XSLT Version 2.0 is Turing-Complete: A Purely Transformation Based Proof

Ruhsan Onder and Zeki Bayram

Department of Computer Engineering/Internet Technologies Research Center
Eastern Mediterranean University
Famagusta, Cyprus
{ruhsan.onder, zeki.bayram}@emu.edu.tr
http://www.emu.edu.tr

Abstract. XSLT is a programming language, originally designed to convert XML documents to XHTML for presentation on browsers. XSLT works by matching predefined patterns to a source XML document and producing output for each kind of construct that is matched. In spite of its relatively humble goals, XSLT has the full power of a Turing machine, i.e. it is "Turing-complete." We show this is so by implementing an interpreter for a generic Turing machine in XSLT version 2.0. We use only the constructs available in the official specification of XSLT version 2.0 by the World Wide Web consortium, and no extensions to the core specification. Furthermore, we do not resort to string functions (which are also available in XSLT) but rather rely on the innate transformational capabilities of XSLT.

1 Introduction

Extensible Stylesheet Language Transformations (XSLT) was originally conceived as a way to transform data encoded as an XML document into an XHTML page that can be viewed on a user's browser. The motivation behind representing data in XML format was to separate content from presentation of the content, with XSLT acting as the bridge between the two. XSLT is also used to convert XML documents from one format to another, so that different applications can work on the same data, but in different formats.

XSLT works by transforming a source XML document through the application of templates. A template has two main parts: (1) a potentially complex pattern in the form of an XPath expression which specifies a part of the source document, and (2) the way in which the specified part will be transformed.

Recently, we showed that XSLT can be used as a "system level" tool, acting as an interpreter for a simple imperative language [1]. In this paper we show that XSLT can be used to implement the functionality of a universal Turing machine. For this purpose, we designed a Universal Turing Machine Emulating Stylesheet (which we shall call UTMES). UTMES takes as input an XML encoded definition of a Turing machine that accepts by final state, as well as the input for the Turing machine, and "runs" the Turing machine on the provided input.

Table 1. State transition table of a Turing machine that accepts strings of form 0^n1^n, $n \geq 1$

State\Symbol	0	1	X	Y	B
s0	(s1,X,R)	–	–	(s3,Y,R)	–
s1	(s1,0,R)	(s2,Y,L)	–	(s1,Y,R)	–
s2	(s2,0,L)	–	(s0,X,R)	(s2,Y,L)	–
s3	–	–	–	(s3,Y,L)	(s4,B,R)
s4	–	–	–	–	–

In computability theory, a programming language or any other logical system is called *Turing-complete* if it has a computational power equivalent to a universal Turing machine. In other words, the system and the universal Turing machine can emulate each other [2]. Since we are in effect emulating a universal Turing machine by our stylesheet, we are actually proving the Turing-completeness of XSLT. Our stylesheet makes use of only constructs that are available in the most recent working draft specification of XSLT version 2.0 [3], and hence our completeness result pertains to this specification.

The remainder of this paper is organized as follows. In section 2 the new facilities of XSLT version 2.0 are explained, which open the way for a straight-forward transformation oriented proof of the Turing-completeness of XSLT. Using UTMES to interpret/run Turing machines is explained in section 3. Section 4 contains the description of the implementation of UTMES. Related work and a comparative discussion with our work is given in section 5. In section 6 we have the conclusion. Appendix A contains the XSLT template that implements a single move of the Turing Machine, and Appendix B contains the trace of UTMES as it runs a Turing Machine on some input.

2 XSLT Version 2.0

A significant improvement of XSLT version 2.0 over XSLT version 1.0 [4] is the replacement of the concept of "result tree fragments" with the more versatile "temporary trees." In XSLT version 1.0 when we assign the return value of a template to a variable, a result tree fragment is created to hold the return value. However, result tree fragments are not re-usable in the stylesheet as regular nodes, since their contents cannot be pattern-matched through XPath expressions and consequently they cannot be transformed. Rather, they are treated as strings and string manipulation functions are needed to extract their contents. Alternatively, to be able to access and output the contents of a result tree fragment as a node set, extension functions of XSLT processors or parsers are required, such as the msxsl:node-set() function of MSXML [5], which by definition are non-standard.

Temporary trees, which are intrinsically part of the XSLT version 2.0 specification, have the full functionality of node sets. The return value of a template can be held in a variable as a temporary tree and its contents *can* be pattern-matched using XPath expressions. It is this feature of XSLT version 2.0 that allows us to use a pure transformation based approach in the implementation of UTMES.

```
<TM>
   <Spec>
      <StateSet>
         <state>s0</state>
         <state>s1</state>
         <state>s2</state>
         <state>s3</state>
         <state>s4</state>
      </StateSet>
      <InputSymbols>
         <Symbol>0</Symbol>
         <Symbol>1</Symbol>
      </InputSymbols>
      <TapeSymbols>
         <Symbol>0</Symbol>
         <Symbol>1</Symbol>
         <Symbol>X</Symbol>
         <Symbol>Y</Symbol>
         <Symbol>B</Symbol>
      </TapeSymbols>
      <StartState>s0</StartState>
      <BlankSymbol>B</BlankSymbol>
      <FinalStates>
         <FinalState>s4</FinalState>
      </FinalStates>
      <TransitionFunction>
         <Delta CurrentState="s0" read="0"
                NextState="s1" write="X" direction="R"/>
         <Delta CurrentState="s0" read="Y"
                NextState="s3" write="Y" direction="R"/>
         <Delta CurrentState="s1" read="0"
                NextState="s1" write="0" direction="R"/>
         <Delta CurrentState="s1" read="1"
                NextState="s2" write="Y" direction="L"/>
         <Delta CurrentState="s1" read="Y"
                NextState="s1" write="Y" direction="R"/>
         <Delta CurrentState="s2" read="0"
                NextState="s2" write="0" direction="L"/>
         <Delta CurrentState="s2" read="X"
                NextState="s0" write="X" direction="R"/>
         <Delta CurrentState="s2" read="Y"
                NextState="s2" write="Y" direction="L"/>
         <Delta CurrentState="s3" read="Y"
                NextState="s3" write="Y" direction="R"/>
         <Delta CurrentState="s3" read="B"
                NextState="s4" write="B" direction="R"/>
      </TransitionFunction>
   </Spec>
   <input>
      <Symbol>0</Symbol>
      <Symbol>0</Symbol>
      <Symbol>1</Symbol>
      <Symbol>1</Symbol>
   </input>
</TM>
```

Fig. 1. Specification for a Turing machine for accepting strings of form 0^n1^n, $n \geq 1$

```
<TM>
  <Tape>
     <TapeHead state="s0"/>
     <tapeEl>0</tapeEl>
     <tapeEl>0</tapeEl>
     <tapeEl>1</tapeEl>
     <tapeEl>1</tapeEl>
  </Tape>

  <Transitions>
     <Delta CurrentState="s0" read="0"
            NextState="s1" write="X" direction="R" />
     <Delta CurrentState="s0" read="Y"
            NextState="s3" write="Y" direction="R" />
     <Delta CurrentState="s1" read="0"
            NextState="s1" write="0" direction="R" />
     <Delta CurrentState="s1" read="1"
            NextState="s2" write="Y" direction="L" />
     <Delta CurrentState="s1" read="Y"
            NextState="s1" write="Y" direction="R" />
     <Delta CurrentState="s2" read="0"
            NextState="s2" write="0" direction="L" />
     <Delta CurrentState="s2" read="X"
            NextState="s0" write="X" direction="R" />
     <Delta CurrentState="s2" read="Y"
            NextState="s2" write="Y" direction="L" />
     <Delta CurrentState="s3" read="Y"
            NextState="s3" write="Y" direction="R" />
     <Delta CurrentState="s3" read="B"
            NextState="F" write="B" direction="R" />
  </Transitions>
</TM>
```

Fig. 2. Instantaneous description of the Turing machine before it starts execution

3 Running Turing Machines on the Universal Turing Machine Emulator Stylesheet UTMES

3.1 Format of the Turing Machines

UTMES is applied to an XML document containing the specification of a Turing machine, as well as the input to the Turing machine. Such an XML document for accepting strings of the form 0^n1^n, $n \geq 1$, is shown in Figure 1. The state transition table for the Turing machine in this document is given in Table 1 (taken from [6]). The specification is structured under a <TM> root which has <Spec> and <input> elements. The <Spec> element has the state set, input symbols, tape symbols, start state, Blank symbol, final states and transition function specifications as sub-elements. The transition

rules in the <TransitionFunction> element are enclosed in <Delta> elements, having @CurrentState, @read, @NextState, @write and @direction attributes, which indicate the current state, symbol read from tape, next state, the symbol to overwrite the symbol read and the direction that the tape head will move (left or right), respectively.

3.2 Execution of a Simple Turing Machine with UTMES

UTMES transforms the Turing machine specification, together with its initial input, into an internal format, which represents the instantaneous description (ID) [6] of the Turing machine in its initial configuration. The initial ID of the Turing machine of Figure 1 is shown in Figure 2. Execution then proceeds by successively generating a new ID, using the current ID and the transition function of the Turing machine.

The result of the execution of the Turing machine, given in Figure 1, on the provided input, is depicted in Appendix B.

4 Implementation of the Universal Turing Machine in XSLT

UTMES is implemented as an XSLT version 2.0 stylesheet. In this section, we describe its overall architecture and some of its implementation details.

4.1 Software Used

We used Java 1.4.2_08 [7] with Xalan Java 2.4.1 [8] as the XML processor for the execution of our examples.

4.2 Overview of the Implementation

We used two named templates, `Execute` and `ChangeTape` and various other unnamed templates to simulate the execution of a Turing machine on its input. First, unnamed templates create the initial ID of the Turing machine. The template `Execute` takes this initial ID, and produces consecutive ID's according to the transition rules of the Turing machine, calling the template `ChangeTape` to generate the next ID, given the current ID. The execution stops when a final state is reached.

4.3 Overview of the Execution Process

Figure 3 depicts the top-level templates of UTMES. After matching the `<TM>` element, a variable `$TuringM` is created to hold the initial ID, generated by the applied templates (lines 7-10). Then this variable is used as the temporary tree for the call to the template `Execute`. The result returned by this call, which is the the list of state transitions and rules applied during execution, is held in variable $tapeVar (lines 11-16), which in turn is used to print the output (lines 18-20).

```
1. <xsl:template match="/">
2.    <TM>
3.       <xsl:apply-templates select="TM"/>
4.    </TM>
5. </xsl:template>
6. <xsl:template match="TM">
7.     <xsl:variable name="TuringM">
8.          <xsl:apply-templates select="input"/>
9.            <xsl:apply-templates select="Spec"/>
10.    </xsl:variable>
11.    <xsl:variable name="tapeVar">
12.        <xsl:call-template name="Execute">
13.             <xsl:with-param name="tape"  select="$TuringM/Tape"/>
14.             <xsl:with-param name="rules" select=
                                          "$TuringM/Transitions"/>
15.        </xsl:call-template>
16.    </xsl:variable>
17.    State Transitions : <br/>
18.    <xsl:for-each select="$tapeVar/child::node()">
19.             <xsl:copy-of select="."/>
20.    </xsl:for-each>
21.</xsl:template>
```

Fig. 3. Top-Level Templates

4.4 Matching a Rule to the Current State

The recursive named template `Execute`, depicted in Figures 4 and 5, takes <`Tape`> and <`Transitions`> tree fragments in separate parameters from the main template which matches the `<TM>` element. It finds the transition rule which applies to the current ID (line 27) and sends it to the modifier template `ChangeTape`, together with the current configuration.

`Execute` implements a recursive algorithm whose basis step (lines 29-37) checks if the Turing machine stopped in a final state (i.e. the current state is final and there are no matching rules). This is in accordance with [6] (page 327). If this is the case, then the final state as well as the message “accepted” are returned to its caller. If there exists a matching rule (lines 38-61), then a variable `$tst` is created to hold the result of the call to the template `ChangeTape` (lines 50-56), to be subsequently used in the recursive call to `Execute`.

Variables `$Cstate`, `$TpHdPos`, `$currentSymb` and `$MatchRule` hold the current state, position of the tape head in the tape, the symbol that is currently scanned, and the rule that matches the current ID. Continually (at every call) `Execute` outputs the tape contents to its caller (lines 39-49).

```
1.  <xsl:template name="Execute">
2.       <xsl:param name="tape"/>
3.       <xsl:param name="rules"/>
4.       <xsl:variable name="Cstate">
5.            <xsl:value-of select=
                     "$tape/child::*[name()='TapeHead']/@state"/>
6.       </xsl:variable>
7.       <xsl:value-of select="'TAPE :  |'"/>
8.       <xsl:for-each select="$tape/child::*">
9.            <xsl:if test="name()='TapeHead'">
10.               <xsl:value-of select="name()"/>
11.               <xsl:value-of select="' '"/>
12.               <xsl:value-of select="@state"/>
13.           </xsl:if>
14.          <xsl:value-of select="."/>
15.          <xsl:value-of select="'|'"/>
16.      </xsl:for-each> <br/>
17.      <xsl:variable name="TpHdPos">
18.          <xsl:for-each select="$tape/child::*">
19.               <xsl:if test="name()='TapeHead'">
20.                    <xsl:value-of select="position()"/>
21.               </xsl:if>
22.          </xsl:for-each>
23.      </xsl:variable>
24.      <xsl:variable name="currentSymb">
25.          <xsl:value-of select="$tape/child::*[$TpHdPos+1]"/>
26.      </xsl:variable>
```

Fig. 4. The template "Execute" which recursively applies the transitions to the tape (part 1)

4.5 Implementing a one-step move of the Turing Machine

Although the procedure for generating the next ID from the current ID is simple in principle, there are special cases to consider when the tape head is at either end of the tape, which complicates the situation.

In the following discussion, "tape" refers to the `<tape>..</tape>` element, "tape head" refers to the `<TapeHead state=".."/>` element, and "symbol" refers to `<tapeEl>..</tapeEl>` elements in the XML document.

The template `ChangeTape`, given in Appendix A, takes the matching rule (in parameter `$MRule`), and the current ID (in parameters `$tape`, `$HdPos`) as inputs. `$tape` holds the current contents of the Turing machine's tape, `$HdPos` holds the position of the tape head and `$MRule` holds the the matched rule. It applies the matching rule to the current ID to produce the new ID.

Moving Right If the applicable transition rule dictates a right move, then all the tape symbols up to the tape head are copied as they are. In normal operation, the symbol which comes directly after the tape head is replaced by the symbol indicated by the

```
27.      <xsl:variable
            name="MatchRule" select=
                    "$rules/child::*[./@CurrentState=$Cstate and
                                      ./@read=$currentSymb]"/>
28.           <xsl:choose>
29.               <xsl:when test="count($MatchRule)=0">
30.                    <br/><xsl:value-of select="'FINALS: '"/>
31.                    <xsl:for-each
                        select="/TM/Spec/FinalStates/child::*">
32.                        <xsl:value-of select="."/>
33.                        <xsl:if test="$tape/child::*[name()=
                                        'TapeHead']/@state=.">
34.                             <br/>accepted
35.                        </xsl:if>
36.                    </xsl:for-each>
37.               </xsl:when>
38.               <xsl:otherwise>
39.                  <xsl:value-of select="'CurrentState:'"/>
40.                  <xsl:value-of select=
                                "$MatchRule/@CurrentState"/>
41.                  <xsl:value-of select="'   read: '"/>
42.                  <xsl:value-of select="$MatchRule/@read"/>
43.                  <xsl:value-of select="'   NextState:'"/>
44.                  <xsl:value-of
                              select="$MatchRule/@NextState"/>
45.                  <xsl:value-of select="'   write: '"/>
46.                  <xsl:value-of select="$MatchRule/@write"/>
47.                  <xsl:value-of select="'   direction: '"/>
48.                  <xsl:value-of select="$MatchRule/@direction"/>
49.                  <br/><br/>

50.                  <xsl:variable name="tst">
51.                     <xsl:call-template name="ChangeTape">
52.                        <xsl:with-param name="tape" select="$tape"/>
53.                        <xsl:with-param name="MRule"
                                             select="$MatchRule"/>
54.                        <xsl:with-param name="HdPos"
                                               select="$TpHdPos"/>
55.                     </xsl:call-template>
56.                  </xsl:variable>

57.                  <xsl:call-template name="Execute">
58.                     <xsl:with-param name="tape" select="$tst/Tape"/>
59.                     <xsl:with-param name="rules" select="$rules"/>
60.                  </xsl:call-template>
61.              </xsl:otherwise>
62.           </xsl:choose>
63.</xsl:template>
```

Fig. 5. The template “Execute” (part 2)

transition rule, the tape head "moves right," and all subsequent symbols are copied as they are. Special cases arise if the tape head writes a blank to the leftmost position of the tape, or if the tape head moves past the last symbol on the tape.

Blanks that come before the leftmost non-blank symbol are not explicitly represented. On the other hand, a Blank symbol is added at the end, if the tape head moves right past the last existing symbol, since the tape head needs to be scanning a symbol at all times.

Moving Left If a left move is specified by the matching rule, then again all the symbols up to the tape head are copied as they are. Under normal circumstances, the symbol that is currently scanned by the tape head is replaced by the symbol dicated by the applicable transition rule, the head "moves left," by changing places with the symbol on its left, and the remaining symbols that come after the replaced symbol are copied as they are. If the tape head was pointing at the first symbol on the tape before the move, a Blank symbol is inserted after the tape head (i.e. the first of the implicit Blanks on the left side of the tape become explicit). If the tape head was pointing to the last symbol on the tape, and the symbol to write is a Blank then the last symbol is removed (a trailing Blank is not represented explicitly, unless it is directly being scanned by the tape head).

The full code of our implementation, as well as sample Turing machines, are available at [9].

5 Related Work

Our literature search on proving the Turing-completeness of XSLT revealed several other approaches, although all of them deal with XSLT version 1.0 rather than version 2.0.

Kepser [10] showed the Turing-completeness of XSLT version 1.0 and XQuery by coding *mu-recursive* functions in XSLT, which are themselves Turing-complete. *mu-recursive* functions are inductively defined based on a set of functions that always return zero, a set of functions that select a component out of a tuple and the successor function which adds 1 to any given natural number. These functions are combined to obtain more complex functions using composition, primitive recursion and mu-recursion. This approach is far from being straightforward, and relies on the Turing-completeness of mu-recursive functions. Our approach, in contrast, is much more straightforward, in that we directly implemented a universal Turing machine emulator.

Lyons [11] implemented a universal Turing machine in XSLT version 1.0 which takes the initial tape as an input *string* parameter from the command line at execution time. Then, the input tape, *represented as a string* is manipulated according to the rules of the input Turing machine *using string manipulation functions* of XSLT. Although there is nothing wrong in principle with using string functions, we believe they do not represent the core essence and nature of XSLT, which is based on transformations. Our work differs significantly from his: we exclusively use transformations for the computation and pure XML for representing states of the computation.

Korlyukov [12] used the transformation oriented approach with recursive templates like we do, but made use of the result tree fragments of XSLT 1.0 together, with non-

standard features (specifically the *node-set()* function) of the XML processor *XT* developed by [13]. He used a different XML syntax for the specification of input Turing machines and tape contents. His implementation does not however follow the textbook concept of an "instantaneous description," and his work has not been published in literature.

Our observation is that our work is the only one so far that shows the Turing-completeness of the official XSLT version 2.0.

6 Conclusion

We showed the Turing-completeness of XSLT version 2.0, which provides features for storing and accessing result of intermediate computations as regular node sets. We developed an XSLT stylesheet, called UTMES, that emulates a universal Turing machine using only native XSL transformations (as opposed to non-standard features or string manipulation functions) by exploiting the temporary tree concept introduced by XSLT version 2.0. The input to UTMES is a Turing machine, together with the initial tape, encoded as an XML document. UTMES then "executes" the given Turing machine on the given initial tape.

XSLT version 2.0 is very new, and we believe this is the first demonstration of its Turing-completeness.

References

1. Onder, R., Bayram, Z.: Interpreting imperative programming languages in XSLT. In: Proceedings of the Ninth IASTED International conference on Internet and Multimedia Systems and Applications (EuroIMSA2005), IASTED (2005) 131–136
2. Brainerd, W., Landweber, L.: Theory of Computation. Wiley (1974)
3. Michael Kay (Editor): XSL transformations (XSLT) version 2.0 of W3C working draft. Available at http://www.w3.org/TR/xslt20 (2005)
4. James Clark (Editor): XSL transformations (XSLT) version 1.0 of W3C working draft. Available at http://www.w3.org/TR/xslt (1999)
5. MSXML Documentation Team: Microsoft XML core services (MSXML) 1.0. Available at http://www.nedcomp.nl/support/origdocs/xml4/ (1998)
6. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (2001)
7. Java 2 Developer Team: Java 2 SDK, standard edition documentation version 1.4.2. Available at http://java.sun.com/j2se/1.4.2 (2005)
8. Java 2 Developer Team: Java API for XML processing (JAXP). Available at http://java.sun.com/xml/jaxp/index.jsp (2005)
9. Onder, R., Bayram, Z.: Universal Turing Machine Emulator Stylesheet UTMES. Available at http://itrc.emu.edu.tr (2006)
10. Kepser, S.: A simple proof for the Turing-completeness of XSLT and XQuery. In: Proceedings of the Extreme Markup Languages, Montreal, Quebec. (2004)
11. Lyons, B.: Universal Turing machine in XSLT. Available at http://www.unidex.com/turing/utm.htm (2001)
12. Korlyukov, A.: Turing machine. Available at http://www.refal.net/ korlukov/tm/ (2001)
13. Clark, J.: XT: an XSLT processor in Java. Available at http://www.blnz.com/xt/index.html (2002)

APPENDIX

A Code of the Template Applying the Appropriate Rule to the Current State

```
1.  <xsl:template name="ChangeTape">

2.     <xsl:param name="tape"/>
3.     <xsl:param name="MRule"/>
4.     <xsl:param name="HdPos"/>

5.    <xsl:element name="Tape">
6.        <xsl:choose>
7.            <xsl:when test="$MRule/@direction='R'">
8.                  <xsl:for-each
                         select="$tape/child::*[position() &lt; $HdPos]">
9.                      <xsl:copy-of select="."/>
10.                 </xsl:for-each>

11.               <xsl:if
                   test="count($tape/child::*[position() &lt; $HdPos])!=0
                                                 or $MRule/@write!='B'">
12.                   <xsl:element name="tapeEl">
13.                      <xsl:value-of select="$MRule/@write"/>
14.                   </xsl:element>
15.               </xsl:if>

16.               <xsl:element name="TapeHead">
17.                   <xsl:attribute name="state">
18.                       <xsl:value-of select="$MRule/@NextState"/>
19.                   </xsl:attribute>
20.               </xsl:element>

21.               <xsl:if
                     test="count($tape/child::*[position() = $HdPos + 1]/
                                                 following-sibling::*)=0">
22.                    <xsl:element name="tapeEl">
23.                         <xsl:value-of select="'B'"/>
24.                    </xsl:element>
25.               </xsl:if>

26.               <xsl:for-each
                     select="$tape/child::*[position() &gt; $HdPos + 1]">
27.                    <xsl:copy-of select="."/>
28.               </xsl:for-each>
29.           </xsl:when>

30.           <xsl:otherwise>
```

```
31.              <xsl:for-each
                    select="$tape/child::*[position() &lt; $HdPos - 1]">
32.                    <xsl:copy-of select="."/>
33.              </xsl:for-each>

34.              <xsl:element name="TapeHead">
35.                  <xsl:attribute name="state">
36.                      <xsl:value-of select="$MRule/@NextState"/>
37.                  </xsl:attribute>
38.              </xsl:element>


39.              <xsl:if
                    test="count($tape/child::*[position() = $HdPos + 1]/
                           preceding-sibling::*[name()!='TapeHead'])=0">
40.                   <xsl:element name="tapeEl">
41.                        <xsl:value-of select="'B'"/>

42.                   </xsl:element>
43.              </xsl:if>

44.              <xsl:for-each
                        select="$tape/child::*[position()=$HdPos - 1]">
45.                  <xsl:copy-of select="."/>
46.              </xsl:for-each>

47.              <xsl:if
                  test="count($tape/child::*[position() &gt; $HdPos+1])!=0
                                                  or $MRule/@write!='B'">
48.                   <xsl:element name="tapeEl">
49.                        <xsl:value-of select="$MRule/@write"/>
50.                   </xsl:element>
51.              </xsl:if>

52.              <xsl:for-each
                    select="$tape/child::*[position() &gt; $HdPos + 1]">
53.                   <xsl:copy-of select="."/>
54.              </xsl:for-each>
55.           </xsl:otherwise>
56.         </xsl:choose>
57.    </xsl:element>
58. </xsl:template>
```

B Execution Trace of the Turing Machine given in Figure 1 on its input

```
State Transitions :

TAPE : |TapeHead s0|0|0|1|1| CurrentState: s0 read: 0 NextState:
s1 write: X direction: R

TAPE : |X|TapeHead s1|0|1|1| CurrentState: s1 read: 0 NextState:
s1 write: 0 direction: R

TAPE : |X|0|TapeHead s1|1|1| CurrentState: s1 read: 1 NextState:
s2 write: Y direction: L

TAPE : |X|TapeHead s2|0|Y|1| CurrentState: s2 read: 0 NextState:
s2 write: 0 direction: L

TAPE : |TapeHead s2|X|0|Y|1| CurrentState: s2 read: X NextState:
s0 write: X direction: R

TAPE : |X|TapeHead s0|0|Y|1| CurrentState: s0 read: 0 NextState:
s1 write: X direction: R

TAPE : |X|X|TapeHead s1|Y|1| CurrentState: s1 read: Y NextState:
s1 write: Y direction: R

TAPE : |X|X|Y|TapeHead s1|1| CurrentState: s1 read: 1 NextState:
s2 write: Y direction: L

TAPE : |X|X|TapeHead s2|Y|Y| CurrentState: s2 read: Y NextState:
s2 write: Y direction: L

TAPE : |X|TapeHead s2|X|Y|Y| CurrentState: s2 read: X NextState:
s0 write: X direction: R

TAPE : |X|X|TapeHead s0|Y|Y| CurrentState: s0 read: Y NextState:
s3 write: Y direction: R

TAPE : |X|X|Y|TapeHead s3|Y| CurrentState: s3 read: Y NextState:
s3 write: Y direction: R

TAPE : |X|X|Y|Y|TapeHead s3|B| CurrentState: s3 read: B
NextState: s4 write: B direction: R

TAPE : |X|X|Y|Y|B|TapeHead s4|B|

FINALS: s4 accepted
```