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

2 he Hopcroft & Ullman uring machine

3 A DA R 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 $\texttt{M:<ID}_{i+1}\texttt{>}$. 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`.

$\texttt{M:<ID}_1\texttt{>}$	$\texttt{= M:<delta}\ \delta(q,\gamma)\ \texttt{ID}_1\texttt{>}$
	$\texttt{= Apply\_delta:<}\delta(q,\gamma)\ \texttt{ID}_1\texttt{>}$
	$\texttt{= M:<ID}_2\texttt{>}$
	$\texttt{= M:<delta}\ \delta(q,\gamma)\ \texttt{ID}_2\texttt{>}$
	$\texttt{= Apply\_delta:<}\delta(q,\gamma)\ \texttt{ID}_2\texttt{>}$
	$\texttt{= M:<ID}_3\texttt{>}$
	$\vdots \quad \vdots$
	$\texttt{= M:<ID}_n\texttt{>}$
	$\texttt{= M:<delta undef ID}_n\texttt{>}$
	`= accept`

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

We now present the DATR translation of TM $M = (Q,$

```
#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 `<`$\mathtt{x}_0$ $\mathtt{x}_1$ $\mathtt{x}_2$ $\ldots$ $\mathtt{x}_{i-1}$ `; q ;`

The number of statements comprising node **Apply_delta** is $|Q| \times |\Gamma| \times |\{$

```
= M:< x x y y b ;
```

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% 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.
```