USING PROLOG TO PRESENT ABSTRACT MACHINES

D Crookes

Department of Computer Science
The Queen's University of Belfast
BELFAST BT7 1NN, Northern Ireland

ABSTRACT

Introductory courses in Theory of Computation usually include a study of abstract machines such as finite state machines and Turing machines. This paper demonstrates that a neat and useful way of presenting these automata is to use a logic programming language such as Prolog, making the approach useful from a teaching point of view. Not only does a Prolog specification provide a precise definition of an automoton's behaviour, but it also gives an immediately-executable simulator. The reversible execution property of Prolog programs can make these simulators inherently more powerful than traditional simulators. The paper includes Prolog specifications for finite state machines, Turing machines, linear bounded automata, and pushdown automata.

1 INTRODUCTION

Introductory courses in Theory of Computation usually include a study of different types of abstract machine[1,2,3,4]. These automata have different capabilities, and are commonly used to illustrate the different types of language identified by Chomsky. Indeed, Chomsky's hierarchy of four language classes has its direct counterpart in automata theory: there is a hierarchy of four abstract machines, each of which corresponds to a class of language in the Chomsky hierarchy. The four types of abstract machine, in increasing order of power, are:

- finite state machines
- pushdown automata
- linear bounded automata
- Turing machines

When teaching this material, we have often found it useful to supplement an informal and intuitive machine description by giving a more precise definition of each type of machine, in the form of an outline of a simulator in some sort of high level language. This constitutes an operational specification of the machine. Recently, we tried using Prolog for this, and found it to be particularly well suited to the task. The main advantages (from an educational point of view) in using Prolog are:

(i) The specification can be executed as it stands, and acts as a simulator for the machine.

(ii) The trace facilities typically available with Prolog systems can be used to give a useful single-step trace of the machine's behaviour, with no extra programming effort.

(iii) Since a Prolog clause merely specifies a logical relationship between its arguments, the reversible execution feature of Prolog programs means that the Prolog program can be used as a multi-purpose simulator. For instance, rather than supplying the input and generating the output, the same program can sometimes be used to generate the input which would be required to produce a given output.

(iv) Because of the conciseness of the Prolog versions compared with, say, a definition in Pascal, complete specifications can be presented to students, rather than just an outline.

The use of logic for specifying Turing machines has become a standard tool in Recursion Theory and Complexity Theory[5], beginning with Turing's pioneering paper[6]. Also, Tarnlund uses Horn clauses, thereby establishing Horn clauses (and hence Prolog) as a system powerful enough to express any computational process[7]. Our approach in using Prolog has a more practical motivation, however, and is more concerned with the presentation of these concepts in an educational context.

In the remainder of this paper, Prolog specifications are developed for finite state machines, Turing machines, the execution mechanism of linear bounded automata, and for pushdown automata. Where there are different forms of each automata which could be modelled, the simplest version is usually chosen. (For instance, Mealy machines are chosen for finite state machines). This is so that the approach of using Prolog is not lost in a detailed discussion of abstract machines (of which there is plenty already in the literature). It should be obvious to the reader, though, how variations on the machines presented here can be accommodated. The Prolog descriptions in this paper are presented in standard Prolog, such as defined by Clocksin and Mellish[8].

2 FINITE STATE MACHINES

A finite state machine undergoes a sequence of discrete *events*, where an event involves:

- the input of a symbol from the environment;
- a change of state;
- the output of a symbol to the environment.

These actions take place simultaneously and instantaneously.

Defining a particular finite state machine requires all posssible events to be enumerated. In Prolog, a valid event is defined as a quadruple:

```
event (current_state, input_symbol,
       output_symbol, new_state)
```

Given a set of such event descriptions, an actual finite state machine will in general be defined by the Prolog relation:

```
fsm (initial_state, input_stream,
     output_stream, final_state)
```

This states that a finite state machine starting in state *initial_state*, supplied with a stream of input symbols *input_stream*, will end up in state *final_state*, having generated a stream of output symbols *output_stream*. Note that the environment of a finite state machine is here being modelled by two lists of symbols *input_stream* and *output_stream*.

The relation *fsm*, which specifies the behaviour of a finite state machine, can be defined quite easily in Prolog. The simplest case is when there is no (more) input, in which case nothing happens:

```
fsm (Q, [], [], Q).
```

This states that a finite state machine in any state *Q*, which is supplied with no further input, will produce no output and will remain in state *Q*. In the rather more useful case where input is available, the overall effect is defined as the effect of a single event followed by the effect of a new fsm starting in the resulting new state:

```
fsm (Q, [X|S], [Y|T], Q') :-
     event (Q, X, Y, Q"),
     fsm (Q", S, T, Q').
```

Put together, these two clauses define the basic execution mechanism of any finite state machine. All that remains is to enumerate the different possible events, using the Prolog relation *event* described above. As an example, a finite state machine to check the *parity* of an input stream of 0's and 1's would require the clauses:

```
event (even, 0, 0, even).
event (even, 1, 1, odd).
event (odd,  0, 0, odd).
event (odd,  1, 1, even).
```

Prolog language features such as the anonymous variable '_' can conveniently be used when enumerating events. For instance,

```
event (_, *, error, error_state).
```

will trap the first occurrence of '*' in the input, irrespective of the state of the machine.

Executing the specification

Once the events have been defined, we have a complete Prolog program, which can be executed. This means that the specification as it stands is also a simulator. For instance, if *event* is defined as above for parity checking, consider executing the goal:

```
fsm (even, [1,0,1,1,0], X, Y)
```

If *X* and *Y* are initially uninstantiated variables, then executing this goal will result in *X* and *Y* being instantiated to *[1,0,1,1,0]* and *odd* respectively. Should the program ever reach the state where no event description matches the current configuration, *fsm* will fail.

Since the predicate *fsm* merely specifies the logical relationship in a valid finite state machine between the initial state, input stream, output stream and finite state, then it need not be dedicated to finding the final configuration from the supplied initial configuration. For instance,

```
fsm (X, Y, [1,1,1], odd)
```

effectively asks - "What initial state and input stream are required to cause the machine to end up in state *odd* having generated the output stream *[1,1,1]*?". Execution in this case would instantiate *X* and *Y* to *even* and *[1,1,1]* respectively. This feature of Prolog programs means that the simulator is multi-purpose, at no extra programming cost. However, caution must be exercised in using the simulator in this way; for instance, if we ask for an input sequence which will cause a specified final state to be reached in the following way:

```
fsm (even, X, Y, odd)
```

then executing this goal will never terminate (or will run out of memory because of the recursion). The Prolog interpreter will attempt to satisfy this goal effectively by beginning in state *even* and executing events until state *odd* is reached. In doing so, it will repeatedly select the first matching event:

```
event (even, 0, 0, even)
```

as the next event, and so will never escape from state *even*. This attempted execution will demand an unbounded input stream of 0's, which, since *X* is uninstantiated, can be made available. To use *fsm* safely as a multi-purpose program (for any set of event descriptions), one must ensure that either the input stream or the output stream is initially instantiated (and hence is finite in length). This puts an upper bound on any execution path.

3 TURING MACHINES

A Turing machine is a finite state machine with an unlimited memory in the form of a tape, accessed by a moving read/write head, as shown in figure 1. The tape of a Turing machine replaces the environment of a finite state machine. At any instant, all but a finite number of squares on the tape will be blank (indicated by the symbol *b*).

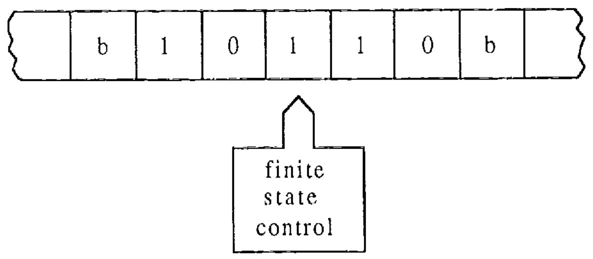

Figure 1. *A Turing machine*

An event in the life of a Turing machine can be defined by a quintuple, written in Prolog as:

```
quintuple (current_state, input_symbol,
           output_symbol, new_state,
           head_movement)
```

where input/output is from/to the current tape square. Having selected a quintuple which matches the current configuration, an event involves:

- overwriting the current square with *output_symbol;*
- entering state *new_state;*
- moving the read/write head one square in the direction indicated by *head_movement* (either <-, -> or -, standing for left, right or not at all).

The first task is to model a tape configuration – the tape contents and the position of the read/write head. One way of representing a tape configuration in Prolog would be to maintain a list of symbols for the tape, with a special marker to indicate the head position. However, specifying a head movement operation using this representation is rather awkward, and a neater way is to divide the tape into three sections, as shown in figure 2. This avoids the need for a special position marker, and limits changes in a list to just one end. This is the method which Minsky uses for arithmetisation of a tape configuration[1].

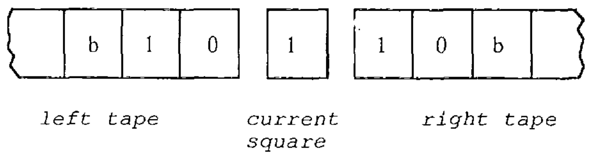

Figure 2 *Tape configuration of a Turing machine*

Such a tape configuration can be represented by a single Prolog term:

```
tape (left_part, current_square,
                 right_part)
```

where *left_part* and *right_part* are just Prolog lists. For instance, the tape configuration illustrated in figures 1 and 2 would be represented by the term

```
tape ([0,1], 1, [1,0])
```

(Note that the rightmost square of the left tape is the head of the list, so the list representing the left tape appears to be written backwards!)

The action of moving the head is specified by the predicate

```
move (tape_before, direction,
                   tape_after)
```

There are three cases of *move* to be considered – one for each possible direction. The most straightforward case is when *direction* indicates no movement; here, the tape after is the same as the tape before:

```
move (T, -, T).
```

Moving a configuration right lengthens the left tape by one, and reduces the right tape by one:

```
move (tape(L, C, [X|R]), -> ,
      tape([C|L], X, R)).
```

Although a left move could now be defined by a similar clause, this definition of a right move can itself be used to define a left move. By looking at a diagram of a tape, such as in figure 2, it can be seen that moving the head left is the same as turning the tape upside down, moving the head right, and inverting the tape once again. This is expressed by the predicate

```
move (tape(L,C,R), <- ,
      tape(L',C',R')) :-
    move(tape(R,C,L), -> ,
    tape(R',C',L')).
```

Were it not for the possibility of running off the end of the tape (to be discussed presently), a left move could have been defined even more simply as

```
move (T, <- , T') :-
    move (T', -> , T).
```

Representing an infinite tape

Strictly speaking, the length of a Turing machine tape is unbounded rather than infinite. Should the read/write head attempt to move off the end, we can imagine the tape being dynamically extended with some more blank tape. Our predicate *move* as it now stands will fail when it attempts to move off the end of the tape, say when moving right:

```
move (tape([1], 1, []), -> , T).
```

To model the action of creating a new (blank) square, we add an additional *move* clause to cover this special case:

```
move (tape(L, C, []), -> ,
      tape([C|L], b, [])).
```

Since a move to the left is defined in terms of a right move, no additional clause is required to handle moving off the left hand end of the tape.

We can now move on to an event in the life of a Turing machine, to be defined by the predicate:

```
event (current_state, current_tape,
       new_state, new_tape)
```

Such an event involves selecting a quintuple, and carrying out the three actions mentioned previously:

```
event (Q, tape(L,C,R), Q', T) :-
    quintuple (Q, C, S, Q', D),
    move (tape(L,S,R), D, T).
```

Assuming that all the quintuples have been defined, a complete Turing machine can now be specified by the predicate

```
tm (initial_state, initial_tape,
                   final_tape)
```

This says that a Turing machine starting in state *initial_state* with tape configuration *initial_tape* will halt (in a special state *halt*) with tape configuration *final_tape*. To write Prolog clauses defining the relation *tm*, take the simplest case when the Turing machine is already in the 'halt' state:

```
     tm (halt, T, T).
```

The more general case executes one event and then runs from the new state:

```
     tm (Q, T, T') :-
          event (Q, T, Q', T"),
          tm (Q', T", T').
```

Gathering together all the necessary clauses developed to date, we obtain a complete Prolog specification for a Turing machine as follows:

```
     tm (halt, T, T).
     tm (Q, T, T') :-
          event (Q, T, Q', T"),
          tm (Q', T", T').

     event (Q, tape(L,C,R), Q', T) :-
          quintuple (Q, C, S, Q', D),
          move (tape(L,S,R), D, T).

     move (T, - , T).
     move (tape(L,C,[X|R]), -> ,
                             tape([C|L],X,R)).
     move (tape(L,C,[]),       -> ,
                            tape([C|L],b,[])).
     move (tape(L,C,R),        <- ,
           tape(L',C',R'))  :-
          move (tape(R,C,L), -> ,
                tape(R',C',L')).
```

Executing the specification

Assuming that a set of quintuples has been defined, the above specification as it stands constitutes a complete Prolog program which will run and simulate the behaviour of a Turing machine. For instance, we can define quintuples for a parity checker by adding the following Prolog clauses to the program:

```
     quintuple(even, 0, 0, even, ->).
     quintuple(even, 1, 1, odd,  ->).
     quintuple(odd,  0, 0, odd,  ->).
     quintuple(odd,  1, 1, even, ->).
     quintuple(even, b, even, halt, - ).
     quintuple(odd,  b, odd,  halt, - ).
```

If we executed the goal:

```
     tm (even, tape([],1,[0,1,1,0]), T)
```

the final result tape T would be instantiated to:

```
     tape([0,1,1,0,1],odd,[])
```

where the symbol under the read/write head (*odd*) indicates the result. Note that leading or trailing blanks are not removed from the tape representation.

When using reverse execution of *tm*, one again needs to be careful that the computation is definitely finite. Now that we can have a potentially infinite tape, this is more difficult than for a finite state machine. A Turing machine computation even on a finite tape can go on for ever. However, it can still be possible to use the facility, as in:

```
    tm (Q, tape([],1,[0,1,1,0]),
           tape([0,1,1,0,1],odd,[]))
```

which asks what initial state *Q* will cause the machine with the given initial and final tape configurations to halt. With the above quintuples defined for parity checking, this will instantiate *Q* to *even*.

4 LINEAR BOUNDED AUTOMATA

The basic execution mechanism of a linear bounded automaton is similar to that of a Turing machine. The only difference is that the length of tape available to a linear bounded automaton is bounded rather than potentially infinite. In our formulation in Prolog, we therefore remove the ability to extend the tape dynamically from the basic execution mechanism. It is assumed that the entire tape available to the mechanism is supplied as input (either by the user, or by an intermediate operation which extends the tape linearly). Removing this extension capability from the previous specification of a Turing machine thus gives us the following Prolog definition of the execution mechanism of a linear bounded automaton. Strictly speaking, it is not as it stands a complete linear bounded automaton - only the execution mechanism of one.

```
     lba (halt, T, T).
     lba (Q, T, T') :-
          event (Q, T, Q', T"),
          lba (Q', T", T').

     event (Q, tape(L,C,R), Q', T) :-
          quintuple (Q, C, S, Q', D),
          move (tape(L,S,R), D, T).

     move (T, - , T).
     move (tape(L,C,[X|R]), -> ,
           tape([C|L],X,R)).
     move (T, <- , T') :-
          move (T', -> , T).
```

The parity checking mechanism can be solved by *lba* using the same quintuples as before, as follows:

```
     lba (even, tape([],1,[0,1,1,0,b]), T)
```

Note that the working space (one blank square in this case, to write the result) must be supplied to *lba*.

5 PUSHDOWN AUTOMATA

Architecturally, a pushdown automaton is the most complex of the four common automata, because it has two memories: a bounded read-only input tape, and an unbounded read/write pushdown stack, illustrated in figure 3.

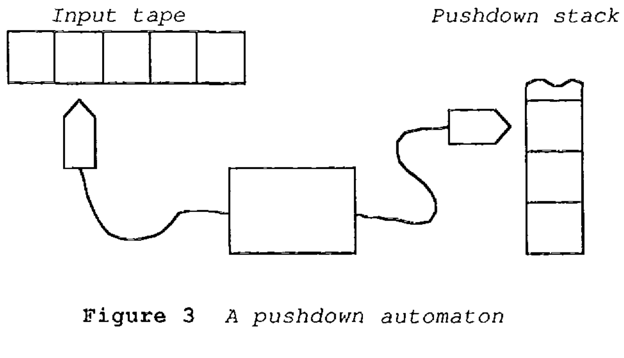

Figure 3 *A pushdown automaton*

Again, all possible events must be enumerated for a definition of a specific pushdown automaton. An event is selected on the basis of the current input symbol (on the input tape), the current state of the machine, and the symbol on top of the stack. An event causes the following actions:

- move the input reading head one square right (or perhaps leave it where it is – see below);
- perform a stack action: either pop, do nothing, or push a symbol;
- change state.

If no event description matches the current configuration, the machine halts.

In Prolog, an event in a pushdown automaton is defined by a quintuple:

```
quintuple (current_state,
           input_symbol,top_of_stack,
           new_state, stack_action)
```

where stack_action is either *pop*, '-' (do nothing), or any other symbol (which will be pushed). Note that there is no explicit directive for controlling the input reading head: by default it moves right, but in the special case where *input_symbol* in the selected quintuple is the token ¬ (which will match any input tape symbol, or the empty string), the reading head is left unchanged.

Representation of a pushdown automaton in Prolog is not difficult. The input tape can be defined by a Prolog list. So too can the stack; and the effect of a stack action can be defined by a predicate:

```
stack_action (old_stack, action,
              new_stack)
```

where the three action types are defined by separate clauses:

```
stack_action ([X|S], pop, S).
stack_action (S, - , S).
stack_action (S, X, [X|S]).
```

There are three types of event to be defined: an 'ordinary' event, the special case where ¬ appears in the selected quintuple, and the case when no quintuple matches (halting the machine). Each of these cases is specified by a predicate:

```
event (current_state, input_tape,
       current_stack, new_state,
       new_tape, new_stack)
```

as follows:

```
event (Q, [X|I], [Y|S], Q', I, S') :-
      quintuple (Q, X, Y, Q', D),
      stack_action ([Y|S], D, S').

event (Q,  I,  [Y|S], Q', I, S') :-
      quintuple (Q, ¬, Y, Q', D),
      stack_action ([Y|S], D, S').

event (Q, I, [Y|S], halt, I, [Y|S]).
```

Note that an event always requires the stack to be non-empty, since the top of stack is used in selecting a quintuple. If the stack should ever become empty, the event is undefined and will fail.

Finally, a complete pushdown automaton can be specified by the predicate:

```
pda (initial_state, initial_tape,
     initial_stack, final_tape,
     final_stack)
```

as follows:

```
pda (halt, I, S, I, S).
pda (Q, I, S, I', S') :-
     event (Q, I, S, Q', I", S"),
     pda (Q', I", S", I', S').
```

6 CONCLUSION

This study has shown that Prolog can be a useful formalism for communicating the nature and behaviour of abstract machines in a precise but understandable way (provided the reader has some familiarity with Prolog). It also leads to an immediately- executable simulator, whose usefulness can be further extended, at no extra programming effort, by using the Prolog trace facility to emulate single step operation. This makes the approach a useful teaching aid. It can be seen that the same approach could also be applied to the modelling of more complex machines – in particular, existing or proposed computer systems.

REFERENCES

[1] Minsky, M.L., "Computation: finite and infinite machines", Prentice-Hall Inc., Englewood Cliffs, N.J. (1967).

[2] Hopkin, D. and Moss, B., "Automata", Macmillan, London (1976).

[3] Kain, R.Y., "Automata theory: machines and languages", McGraw-Hill Inc., London (1972).

[4] Brady, J.M., "The theory of computer science: a programming approach", Chapman and Hall, London (1977).

[5] Borger, "Spekralproblem and completeness of logical decision problems", Lecture Notes in Computer Science 171, pp.333-356, Springer-Verlag.

[6] Turing, A.M., "On computable numbers with an application to the Entscheidungsproblem", Proc. London Math. Soc., Ser. 2, Vol 42 (1936/37).

[7] Tarnlund, S., "Horn clause computability", BIT 17, pp. 215- 226 (1977).

[8] Clocksin and Mellish, C.S., "Programming in Prolog", 2nd edition, Springer-Verlag (1984).