon"'] & & \\
\{[q]\} \arrow[r, "b"] & \{[q,b]\} \arrow[r, "c"] & \{[q,b,c]\} \arrow[d, "\varepsilon"'] & \\
& \{[r]\} \arrow[r, "d"] & \{[r,d]\} \arrow[r, "e"] & \{[r,d,e]\}^*.
\end{tikzcd}$$

The two downward arrows associated with ε represent two reversed implications in the first and the second clauses. The accepting path

$$\{[p]\} \xrightarrow{a} \{[p,a]\} \xrightarrow{\varepsilon} \{[q,b]\} \xrightarrow{c} \{[q,b,c]\} \xrightarrow{\varepsilon} \{[r,d]\} \xrightarrow{e} \{[r,d,e]\}^*$$

admits the string ace, corresponding to the above answer by Prolog systems.

In this example we can easily find the states to be connected to a bottom state. In general, however, a body literal may contain a longer list of functors than that of any clause head unifying with the literal. Suppose we replace the body predicate $q(b(X))$ in the first clause by a literal $q(b(c(e(X))))$. It is not immediately apparent what the goal states of this new literal are. After connecting $\{[q,b,c]\}$ to $\{[r,d]\}$, however, we have a goal state $\{[r,d,e]\}$ of the new literal.

Computing goal states. By examining refutations of a body literal $q(\bar{h}(X))$, we can derive its goal states $\mathscr{GS}\{q(\bar{h}(X))\}$.

In some resolution steps in a refutation, prefixes of $\bar{h}$ in the initial goal are removed from chosen literals by clauses unifying with them. We want to know the clauses which remove *the last functor* of $\bar{h}$, because the heads of such clauses contain the goal states we are searching for.

Consider a literal $q(\bar{h}(X))$ and its refutation

$$G_0 = \leftarrow q(\bar{h}(X)) \xrightarrow{C_1} G_1 \xrightarrow{C_2} G_2 \xrightarrow{C_3} \cdots \xrightarrow{C_k} \Box$$

If the clause C_1 is a unit clause, G_1 is in fact the final $\Box$, and C_1 contains a goal state. Otherwise G_1 consists of subgoals $g_1, \ldots, g_n$ $(n > 0)$. The original goal G_0 is reduced to an empty goal when $g_1, \ldots, g_n$ are all reduced to empty goals. Then, the sequence of goals $G_0, G_1, \ldots, \Box$ can be represented as a proof tree [8], in which nodes at depth j denote the subgoals in G_j from left to right order. Every leaf of the tree is an empty goal, and each edge connects a literal with either its descendant subgoals, or (possibly an instance of) the same literal in the next goal.

Example 4 (*Proof tree*). Assuming the *left to right selection rule* [8] of Prolog systems, a refutation of a literal $G_0 = \leftarrow p(f(g(X)))$ by clauses $C_1, \ldots, C_6$ is represented as a proof tree:

Refutation

$$\begin{array}{lll}
G_0 & \leftarrow & p(f(g(X))). \\
C_1 & & p(f(X)) \leftarrow q(h(X)), r(k(X)). \\
G_1 & \leftarrow & q(h(g(X))), r(k(g(X))). \\
C_2 & & q(h(g(l(X)))) \leftarrow s(X). \\
G_2 & \leftarrow & s(X), r(k(g(l(X)))). \\
C_3 & & s(0). \\
G_3 & \leftarrow & r(k(g(l(0)))). \\
C_4 & & r(k(X)) \leftarrow t(X), u(X). \\
G_4 & \leftarrow & t(g(l(0))), u(g(l(0))). \\
C_5 & & t(X). \\
G_5 & \leftarrow & u(g(l(0))). \\
C_6 & & u(g(l(X))). \\
G_6 & \Box &
\end{array}$$

Proof tree

$$\begin{tikzcd}
p(f(g(X))) \arrow[d] \arrow[dr] & & \\
q(h(g(X))) \arrow[d, Rightarrow] & r(k(g(X))) \arrow[d] & \\
s(X) \arrow[d] & r(k(g(l(X)))) \arrow[d] & \\
\Box & r(k(g(l(0)))) \arrow[d] \arrow[dr] & \\
 & t(g(l(0))) \arrow[d, Rightarrow] & u(g(l(0))) \arrow[d] \\
 & \Box & u(g(l(0))) \arrow[d, Rightarrow] \\
 & & \Box
\end{tikzcd}$$

Since every leaf of the tree is an empty goal, for each path from the root to a leaf, there is exactly one edge connecting a subgoal containing a suffix of $f(g(X))$ with either a subgoal which does not contain any suffix, or an empty goal. Thick down arrows ($\Downarrow$) indicate these edges. Each clause used in such a resolution step must either contain a suffix of $f(g(X))$ in its head, or else be a unit clause with a non-ground term. The goal state derived from this refutation is the compound state, each element of which is a state in A^0 corresponding to one of these clause heads.

In the above example, because the head of C_1 contains only a prefix of the argument of G_0, both subgoals in G_1 contain g. Whereas, in the second resolution step, the head of C_2 has the argument $h(g(l(X)))$ which contains g. Thus the state $\{[q,h,g]\}$ is a component of the goal state. From the resolution step involving G_4 and C_5, another component $\{[t]\}$ is derived. The last component $\{[u,g]\}$ comes from clause C_6. The goal state from this refutation is

$$\{[q,h,g],[t],[u,g]\}.$$

The goal states $\mathscr{GS}\{g\}$ of a goal g is a set of such compound states, one from each refutation of g.

Example 5. Consider a program

$$\begin{array}{lll} p(\bar{f}(X)) \leftarrow q(\bar{h}(X)). & :: & C_1 \\ q(\bar{h}(\bar{g}(X))) \leftarrow s(\bar{a}(X)). & :: & C_2 \\ q(\bar{h}_1(X)) \leftarrow r(\bar{f}(X)). & :: & C_3 \\ r(\bar{f}(\bar{h}_2(\bar{g}(X)))) \leftarrow t(\bar{b}(X)). & :: & C_4 \end{array}$$

where in C_3 and C_4 $(\bar{h}_1, \bar{h}_2)$ is a partition of $\bar{h}$. We search for goal states of the literal $q(\bar{h}(X))$ in C_1. One derivation starts with a step using C_2.

$$\leftarrow q(\bar{h}(X)) \xrightarrow{C_2} \leftarrow s(\bar{a}(X)).$$

The last goal does not contain suffix of $\bar{h}$, since the term $\bar{h}(\bar{g}(X))$ in C_2 is an instance of the term $\bar{h}(X)$ in the literal. The state $\{[q^1, \bar{h}]\}$ is a goal state of $q(\bar{h}(X))$.

There is another derivation

$$\leftarrow q(\bar{h}(X)) \xrightarrow{C_3} \leftarrow r(\bar{f}(\bar{h}_2(X))) \xrightarrow{C_4} \leftarrow t(\bar{b}(X)).$$

The clause C_3 is used in the first step removing $\bar{h}_1$, then the clause C_4 is used, removing $\bar{h}_2$. So the state $\{[r, \bar{f}, \bar{h}_2]\}$ is another goal state of $q(\bar{h}(X))$, and $\mathscr{GS}\{q(\bar{h}(X))\} = \{\{[q^1, \bar{h}]\}, \{[r, \bar{f}, \bar{h}_2]\}\}$.

4.3. *Construction of A*

Since every body literal in a clause has a variable X in common, conjunction of literals implies that terms satisfying individual literals must share a suffix denoted by X. In the context of a finite automaton, a clause $p(\bar{f}(X)) \leftarrow q_1(\bar{h}_1(X)), \ldots, q_k(\bar{h}_k(X))$ can be interpreted as:

> If there is a path $\bar{t}$ such that, for all $1 \leqslant j \leqslant k$ each path $\bar{h}_j$ followed by $\bar{t}$ leads the automaton from $\{[q_j]\}$ to a final state *simultaneously*, then the path $\bar{f}$ followed by $\bar{t}$ leads the machine from $\{[p]\}$ to a final state.

So we need to check whether the k paths, one from each literal, share a sequence of alphabet symbols. In other to convert the conjunctive goal $q_1(\bar{h}_1(X)), \ldots, q_k(\bar{h}_k(X))$ into a graph, then, we must examine all possible k-tuples of paths, since each literal denotes a set of paths starting from corresponding goal states.

Suppose each conjunct $q_j(\bar{h}_j(X))$ has l_j goal states, i.e.

$$\mathscr{GS}\{q_j(\bar{h}_j(X))\} = \{s_j^1, \ldots, s_j^{l_j}\} \quad (1 \leqslant j \leqslant m).$$

Then we consider each of $(l_1 \times \cdots \times l_k)$ states $\{s_1^{m_1} \cup \cdots \cup s_k^{m_k}\}$ $(1 \leqslant j \leqslant k,\ 1 \leqslant m_j \leqslant l_j)$. If it is not in A, then add it to the graph and add an ε-move connecting the bottom state $\{[p, \bar{f}]\}$ with it.

Then we examine whether every component of the state has an out-going arc with the same symbol. If so, we consider another compound state consists of destination states of each component. If it is not in A add it and connect with the previous state by an

```
Procedure Expand($s, e$)

  1. If $s$ is not in $A$, then add $s$ to $A$.
  2. If $e$ is not in $A$, then add $e$ to $A$.
  3. If every component of $s = \{a_1, \ldots, a_m\}$ has a transition with the same symbol,
     say, $g$, then
     (a) consider the state $t = \{b_1, \ldots, b_m\}$ such that $(a_j \xrightarrow{g} b_j)$ are in $A$, and
     (b) recursively apply $Expand(t, s \xrightarrow{g} t)$ to $A$.
  4. Else if a component $a_k$ of $s$ has an empty move which is not in $E$, then
     (a) add the empty move to $E$,
     (b) consider the state $t = \{a_1, \ldots, a_{k-1}, b, a_{k+1}, \ldots, a_m\}$ where $(a_k \xrightarrow{\varepsilon} b)$ is the
         empty move,
     (c) recursively apply $Expand(t, s \xrightarrow{\varepsilon} t)$ to $A$.
  5. Else if every component $a_j$ in $s$ is a final state, then make $s$ a final state.
```

Fig. 7. The procedure Expand.

edge associated with the symbol. Every compound state has at most one outgoing arc with a symbol. We can view the part of the automaton corresponding to a conjunctive body as a *shadow* of a part of A^0, because every path in the new part has a set of copies in A^0. We repeat the process until no new state or arc can be added to the graph. We construct A from A^0 by converting all non-unit clauses one by one, iteratively. The automaton A is a fixpoint of this iteration.

Using an auxiliary procedure *Expand*, the formal procedure to derive A from A^0 is defined as follows.

1. Initialise $A = A^0$ and $E = \phi$. E is a set of empty moves from bottom states already considered.
2. Repeatedly apply the following procedure to A until we reach a fixpoint.
   (a) For each non-unit clause $p(\bar{f}(X)) \leftarrow q(\bar{h}_1(X)), \ldots, q_n(\bar{h}_n(X))$ construct the cross product $S = S_1 \times \cdots \times S_n$ where $S_j = \mathscr{GS}\{q_j(\bar{h}_j(X))\}$.
   (b) For each $(s_1, \ldots, s_n) \in S$
      (i) construct $s = s_1 \cup \cdots \cup s_n = \{a_1, \ldots, a_m\}$ where a_j are states in A^0.
      (ii) Apply the procedure $Expand(s, \{[p, \bar{f}\} \xrightarrow{\varepsilon} s)$ to A.

The procedure *Expand* is defined in Fig. 7.

Termination of the process: A^0 has N states, where N is the number of occurrences of functors plus the number of predicates defined in P^{us}. Every compound state in A is denoted by a set of names in A^0. So, the number of states in A cannot exceed the size of power set of the set of names in A^0, i.e. 2^N. Since the number of possible transitions is finite and the number of actual transitions in the graph increases at each step, the process always terminates.

Example 6. Consider the US program P in Fig. 8. Fig. 9 is the basis A^0 of the program P. As there are two clauses for the predicate q, the state $\{[q]\}$ has two

$p(f(g(X))) \leftarrow q(X).$		$:: C_1$
$q(h(0)).$		$:: C_2$
$q(f(X))$	$\leftarrow r(h(X)).$	$:: C_3$
$r(h(g(X)))$	$\leftarrow q(X).$	$:: C_4$
$s(X)$	$\leftarrow p(X), q(X).$	$:: C_5$

Fig. 8. A US program P.

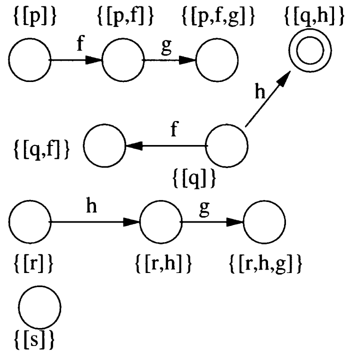

Fig. 9. A^0 of P.

outgoing edges. The double circle in the top-right corner is a final state, derived from the unit clause $q(h(0))$.

Fig. 10 shows the configuration 02 the first iteration of the conversion of non-unit clauses. In total four ε-moves (dashed arrows) are added to A^*, i.e.

$$\begin{array}{lll}
\{[p,f,g]\} & \xrightarrow{\varepsilon} \{[q]\} & \text{for } C_1, \\
\{[q,f]\} & \xrightarrow{\varepsilon} \{[r,h]\} & \text{for } C_3, \\
\{[r,h,g]\} & \xrightarrow{\varepsilon} \{[q]\} & \text{for } C_4, \text{and} \\
\{[s]\} & \xrightarrow{\varepsilon} \{[p],[q]\} & \text{for } C_5.
\end{array}$$

The state $\{[p],[q]\}$ is the first compound state generated for the conjunctive body of the clause C_5. From this state, we start tracing two paths, one from $\{[p]\}$, another from $\{[q]\}$ in parallel.

In Fig. 11, since both $\{[p]\}$ and $\{[q]\}$ have outgoing edge associated with f, a new compound state $\{[p,f],[q,f]\}$ and an edge with f are added to A. The move $\{[p,f],[q,f]\}\xrightarrow{\varepsilon}\{[p,f],[r,h]\}$ is caused by the ε-move $\{[q,f]\}\xrightarrow{\varepsilon}\{[r,h]\}$ in A^0. From $\{[p,f],[r,h]\}$ we can proceed by an edge associated with g to $\{[p,f,g],[r,h,g]\}$. Another ε-move $\{[p,f,g],[r,h,g]\}\xrightarrow{\varepsilon}\{[q],[r,h,g]\}$ is caused by $\{[p,f,g]\}\xrightarrow{\varepsilon}\{[q]\}$ in A^0. Finally, the move $\{[r,h,g]\}\xrightarrow{\varepsilon}\{[q]\}$ in A^0 leads the automaton from $\{[q],[r,h,g]\}$

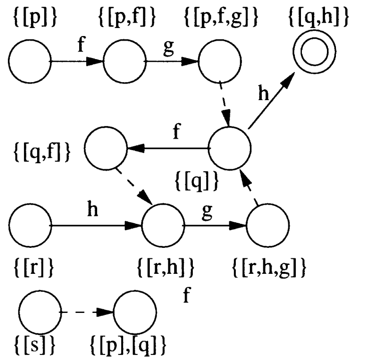

Fig. 10. The first iteration.

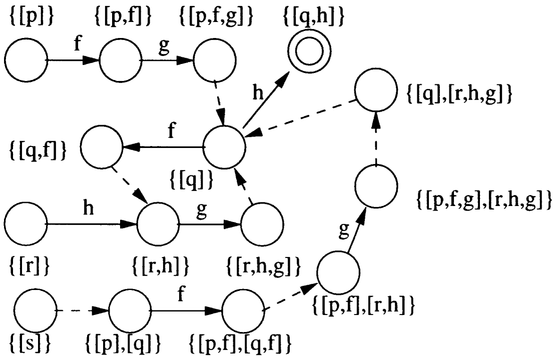

Fig. 11. The final graph.

to $\{[q]\}$. Note that in the last move, the two names in the source state reduce to one, because of union operation in building a name of a state. The subgraph starting from the state $\{[p],[q]\}$ is the intersection of subgraphs from $\{[p]\}$ and $\{[q]\}$.

4.4. Verification of the construction

The following theorem states that the constructed automaton A is equivalent to P^{us}.

Theorem 2. *The finite automaton A, constructed from P^{us} by the procedure described in the previous section, accepts a string $f_1 \dots f_n$ from a state q if and only if the goal $\leftarrow q(f_1 \dots f_n(0)\dots)$ has a refutation w.r.t. P^{us}.*

Proof. *If*: Suppose the goal $G_0 = \leftarrow q(f_1 \dots f_n(0)\dots)$ has a refutation

$$G_0 \xrightarrow{C_1} G_1 \xrightarrow{C_2} G_2 \xrightarrow{C_3} \cdots \xrightarrow{C_m} \Box \quad (m > 0),$$

where $C_1, \ldots, C_m$ are clauses in P^{us}, $G_1, G_2, \ldots$ are successive intermediate goals and $\square$ is an empty goal.

We show that there is a sequence of moves, possibly interspersed by ε-moves, from the state q to a final state, associated with symbols $f_1, \ldots, f_n$ in this order. The proof is by induction on the length of the derivation.

– *Base case*: If the goal has a one-step refutation ($m = 1$), then P^{us} must contain a unit clause C_1 of the form

$$\text{either } q(f_1 \ldots f_n(0) \ldots).$$
$$\text{or } q(f_1 \ldots f_j(X) \ldots) \quad (j \leqslant n).$$

Then, by construction, the graph contains

$$\text{either } \{[q]\} \xrightarrow{f_1} \{[q, f_1]\} \xrightarrow{f_2} \cdots \xrightarrow{f_n} \{[q, f_1, \ldots, f_n]\}^*,$$
$$\text{or } \{[q]\} \xrightarrow{f_1} \{[q, f_1]\} \xrightarrow{f_2} \cdots \xrightarrow{f_j} \{[q, f_1, \ldots, f_j]\}^* \xrightarrow{\Sigma} \{[q, f_1, \ldots, f_j]\}^*$$

where the expression $\xrightarrow{\Sigma}$ denotes the set of loops — one for each alphabet symbol. Both $\{[q, f_1, \ldots, f_n]\}^*$ and $\{[q, f_1, \ldots, f_j]\}^*$ are final states, so in either case there is a path associated with $f_1, f_2, \ldots, f_n$ from $\{[q]\}$ to a final state.

– *Inductive case*: Assume that for each goal with a refutation of length at most $k - 1$, the automaton A accepts the corresponding string, and $G_0 = \leftarrow q(f_1 \ldots f_n(0) \ldots)$ has a refutation of length k.

The first clause used in the refutation must have the form

$$q(f_1 \ldots f_i(X) \ldots) \leftarrow q_1(\bar{h}_1(X)), \ldots, q_m(\bar{h}_m(X)) \quad (0 \leqslant i \leqslant n,\ m > 0) \qquad (1)$$

Let $n_j = \#\mathscr{GS}\{q_j(\bar{h}_j(X))\}$ $(1 \leqslant j \leqslant m)$. Then the automaton contains $n_1 \times \cdots \times n_m$ ε-moves from the bottom state $\{[q, f_1, \ldots, f_i]\}$ to each of the compound states:

$$\{[q]\} \xrightarrow{f_1 \ldots f_i} \{[q, f_1, \ldots, f_i]\} \xrightarrow{\varepsilon} s_1^{i_1} \cup \cdots \cup s_m^{i_m} \quad (1 \leqslant i_j \leqslant n_j,\ 1 \leqslant j \leqslant m) \qquad (2)$$

where $s_j^{i_j} \in \mathscr{GS}\{q_i(\bar{h}_j(X))\}$.

After one step resolution, G_1 is a conjunction of m subgoals:

$$G_1 = \leftarrow q_1(\bar{h}_1(f_{i+1} \ldots f_n(0) \ldots)), \ldots, q_m(\bar{h}_m(f_{i+1} \ldots f_n(0) \ldots)).$$

Since each subgoal has a refutation of length at most $k - 1$, from the induction hypothesis, for each $1 \leqslant j \leqslant m$ A contains an accepting path:

$$\{[q_j]\} \xrightarrow{\bar{h}_j} s_j \xrightarrow{f_{i+1} \ldots f_n} \odot$$

The goal state $s_j \in \mathscr{GS}\{q_j(\bar{h}_j(X))\}$ appears in the path because each subgoal in G_1 is an instance of a body literal in (1).

Therefore A contains at least one compound state $s_j \cup \cdots \cup s_m$ leading to a final state via string $f_{i+1} \ldots f_n$:

$$s_1 \cup \cdots \cup s_m \xrightarrow{f_{i+1} \ldots f_n} \odot \qquad (3)$$

Combining (2) and (3), A contains at least one path

$$\{[q]\} \xrightarrow{f_1 \ldots f_i} \{[q, f_1, \ldots, f_i]\} \xrightarrow{\varepsilon} s_1 \cup \cdots \cup s_m \xrightarrow{f_{i+1} \ldots f_n} \odot$$

which shows that A accepts the string $f_1 \ldots f_n$.

'*Only if*': Suppose A accepts the string $f_1 \ldots f_n$ from an initial state $\{[q]\}$. We prove, by induction on the number of ε-moves in the path, that the goal

$$G = \leftarrow q(f_1 \ldots f_n(0) \ldots)$$

has a refutation in P^{us}.

– *Base case*: Firstly, consider the case where there is no ε-move in the accepting path, i.e. the path consists of exactly n moves:

$$\{[q]\} \xrightarrow{f_1} \circ \xrightarrow{f_2} \cdots \xrightarrow{f_{n-1}} \circ \xrightarrow{f_n} \odot$$

The path can be derived only from a unit clause, since any path constructed from a non-unit clause must contain at least one ε-move. Then, the program must contain a unit clause is one of the following two forms:

$$q(f_1 \ldots f_n(0)).$$
$$q(f_1 \ldots f_i(X)) \quad (0 \leqslant i \leqslant n)$$

either of which gives a one step refutation of the goal G.

– *Inductive case*: Assume that every goal corresponding to an accepting path with at most $k-1$ ε-moves has a refutation in P^{us}, and the path $\{[q]\} \xrightarrow{f_1 \ldots f_n} \odot$ contains k ε-moves. We show that the goal $G = \leftarrow q(f_1 \ldots f_n(0) \ldots)$ has a refutation. Since the path contains at least one ε-move, the program contains a clause in the form of

$$q(f_1 \ldots f_i(X) \ldots) \leftarrow q_1(\bar{h}_1(X)), \ldots, q_m(\bar{h}_m(X)) \quad (m > 0,\ 0 \leqslant i \leqslant n). \tag{4}$$

Resolution of G with this clause results in a new goal:

$$\leftarrow q_1(\bar{h}_1(f_{i+1} \ldots f_n(0) \ldots)), \ldots, q_m(\bar{h}_m(f_{i+1} \ldots f_n(0) \ldots)). \tag{5}$$

If every subgoal in (5) has a refutation, so does G.

The clause (4) corresponds to the first ε-move of the whole path:

$$\{[q]\} \xrightarrow{f_1} \cdots \xrightarrow{f_i} \{[q, f_1, \ldots, f_i]\} \xrightarrow{\varepsilon} s_1 \cup \cdots \cup s_m \xrightarrow{f_{i+1}} \cdots \xrightarrow{f_n} \odot \tag{6}$$

where each s_j is a goal state of the literal $q_j(\bar{h}_j(X))$ in (4).

Then, for each s_j, there is a derivation from $\leftarrow q_j(\bar{h}_j(\bar{f}_{i+1} \cdots \bar{f}_n(0) \ldots))$ to

$$\leftarrow r_1(\bar{g}_1(\bar{h}_1'(f_{i+1} \ldots f_n(0) \ldots))), \ldots, r_b(\bar{g}_b(\bar{h}_b'(f_{i+1} \ldots f_n(0) \ldots))) \tag{7}$$

such that $\bar{h}_1', \ldots, \bar{h}_b'$ are suffixes of $\bar{h}_j$ and $s_j = \{[r_1, \bar{g}_1, \bar{h}_1'], \ldots, [r_b, \bar{g}_b, \bar{h}_b']\}$.

The number of ε-moves in the path from $s_1 \cup \cdots \cup s_m$ to the final state in (6) is $k-1$ which is the sum of the number of ε-moves in each path starting from an element of $s_1 \cup \cdots \cup s_m$. Then every path corresponding to a subgoal in (7) has at most $k-1$ ε-moves, and hence from the induction hypothesis, each subgoal in (7) has a refutation.

Since each subgoal in (5) has a refutation, the initial goal has a refutation. □

5. Discussion and conclusion

5.1. Expressiveness of US and RUS

This subsection considers expressive differences between US and RUS. We have proven two lemmas; any US program can be transformed into an equivalent NFA with ε-moves, and any DFA can be transformed into an equivalent RUS program. Since any NFA with ε-moves can be converted into an equivalent DFA, we obtain a translation scheme from US to its subset RUS:

$$US \Rightarrow NFA \quad \text{with } \varepsilon\text{-moves} \Rightarrow DFA \Rightarrow RUS.$$

The resulting RUS program and the original US program accept exactly the same language. So the syntactic restriction of US into RUS does not hamper its expressive power in describing language acceptors.

However, two equivalent shortest programs written in US and RUS may be quite different in size. In the automaton A^0 derived from the set of clause heads in a US program, the number of states is the sum of number of predicates (N_p) and the number of occurrences of functors (N_f) in the US program. Since each state in A corresponds to a subset of sets in A^0, the number of possible states in A is $2^{(N_p+N_f)}$.

Moreover, we have to convert the automaton into a DFA. An NFA with ε-moves can be converted into an equivalent DFA in two steps, i.e. first, removing every-ε-move in a NFA, then converting the NFA into an equivalent DFA [4]. When ε-moves are removed, the number of states does not change, but conversion of an NFA into a DFA causes, potentially, an exponential increase in the number of states, since the states of resulting DFA correspond to a subset of states in the NFA. Thus the number of states in the DFA, could be, in the worst case, $2^{2^{(N_p+N_f)}}$!

However, in an automaton derived from a US program, only predicate states, bottom states, and compound states containing a bottom state of its component, have multiple outgoing edges. Besides, all outgoing edges from bottom states are ε-moves. After eliminating ε-moves by the procedure described in [4], the number of states having multiple outgoing edges, where non-determinism could occur, does not change.

5.2. Expressive power and answer set

Matos gives a different proof of a very similar equivalence theorem [6]. In his terminology *monadic logic programs* corresponds to US programs in this paper, except that Mato’s *monadic logic programs* may contain several constants but our US programs contain only one constant 0.

In order to prove equivalence between regular sets and monadic programs, Matos reduces his monadic logic programs into a subset called *linear* monadic programs, which he proves to accept exactly the set of regular languages. The resulting linear program accepts the same languages as the original general monadic program does.

Firstly he shows that if a monadic program does not contain clauses with conjunctive bodies, it can be reduced to a set of clauses in which every body predicate has no functor symbol.

Then any monadic program containing a clause with a conjunctive body is shown to be reducible to a set of linear clauses plus a clause containing only two body predicate terms each of which does not contain functors.

Finally, from this program Matos derives simultaneous equations in which every predicate name is represented by a language variable. By solving these equations he again reduces the clause with two body predicates into a linear form.

The main difference between Mato's proof and ours is that he avoids direct construction of intersection of automata corresponding to each member of a conjunctive body as we did, by first reducing the clause body to its minimal form, then proving that we can solve the corresponding algebraic equation.

In both cases the hardest part of the proof is how to handle conjunctive clause bodies. In our proof, we directly build up the intersection of body predicates by tracing the path on the graph built from clause heads. The idea that any body predicate term must have corresponding states (goal states) in the basis (A^0) is essential to our proof.

5.3. Different types of correspondences

The correspondence between US programs and regular sets in this paper is very different from the correspondences between logic programs and grammars more usually considered in the literature. Papers such as [1–3] discuss the similarity of derivation trees of grammars and proof trees of logic programs. A proof tree of a definite logic program of propositional form is similar to a derivation tree of a context free grammar (CFG). Each propositional term in the logic program corresponds to a non-terminals in the CFG. The arguments of predicate terms in more general logic programs can be viewed as an embedded stack for an indexed grammar [1], or a set of attributes in an attribute grammar [3]. Because of this extra information, their proof-tree grammars are some extension of CFG. However, authors studying such correspondences are not interested in the answer set of the logic program.

There seem to be few papers discussing the power of logic programming systems in describing language acceptors. The expressive power of logic language can be discussed in the context of the process of calculating the answer set as in this paper. Such research may provide clues about the source of the expressive power of logic languages.

5.4. Accepting power of less restricted string logic programs

This paper is based on a part of the first author's dissertation [7] which contains two other results about the accepting power to restricted pure Prolog with only unary function symbols.

If we allow the language to contain *binary* predicates, and every clause in the language contains at most one body literal, it is already powerful enough to describe acceptors of any recursively enumerable language. In other words, it can describe Turing machines: the two string arguments in a predicate can serve as the left and the right part of the tape.

If we now restrict the above binary predicate language so that the set of function symbols appearing in the first and the second arguments is disjoint, then only Context Free language can be recognised. One of the arguments can serve as a stack, while the other represents the rest of the input string.

5.5. *Conclusion and further work*

We have shown that Unary String logic programs accept exactly the set of regular languages. Also, the set of Regular Unary String logic programs, a subset of Unary String logic programs, has already enough power to accept the set of regular languages, although a shortest RUS program may have a number of predicates exponentially greater than the number of predicates and functors of its shortest US counter part.

References

[1] E. Bertsch, On the relationship between indexed grammar and logic programs, J. Logic Programming 18 (1994) 81–98.

[2] P. Deransart, J. Maluszynski, Relating logic programs and attribute grammars, J. Logic Programming 2 (1985) 119–155.

[3] P. Deransart, J. Maluszynski, What kind of grammars are logic programs?, in: P. Saint-Dizier, S. Szpakowicz (Eds.), Logic and Logic grammars for Language Processing, Ellis Horwood, Chichester, UK, 1990, pp. 29–55.

[4] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Series in Computer Science, Addison-Wesley, Reading, MA, 1979.

[5] J.W. Lloyd, Foundation of Logic Programming, 2nd Edition, Springer, Berlin, 1987.

[6] A.B. Matos, Monadic logic programs and functional complexity, Theoret. Comput. Sci. 176 (1) (1997) 175–204.

[7] T. Matsushita, Expressive power of declarative programming languages, Ph.D. Thesis, Department of Computer Science, University of York, October 1998, YCST.

[8] U. Nilsson, J. Maluszynski, Logic Programming and Prolog, 2nd Edition, Wiley, New York, 1995.

[9] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge, MA, 1986.