Simulating Turing Machines in DATR

Lionel Moser
School of Cognitive & Computing Sciences
University of Sussex
Brighton, U.K.

July 1992

Abstract

In this paper we show how an arbitrary Turing machine can be simulated in DATR, and show that the computational complexity of DATR is Turing equivalent – and hence termination of query evaluation is undecidable.

Contents

1 Introduction 1

2 The Hopcroft & Ullman Turing machine 2

3 A DATR representation 3

4 Conclusion 6

Appendices 8
turing.dtr - A Turing machine simulator recognising $L = \{0^n1^n \mid\mid n \geq 1\ \}$ 8
Argument manipulation primitives 10

1 Introduction

(Moser 1992b) proved a lower bound of co-NP-hard for the computational complexity of DATR (Evans and Gazdar 1989a, Evans and Gazdar 1989b, Evans and Gazdar 1990) query evaluation, first by showing how an NP-complete problem – determining satisfiability of an arbitrary boolean formula – could be solved in DATR. It then showed that the complement of the set of satisfiable boolean formulas of propositional calculus, those which are unsatisfiable, can also be recognised by a DATR theory.

To say that the computational complexity is undecidable means that it is impossible to establish a lower bound on the time taken for a well-formed, functional theory to evaluate a query. Such a theory may recognise a language for which there is no effective algorithm (one which halts) for recognising non-membership in the language. Recognising membership of a string in the language will halt, but one cannot predict in general whether the string being tested is in the language or not.

2 The Hopcroft & Ullman Turing machine

Following Hopcroft & Ullman (1979, pp. 147-150) we define a Turing machine as an ordered tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where

Q is the finite set of states,
Γ is the finite set of allowable tape symbols,
B, a symbol of Γ, is the blank,
Σ, a subset of Γ not including B, is the set of input symbols,
δ is the next move function, a partial function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
$q_0 \in Q$ is the start state,
$F \subseteq Q$ is the set of final states.

An *instantaneous description* (ID) of the Turing machine M is denoted by $\alpha_1 q \alpha_2$, where q is the current state, and $\alpha_1\alpha_2$, a string in Γ^* is the current content of the tape up to the rightmost nonblank symbol. The tape is bounded on the left (at the leftmost symbol of α_1) and is unbounded on the right, all cells to the right of α_2 having value B. The tape head is scanning the leftmost symbol of α_2, unless α_2 is the empty string, in which case the head is scanning a blank.

A Turing machine which recognises the language $L = \{0^n1^n \mid n \geq 1\}$ is given in Hopcroft & Ullman. The TM M is defined by:
$Q = \{q_0, q_1, q_2, q_3, q_4\}$,
$\Sigma = \{0, 1\}$,
$\Gamma = \{0, 1, X, Y, B\}$,
$F = \{q_4\}$,
and the transition function δ is as defined in Figure 1.

	Symbol				
State	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_3, Y, R)	–
q_2	$(q_0, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 1: The function δ

Figure 2 shows the computation of M accepting string 0011:

$q_0 0011$	$\vdash$	Xq_1011	$\vdash$	$X0q_111$	$\vdash$	Xq_20Y1	$\vdash$
q_2X0Y1	$\vdash$	Xq_00Y1	$\vdash$	XXq_1Y1	$\vdash$	$XXYq_11$	$\vdash$
XXq_2YY	$\vdash$	Xq_2XYY	$\vdash$	XXq_0YY	$\vdash$	$XXYq_3Y$	$\vdash$
$XXYYq_3$	$\vdash$	$XXYYBq_4$					

Figure 2: A computation of M

3 A DATR representation

To simulate a Turing machine we require representations of an infinite tape, a finite-state control, and each of the elements of the ordered tuple M. The simulated Turing machine uses the path to represent the tape, or more precisely as a path prefix; and the control we represent as a combination of a path prefix and an inheritance specification. An ID $\alpha_1 q \alpha_2$ is represented as a three-argument list:

$$\texttt{<x}_0\ \texttt{x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_{i-1}\ \texttt{;}\ \texttt{q}\ \texttt{;}\ \texttt{x}_i\ \texttt{x}_{i+1}\ \ldots\ \texttt{x}_n\ \texttt{;>}$$

where $\alpha_1 = \texttt{x}_0\ \texttt{x}_1\ \texttt{x}_2 \ldots \texttt{x}_{i-1}$, $\alpha_2 = \texttt{x}_i\ \texttt{x}_{i+1} \ldots \texttt{x}_n$, and $\texttt{x}_i$ is the current symbol. Using techniques presented in Moser (1992a) to treat the path as an argument list, we will access the path as a stack accessed from the left side. We define a main node, say `M`, such that the value of a path (representing an ID) at that node is the value which Turing machine M would compute if started from that ID. We do this using two mutually recursive nodes, `M` and `Apply_delta`. A step of computation from ID$_i$ to ID$_{i+1}$ requires two cycles through `M` and one through `Apply_delta`. The first cycle through `M` computes the value of $\delta(q, \gamma)$ (a triple) and pushes it (as a path prefix), prefaced by the atom `delta`. The second cycle through `M` tests whether δ returned a value or is undefined. If it is not defined, then `M` evaluates to either `accept` or `reject` depending on whether q is a final state. If $\delta(q, \gamma)$ is defined, then `M` inherits from `Apply_delta`, which pops $\delta(q, \gamma)$ (as a matched prefix), applies it to the current ID$_i$, and inherits from `M:<`ID$_{i+1}$`>`. Figure 3 outlines the mutual recursion through which the computation of M is simulated, where the last step of computation could be either `accept`, as shown, or `reject`.

M:<ID$_1$>	= M:<delta $\delta(q,\gamma)$ ID$_1$>
	= Apply_delta:<$\delta(q,\gamma)$ ID$_1$>
	= M:<ID$_2$>
	= M:<delta $\delta(q,\gamma)$ ID$_2$>
	= Apply_delta:<$\delta(q,\gamma)$ ID$_2$>
	= M:<ID$_3$>
	$\vdots \quad \vdots$
	= M:<ID$_n$>
	= M:<delta undef ID$_n$>
	= accept

Figure 3: Computation via mutual recursion of `M` and `Apply_delta`

We now present the DATR translation of TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. We represent states ($Q$) and tape symbols ($\Gamma$) as atoms, and define DATR variables ranging over each of these sets.

```
% states
#vars $state: q0 q1 q2 q3 q4.

% tape symbols
#vars $gamma: 0 1 x y b.
```

The definition of node `M` will use argument manipulation primitives defined in Moser (1992a). These primitives are parameterized over several variables, the most interesting being `$terminal`, which enumerates the atoms over which arguments are constructed. The two separators `;` and `!` serve to terminate arguments and the argument list, repsectively. For a Turing machine, the `$terminals` are the state and tape symbols, plus moves L and R:

```
#vars $terminal: q0 q1 q2 q3 q4 0 1 x y b l r.
```

Under the path-as-argument-list interpretation, α_1, q, and α_2 are the first, second and third arguments, respectively, so we define several nodes which function as argument extractors:

```
Alpha1:<> == Arg1.
Curq:<> == Arg2.
Alpha2:<> == Arg3.
```

The transition function δ is simply a table look-up. $\delta(q, \gamma) = (q', \gamma', d)$, or in our DATR notation `Delta:<q g> == (q' ; g' ; d)`, where `q` and `g` are the current state and tape symbol being scanned, `q'`, `g'`, and `d` are the new state, the symbol replacing `g` on the tape, and the direction in which the head moves, respectively. Noting that this particular transition table is a sparse matrix, we define a default of `undef` and the specify the value of δ for the pairs for which it is defined:

```
Delta: <> == undef
       <q0 0> == (q1 ; x ; r ;)
       <q0 y> == (q3 ; y ; r ;)
       <q1 0> == (q1 ; 0 ; r ;)
       <q1 1> == (q2 ; y ; l ;)
       <q1 y> == (q1 ; y ; r ;)
       <q2 0> == (q2 ; 0 ; l ;)
       <q2 x> == (q0 ; x ; r ;)
       <q2 y> == (q2 ; y ; l ;)
       <q3 y> == (q3 ; y ; r ;)
       <q3 b> == (q4 ; b ; r ;).
```

Testing membership in the set of final states requires one non-default statement for each node in F, as shown in node `Final`:[1]

```
Final: <> == false
       <q4> == true.
```

The current symbol scanned by the read/write head of M is the leftmost symbol of α_2, unless α_2 is nil, in which case the current symbol is the blank. Node `Cursym` evaluates to the current symbol using negative path extension: the statement prefixed by `<nil ;>` will be matched when the value of `Alpha2` is the empty list; otherwise the statement prefixed by `<nil>` will be matched:

```
Cursym:<> == <nil Alpha2 ;>
       <nil ;> == b
       <nil> == First:<>.
```

The effect of evaluating an ID at `Cursym` yields the first symbol of α_2. In the second theorem below, α_2 is the empty string:

```
Cursym: <; q0 ; 0 0 1 1 ;> = 0
         <x x y y ; q3 ; ;> = b.
```

Before presenting the definition of `M` we first introduce a new primitive, `Last`, which evaluates to the last atom in an argument, or the empty list if the argument is nil. This will be used to simulate moving the read/write head to the left; the rightmost symbol of α_1 needs to be removed from α_1 and inserted to the right of the current state q.

[1]Moser (1992c) discusses the representation of disjunction in DATR at length.

```
Last:<$terminal ;> == $terminal
     <;> == ()
     <$terminal> == <>.
```

We now present the definition of `M` such that `M:<ID> = accept`, or `M:<ID> = reject`, where ID is of the form `<`x_0 x_1 x_2 ... x_{i-1} `;` q `;` x_i x_{i+1} ... x_n `;>`:

```
M:<> == <delta Delta:<Curq Cursym>>
  <delta undef> == <If:<Final:<Curq:<>> > >
     <then> == accept
     <else> == reject
  <delta> == Apply_delta:<>.
```

The first statement of `M` indicates that

`M:<`ID$_i$`> = M:<delta` $\delta(q, \gamma)$ ID$_i$`>`

and from the last statement of `M` we see that

`M:<delta` $\delta(q, \gamma)$ ID$_i$`> = Apply_delta:<`$\delta(q, \gamma)$ ID$_i$`>`

and we require that

`Apply_delta:<`$\delta(q, \gamma)$ ID$_i$`> = M:<`ID$_{i+1}$`>`

For each triple (q', γ', d) in the range of δ, node `Apply_delta` has a path definition which constructs ID$_{i+1}$ from the ID$_i$ suffixed to the triple. Node `Apply_delta` has two forms of statement, corresponding to transitions which move the read/write head left and right. For example, for (q_1, X, R) $(= \delta(q_0, 0))$ `Apply_delta` has the following statement:

```
Apply_delta: <q1 ; x ; r ;> == M:<Alpha1:<> x ; q1 ; Rest:<Alpha2:<> ;> ; !>.
```

From this statement (and the definitions of the nodes on which it depends), it follows that:

`Apply_delta:<q1 ; x ; r ;` x_0 x_1 x_2 ... x_{i-1} `; q0 ;` x_i x_{i+1} ... x_n `;>`
`= M:<`x_0 x_1 x_2 ... x_{i-1} `x ; q1 ;` x_{i+1} ... x_n `; !>`

When the read/write head moves left, as in (q_2, Y, L) $(= \delta(q_2, Y))$ `Apply_delta` has the following statement:

```
Apply_delta: <q2 ; y ; l ;> == M:< Remove_last:<Alpha1:<> ;> ;
                                   q2 ;
                                   Last:<Alpha1:<> ;> y Rest:<Alpha2:<> ;> ;
                                !>.
```

From this statement (and the definitions of the nodes on which it depends), it follows that:

`Apply_delta:<q2 ; y ; l ;` x_0 x_1 x_2 ... x_{i-1} `; q0 ;` x_i x_{i+1} ... x_n `;>`

`= M:<`x_0 x_1 x_2 ... x_{i-2} `; q2 ;` x_{i-1} `y` x_{i+1} ... x_n `; !>`

Although `Apply_delta` need have only the same number of statements as there are triples mapped onto by δ (in this case 10), it is simpler and clearer to define a statement for every triple in $Q \times \Gamma \times \{L, R\}$, as illustrated below using the DATR variables defined previously:

```
Apply_delta:
   <$state ; $gamma ; r ;> == M:< Alpha1:<> $gamma ;
                                  $state ;
                                  Rest:<Alpha2:<> ;> ;
                               !>
   <$state ; $gamma ; l ;> == M:< Remove_last:<Alpha1:<> ;> ;
                                  $state ;
                                  Last:<Alpha1:<> ;> $gamma Rest:<Alpha2:<> ;> ;
                               !>.
```

The number of statements comprising node `Apply_delta` is $|Q| \times |\Gamma| \times |\{L, R\}|$, since the variables get expanded out to one statement for each atom in their respective ranges.

The first four IDs in the computation of `M:<; q0 ; 0 0 1 1 ;>` are shown in Figure 4, along with the intermediate stages. A comparison with Figure 2 shows that the sequence of IDs is identical to those progressed through by Turing machine M.

```
M:<; q0 ; 0 0 1 1 ;>
  = M:<delta q1 ; x ; r ; ; q0 ; 0 0 1 1 ;>
  = Apply_delta:<q1 ; x ; r ; ; q0 ; 0 0 1 1 ;>
  = M:<x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>

  = M:<delta q1 ; x ; r ; ; q0 ; 0 0 1 1 ;>
  = Apply_delta:<q1 ; 0 ; r ; x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>
  = M:<x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>

  = M:<delta q2 ; y ; l ; x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>
  = Apply_delta:<q2 ; y ; l ; x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>
  = M:<x ; q2 ; 0 y 1 ; !  x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !  ; q0 ; 0 0 1 1 ;>
```

Figure 4: The first few steps of computation

When `Apply_delta` inherits from `M`, the default extension which is appended to the constructed path is the previous sequence of IDs. After n steps of computation the path looks like:

$$\boxed{< \mathrm{ID}_n \ ! \ \ldots \ ! \ \mathrm{ID}_3 \ ! \ \mathrm{ID}_2 \ ! \ \mathrm{ID}_1 \ >}$$

To prevent the trailing sequence of IDs from 'interfering' with the simulated computation, the contructed path is terminated with `!`, a symbol which the argument extractors (`Arg1`, `Arg2`, etc.) never look beyond.

The last computation of M occurs when δ is undefined. In this case the prefix inserted on the first cycle is `delta undef` and on the second cycle evaluation terminates. Figure 5 (page 7) shows the last step of computation for the sequence begun in Figure 4 case where `M` accepts. Theorems of this theory conform to the results computed by the equivalent Turing machine. By starting compution at the initial ID we have:

```
M: <; q0 ; 0 ;> = reject
   <; q0 ; 1 ;> = reject
   <; q0 ; 0 1 ;> = accept
   <; q0 ; 0 0 1 ;> = reject
   <; q0 ; 0 1 1 ;> = reject
   <; q0 ; 0 0 1 1 ;> = accept
   <; q0 ; 0 0 0 1 1 ;> = reject
   <; q0 ; 0 0 0 1 1 1 ;> = accept.
```

4 Conclusion

The mapping from the statement of the Turing machine M to a DATR theory which executes it is straightforward. The size of the DATR theory is linear with respect to the size of the Turing machine. The language accepted by the DATR theory is identical to that accepted by the Turing machine, and the abstract steps of computation are identical. In practice, however, the

```
= M:< x x y y b ; q4 ; ; !
      x x y y ; q3 ; ; !  x x y ; q3 ; y ; !  x x ; q0 ; y y ; !
      x x y y ; q3 ; ; !  x x y ; q3 ; y ; !  x x ; q0 ; y y ; !
      x ; q2 ; x y y ; !  x x ; q2 ; y y ; !  x x y ; q1 ; 1 ; !
      x x ; q1 ; y 1 ; !  x ; q0 ; 0 y 1 ; !   ; q2 ; x 0 y 1 ; !
      x ; q2 ; 0 y 1 ; !  x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !
      ; q0 ; 0 0 1 1 ;>
= M:< delta undef x x y y b ; q4 ; ; !
      x x y y ; q3 ; ; !  x x y ; q3 ; y ; !  x x ; q0 ; y y ; !
      x ; q2 ; x y y ; !  x x ; q2 ; y y ; !  x x y ; q1 ; 1 ; !
      x x ; q1 ; y 1 ; !  x ; q0 ; 0 y 1 ; !   ; q2 ; x 0 y 1 ; !
      x ; q2 ; 0 y 1 ; !  x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !
      ; q0 ; 0 0 1 1 ;>
= M:< then x x y y b ; q4 ; ; !
      x x y y ; q3 ; ; !  x x y ; q3 ; y ; !  x x ; q0 ; y y ; !
      x ; q2 ; x y y ; !  x x ; q2 ; y y ; !  x x y ; q1 ; 1 ; !
      x x ; q1 ; y 1 ; !  x ; q0 ; 0 y 1 ; !   ; q2 ; x 0 y 1 ; !
      x ; q2 ; 0 y 1 ; !  x 0 ; q1 ; 1 1 ; !  x ; q1 ; 0 1 1 ; !
      ; q0 ; 0 0 1 1 ;>
= accept.
```

Figure 5: The final step of computation by node `M`

DATR theory does more work in the parameter passing, as the default extension consisting of the sequence of IDs comprising the computation is appended repeatedly to the path passed from `Apply_delta` to `M`.

References

[Evans and Gazdar 1989a] Roger Evans and Gerald Gazdar. Inference in DATR. In *ACL Proceedings, 4th European Conference*, pages 1–9, Manchester, 1989.

[Evans and Gazdar 1989b] Roger Evans and Gerald Gazdar. The semantics of DATR. In Anthony G. Cohn, editor, *Proceedings of the Seventh Conference of the Society for Artificial Intelligence and the Simulation of Behaviour*, pages 79–87, Brighton, UK, 1989.

[Evans and Gazdar 1990] Roger Evans and Gerald Gazdar, editors. *The DATR Papers, Volume I.* CSRP 139. School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1990.

[Hopcroft and Ullman 1979] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[Moser 1992a] Lionel Moser. DATR paths as arguments. Technical Report CSRP 215, School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1992.

[Moser 1992b] Lionel Moser. Evaluation in DATR is co-NP-hard. Technical Report CSRP 240, School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1992.

[Moser 1992c] Lionel Moser. Lexical constraints in DATR. Technical Report CSRP 216, School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1992.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% File:            turing.dtr                                        %
% Purpose:         Simulate a Turing machine.                        %
% Author:          Lionel Moser, June  1992                          %
% Documentation:   HELP *datr                                        %
% Related Files:   lib datr; args.dtr; arglogic.dtr;                 %
% Version:         2.00                                              %
%     Copyright (c) University of Sussex 1992.  All rights reserved.    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This theory simulates a Turing machine which recognises the language
% L={0^n 1^n | n >= 1}, taken from Hopcroft & Ullman (1979:147-150).
% The DATR theory simultates a TM defined by the ordered tuple
% M=(Q,Sigma,Gamma,Delta,q0,B,F).

#vars $state: q0 q1 q2 q3 q4.    % Q
#vars $move: l r.                % {L,R}

% Tape symbols: 0, 1, x, y, b    % B = b
#vars $gamma: 0 1 x y b.         % Gamma
#vars $sigma: 0 1.               % Sigma

#vars $final: q4.                % F

#vars $terminal: q0 q1 q2 q3 q4 l r 0 1 x y b.

#load 'args.dtr'.       % The required extract of these
#load 'arglogic.dtr'.   % files follows in primitives.dtr

% An instantaneous description (ID) is a string
%   <Alpha1 q Alpha2>
% where Alpha1 and Alpha2 are strings over the tape alphabet.
% We represent this as a 3-argument list:
%   <X0 X1 X2 ... Xi-1 ; q ; Xi ... Xn ;>

Alpha1:<> == Arg1.   % X0 X1 ... Xi-1
Curq:<> == Arg2.     % current state q
Alpha2:<> == Arg3.   % Xi Xi+1 ... Xn

% Current symbol is Xi, or b (blank) if Alpha2 is the empty string.
Cursym:<> == <nil Alpha2 ;>
       <nil ;> == b
       <nil> == First:<>.
```

```
% Final states (= {q4} in this example)
Final: <> == false
       <$final> == true.

% The Delta function (a partial function) is stored as a look-up table.
% Delta:<q a> == (q' ; a' ; {r/l} ;)
Delta:
   <> == undef

   <q0 0> == (q1 ; x ; r ;)
   <q0 y> == (q3 ; y ; r ;)

   <q1 0> == (q1 ; 0 ; r ;)
   <q1 1> == (q2 ; y ; l ;)
   <q1 y> == (q1 ; y ; r ;)

   <q2 0> == (q2 ; 0 ; l ;)
   <q2 x> == (q0 ; x ; r ;)
   <q2 y> == (q2 ; y ; l ;)

   <q3 y> == (q3 ; y ; r ;)
   <q3 b> == (q4 ; b ; r ;).

% Last is the last symbol in an argument, or nil if the arg is nil.
Last:<$terminal ;> == $terminal
     <;> == ()
     <$terminal> == <>.

% M:<ID>
% M:<X0 ... Xi-1 ; q ; Xi Xi+1 ... Xn ;>
M:<> == <delta Delta:<Curq Cursym>>
  <delta undef> == <If:<Final:<Curq:<>> > >
     <then> == accept
     <else> == reject
  <delta> == Apply_delta:<>.

% Apply_delta:<Delta(IDi) IDi> == M:<IDi+1>
% Apply_delta<q1 ; X ; R ; X0 ... Xi-1 ; q ; Xi Xi+1 ... Xn ;>
Apply_delta:
  % M:<X0 ... Xi-1 Xi' ; q' ; Xi+1 ... Xn ;>
  <$state ; $gamma ; r ;> == M:< Alpha1:<> $gamma ;
                                $state ;
                                Rest:<Alpha2:<> ;> ;
                             !>
  % M:<X0 ... Xi-2 ; q' ; Xi Xi' Xi+1 ... Xn ;>
  <$state ; $gamma ; l ;> == M:< Remove_last:<Alpha1:<> ;> ;
                                $state ;
                                Last:<Alpha1:<> ;> $gamma Rest:<Alpha2:<> ;> ;
                             !>.
```

```
% Some theorems ----------------------

% M: <; q0 ; 0 ;> = reject
%    <; q0 ; 1 ;> = reject
%    <; q0 ; 0 1 ;> = accept
%    <; q0 ; 0 0 1 ;> = reject
%    <; q0 ; 0 1 1 ;> = reject
%    <; q0 ; 0 0 1 1 ;> = accept
%    <; q0 ; 0 0 0 1 1 ;> = reject
%    <; q0 ; 0 0 0 1 1 1 ;> = accept
%    <; q0 ; 0 0 0 1 1 1 1 ;> = reject.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%                                                                         %
% File:           primitives.dtr                                          %
% Purpose:        Primitives used by Turing.dtr                           %
% Authors:        Lionel Moser, June 1992.                                %
% Version:        5.00                                                    %
%      Copyright (c) University of Sussex 1992.  All rights reserved.     %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

Arg1: <!> == ()
      <;> == ()
      <$terminal> == ($terminal <>).

Arg2: <> == Arg1:<Pop_arg>.

Arg3: <> == Arg1:<Pop_arg:<Pop_arg>>.

First: <$terminal> == $terminal.

Second: <$terminal> == First:<>.

Pop_arg: <!> == ()
         <;> == Arglist:<>
         <$terminal> == <>.

Pv: <>  == ()
   <!> == ()
   <;> == (; <>)
   <$terminal> == ($terminal <>).

Arglist:<> == Pv.

Rest: <;> == ()
      <$terminal> == Pv_to_;:<>.

Pv_to_;: <;> == ()
         <$terminal> == ($terminal <>).

Remove_last: <> == Reverse:<Rest:<Reverse ;> ;>.

Reverse: <;> == ()
         <$terminal> == (<> $terminal).

If: <true> == then
    <false> == else.
```