Supercompilation of Double Interpretation (How One Hour of the Machine's Time Can Be Turned to One Second)

Aliaksandr Karliukou
Department of Mathematics,
Yanka Kupala State University of Grodno,
22 Ozheshko St., Grodno, 230023, Belarus
+375-(152)-443487
korlyukov@grsu.grodno.by

Andrei P. Nemytykh
State Key Lab of Software Engineering,
Wuhan University, Wuhan,
Hubei, 430072, China
+86-(27)-87682438
nemytykh@whu.edu.cn
and
Program Systems Institute RAS,
Pereslavl-Zalessky,
Yaroslavl region, 152140, Russia
+7-(08535)-98024
nemytykh@math.botik.ru

ABSTRACT

Supercompilation is a program transformation method that can achieve partial evaluation, and in some respect more powerful. In this paper, we describe some experiments with the supercompiler Scp4. We specialize an XSLT-interpreter w.r.t. a Turing Machine interpreter written in XSLT. The running time speedup observed by us is entered in the title of this paper.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Language Constructs and Features – *formal definitions, control structures.*

F.3.2 [**Logic and Meaning of Programs**]: Specifying, Verifying and Reasoning About Programs.

General Terms

Algorithms, Measurement, Performance, Experimentation, Languages, Theory.

Keywords

Automatic program transformation, Supercompilation, Specialization, XML, XSLT, Functional programming languages, Refal.

1. INTRODUCTION

This paper concerns automatic program transformation with the object of running time optimization. We use the supercompiler Scp4 (see [17],[18]) as the transformer. The subjects to be transformed with the supercompiler Scp4 are programs written in a functional programming language Refal (see [26],[31],[23]), a brief introduction to which is given below. The output language of Scp4 is Refal as well. The supercompiling procedure is enough good for processing of interpreting algorithms. The aim of our work is to demonstrate some abilities of the supercompiler Scp4 (see [17],[18]).

The self-applicability of any program transformer is very attractive (see [3]). Here is a step to solve this task (with the supercompiler Scp4): we consider an example of the supercompilation of a double interpretation. The running time speedup observed by us is entered in the title of this paper.

Let us formulate the task. We deal with a number of subjects relating one to another. Following after Turchin ([28],[29],[30],[15]), we denote an abstract functional call with the angular brackets; the left angular bracket is followed by the function name.

1. A Turing Machine takes a **tape** as its argument, let us denote that as **<TM e.tape>**
2. An interpreter **IntTM** of the Turing Machine written in XSLT ([33],[34]) takes an XML document ("written on the tape") and a DTD ([33],[4]) as its arguments, let us denote that as **<IntTM (e.DTD) (e.XML)>** or in more details:

```
<IntTM (e.DTD)      e.tape       >
                <TM  !     >
```

We carried the call to a concrete Turing Machine over to the next line below to emphasize that the subject to be transformed by the interpreter is this call itself rather then its value. The exclamation mark stresses the semantics of the tape: the tape is processed by the concrete Turing Machine, even though indirectly by way of the interpreter of the Turing Machine; a concrete Turing Machine is able to process an arbitrary concrete tape.

3. Let **IntXSLT** be an interpreter of an algorithmically universal subset of the language XSLT (see the section No. 4 below) written in Refal (see the subsection No. 2). It takes an XSLT-program, an XML-document, a DTD-definition as its inputs, let us denote that as

   <IntXSLT (e.XSLT) (e.DTD) (e.XML)>

   or in more details:

```
<IntXSLT                e.DTD            e.tape    >
         <IntTM      (  !        )          !      >
                                  <TM      !    >
```

4. Let us formulate a task for specialization (by the supercompiler Scp4) of the interpreter IntXSLT with respect to the interpreter IntTM, which, in its turn, performs the concrete Turing Machine TM. The tape of the Turing Machine is defined as an XML-document, which, before to be transformed, is syntactically checked with a validator DTD ([33]). Let us denote that as

```
<SCP4                                                >
      <IntXSLT                        e.tape      >
               <IntTM <DTD            !        > >
                              <TM  !      >
```

   The DTD-description is used as a filter by us (that is to say, a recursive dynamic typing with means of the language Refal) for the input XML-documents.

Thus in this example the supercompiler deals with the five subjects mentioned above. If the following task will be formulated for the supercompiler

```
<IntXSLT                       e.TM  e.tape                 >
         <IntTM <DTD        !       !                >>
                           <  !       !          >
```

Figure 1.

then the output of the supercompiler will an interpreter of the Turing Machine written in Refal. It is a substantially interesting question about the running time efficiency of this result program-interpreter. It is also interesting to investigate the structure of the obtained program.

While formulation of the scheme

```
<IntXSLT                          e.tape             >
         <IntTM <DTD              !             >>
                        <TM      !       >
```

Figure 2.

provides a concrete Refal-program. For example, the program TMDoublePQ doubling characters **P** and replacing them with **Q**s, or the program TMPQ replacing **P**s with **Q**s.

1.1 The language Refal

The programming language Refal (by V.F. Turchin) is a first-order functional language with an applicative (inside-out) semantics. Roughly speaking, a program in Refal is a term rewriting system. The semantics of Refal is based on pattern-matching. As usually, the rewriting rules are ordered to match from the top to the bottom. The terms are generated with two constructors. The first is the concatenation. It is binary, associative and is used in infix notation, which allows us to drop its parenthesis. In Refal the blank is used to denote the concatenation. The second constructor is unary. It is syntactically denoted with just its parenthesis (that is without a name). Angular brackets are used to denote a function call. Its name is put after the left bracket. Every function is unary. In Refal the ground terms are referred to as expressions. Empty sequence is a special basic ground term. This term is denoted with nothing and called "empty expression". It is neutral element (both left and right) of the concatenation. All other basic ground terms are named as "symbols". There exist three types of basic non-ground terms (called variables) - *e.name, s.name* and *t.name*. An *e*-variable can take any expression as its value, an *s*-variable can take any symbol as its value, a *t*-variable -- any symbol and any expression enclosed with the parenthesis. The associativity of the concatenation causes the set of Refal terms to be more expressive than the set of Lisp terms.

Example:

```
$ENTRY Go {
= <Search  (Valentin Turchin)
   ((Alanzo Church Lambda-calculus)
    (Andrei Markov Markov-algorithm) (John McCarthy Lisp)
    (Emil Post Post-system) (Guy Steel Scheme)
    (Valentin Turchin Refal)  (Alan Turing Turing-machine))
    >;
}


Search {
   (s.key1 s.key2) ((s.key1 s.key2 e.value) e.table) = (e.value);
   (s.key1 s.key2) ((e.row) e.table)
                                  = <Search (s.key1 s.key2) (e.table)>;
}
```

The result of the program is the following Refal-expression: **(Refal)**. On the left hand side of the function **Go** we see the empty expression. The right side of the function **Go**, the left side of the first sentence and the both sides of the second sentence of the function **Search** show the associativity of the concatenation.

A detailed description of the language is available in an electronic format ([26]).

All residual programs from our paper were constructed automatically by Scp4 and modified by hand-formatting only.

2. TURING MACHINE

In the theory of algorithms the concept of Turing Machine (TM) is broadly used as a precise equivalent of our intuitive idea of an algorithm (see, for example, [14]). The memory of a TM, called *tape*, is an infinite chain of cells (in the both sides) with two adjacent neighbors each. A program for the TM is a finite sequence of instructions. Each instruction has the form:

CurrState CurrSymb NextSymb NextState Movement

Accordingly to the program, a pointer is being moved from one cell to its adjacent neighbor by the TM, this cell's content is changed as well as the current state of the machine. The start state is called '*start'* and the final is called *'stop'*. The input to the TM is a finite array of rightward cells following blanks and followed by blanks (denote the blank with ***B***).

Consider an **example** of a program for the TM. The program *DoublePQ* replaces an array of the characters **P** with another array, where each of the **P**s was changed with two **Q**s (one per cell). If, for instance, in the beginning of the TM's job there are exactly ten **P**s on the tape (followed one by another) and the pointer indicates to the first of them, then, when the machine will stop, the tape will contain twenty **Q**s followed one after another.

Table 1. The program DoublePQ

CurrState	start	start	Start	moveleft	moveleft
CurrSymb	***B***	**Q**	**P**	**Q**	***B***
NextSymb	***B***	**Q**	**Q**	**Q**	**Q**
NextState	stop	start	moveleft	moveleft	Start
Movement	right	right	Left	left	right

As a first experiment we supercompile an interpreter of the TM. The interpreter is written in Refal-5 (see [26],[31]):

Table 2. Interpreter of the Turing Machine written in Refal

```
* Call for a concrete Turing machine ( a program ).
$ENTRY Go {
 (e.LeftTape) (s.CurrSymb) (e.RightTape) =
   <Turing (
    (start      B   B     stop              right)
    (start      Q   Q     start             right)
    (start      P   Q     moveleft          left )
    (moveleft   Q   Q     moveleft          left )
    (moveleft   B   Q     start             right) )
   (start)  (e.LeftTape) (s.CurrSymb) (e.RightTape) >;
 }

* The interpreter itself.
* <Turing (e.Program) (s.CurrState)(e.LeftPartOfTape)
*                            (s.CurrSymb)(e.RightPartOfTape)>
Turing  {
  (e.instr) (stop) (e.left) (s.symbol) (e.right)
            = (e.left) (s.symbol) (e.right) ;

  (e.instr) (s.q) (e.left) (s.symbol) (e.right)
            = <Turing (e.instr) <Turing1
                                   <Search (s.q s.symbol) (e.instr)>
                                    (e.left) (s.symbol) (e.right)>  >;
 }

Turing1   {
 (s.c s.r left) ( ) (s.symbol) (e.right) = (s.r) ( ) (B) (s.c e.right) ;
 (s.c s.r left) (e.left s.a) (s.symbol) (e.right)
                   = (s.r) (e.left) (s.a) (s.c e.right) ;
 (s.c s.r right) (e.left) (s.symbol) ( ) = (s.r) (e.left s.c) (B) ( ) ;
 (s.c s.r right) (e.left) (s.symbol) (s.a e.right)
                   = (s.r) (e.left s.c) (s.a) (e.right) ;
 }
```

Where the definition of the function **Search** can be found in the Introduction.

The entry point **Go** has fixed a context to specialize this interpreter *Turing* w.r.t. the given TM. The result of the supercomilation is the following program:

Table 3. The program DoublePQ translated into Refal

```
* InputFormat:    <Go (e.LeftTape)(s.CurrSymb) (e.RightTape)>
$ENTRY Go {
 (e.Left) (s.Symbol) (e.Right) = <F6 (e.Left) s.Symbol e.Right> ;
}

F59 {
 ()          ()          Q    =    (Q Q Q B) (B) () ;
 ()          (s.3 e.4)   Q    =    <F6 (Q Q Q) s.3 e.4 > ;
 (e.1 s.5)   (e.4)       Q    =    <F59 (e.1) (Q e.4) s.5 > ;
 (e.1)       ()          B    =    (e.1 Q Q B) (B) () ;
 (e.1)       (s.3 e.4)   B    =    <F6 (e.1 Q Q) s.3 e.4 > ;
}

F6 {
 (e.1)       B                =    (e.1 B) (B) () ;
 (e.1)       B     s.5 e.4    =    (e.1 B) (s.5) (e.4) ;
 (e.1)       Q                =    (e.1 Q B) (B) () ;
 (e.1)       Q     s.5 e.4    =    <F6 (e.1 Q) s.5 e.4 > ;
 ()          P                =    (Q Q B) (B) () ;
 ()          P     s.5 e.4    =    <F6 (Q Q) s.5 e.4 > ;
 (e.1 s.6)   P     e.4        =    <F59 (e.1) (e.4) s.6 > ;
}
```

For our experiment, to see the speedup, we input 512 **P**s to the given TM.

The supercompiler Scp4 transforms programs in the dialect Refal-5 ([26],[31]). This paper deals just with a fragment of the dialect, called strict Refal, in which: 1) open *e*-variables and repeated *t*- and e-variables are not allowed in the patterns ([26]); 2) the left side of each sentence is a pattern. Evaluation of a strict Refal-program can be seen as a sequence of elementary actions called Refal-steps ([26]). Running time of a Refal-step, generally speaking, is not uniformly bounded on input data. A concrete display of this non-bounding depends on a given implementation of the Refal Machine. Everywhere below, we mean the strict Refal and a fixed release of Refal-5 ([31]). Under these conditions, running time of one Refal-step, corresponding to a Refal sentence, is uniformly bounded if the sentence does not contain repeated occurrences of *t*- and *e*-variable in its right side. The inverse statement is not correct. Running time of every step of the program from [Table 2] is uniformly bounded on input data, but if the function **Turing** will be declared as the program entry point, then steps corresponding to the second sentence of this function have not such property.

The original program takes 2 634 778 Refal-steps (15.242 seconds) for running, the result of the supercompilation takes 525 319 Refal-steps (2.053 seconds). Thus, the Refal-step speedup *StepSpeedup* is 5.016 and the run-time speedup *TimeSpeedup* is 7.388 (for this example and this input data). All our experiments were performed under the operating system Windows2000 with Intel Pentium-3 CPU, 262 144 KB RAM, 500 MHz.

Consider the residual program. Many sentences handle the borders of the tape (where the blanks are starting). With the theoretical point of view, the tape is infinite in the both sides, but practically it is finite, and we can either add the empty cells by need or assume that always there exist enough many of the empty cells. In the second case all programs look shorter. Further we consider namely this simpler case and note what happens in the general case.

Let us remove the first and the second sentences in the definition of **Turing1** (from the original program), then after the supercompilation we have the following program, which is easier to be followed:

Table 4. The simpler residual program DoublePQ (in Refal)

```
* InputFormat:  <Go (e.LeftTape)(s.CurrSymb) (e.RightTape)>
$ENTRY Go {
 (e.Left) (s.Symbol) (e.Right)  = <F6 (e.Left) s.Symbol e.Right > ;
}

F27 {
 (e.1 s.2)    (e.4)      Q    =    <F27 (e.1) (Q e.4) s.2 > ;
 (e.1)        (s.3 e.4)  B    =    <F6 (e.1 Q Q) s.3 e.4 > ;
}

F6 {
 (e.1)        B    s.3 e.4    =    (e.1 B) (s.3) (e.4) ;
 (e.1)        Q    s.3 e.4    =    <F6 (e.1 Q) s.3 e.4 > ;
 (e.1 s.2)    P        e.4    =    <F27 (e.1) (e.4) s.2 > ;
}
```

The things became clear: the function **F6** corresponds to the inner state *start*, while the function **F27** corresponds to the inner state *moveleft*. Each sentence corresponds to one instruction of the TM. The number of the Refal-steps is equal to the number of the TM's steps. The residual program contains no the terms: *start, moveleft, stop*. That output of the supercompiler is an optimal Refal-program, i.e. a compilation of the program for the TM ([Table 1]) into a Refal-program ([Table 4]) took place. Here, under the "optimalness" we mean an optimalness within the limits of the input algorithm defined with the original program: no remarkable changes of this algorithm (in its essence) have happened.

3. THE DECLARATIVE LANGUAGE XSLT FOR TRANSFORMING DOCUMENTS: EXPERIMENTS WITH THE SUPERCOMPILER SCP4

3.1 The Languages XML, DTD, XSLT

XML describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them (see [33],[32]). XSLT is designed to describe transformations of the XML documents [34]. The pair (XSLT, XML) is a programming language. This language requires typing of the data: the domain of a given program is being described in the language DTD [33].

From point of view of supercompilation, the given language seems an interesting object through the following reasons. On the one hand, this is a real language for transforming the internet-documents, which, eventually, will be broadly disseminated. On the other hand, data of this language is very similar to Refal-data. At last, the language XSLT for transforming of the documents is enough poor in its expressive means and stimulates users to develop programs passing the only time along the data (that is to say, once transformed part of the data is not available for further processings). The last circumstance simplifies the supercompiling procedure, and provides possibility for obtaining more efficient (in running time) residual programs.

Stress the previous remark is also true for a program transformation technique called specialization [5],[6]. The customary separation of data on *static* and *dynamic* ones (known and unknown in transforming time) (see [6],[7],[21]), from a point of view of the transforming procedure, could be made more accurate: on static, dynamic ones being single-passed, dynamic ones being twice-passed and so on. Interesting transformation of the algorithms (in "essence") can be achieved only by tools for analyzing and transforming the poly-processing of the data. The supercompiler Scp4 has a number of such simple tools (see [18]).

The below-considered interpreter of a TM, certainly, has the poly-processing tape. Refal-5 is the input and the output language for the supercompiler Scp4. Therefore, XSLT-programs are transformed indirectly through an interpreter of XSLT written in Refal-5. And, hence, on the output we obtain a program written in Refal-5 as well.

Finally, we note the tools analyzing the poly-processing make possibilities for dynamic typing of the data in supercompile-time. This information can lead to some additional transformations. The requirement to describe the domain of an XSLT-program (by means of DTD) makes such typing natural.

The language DTD represents an extended variant of the language of the regular expressions [22],[33].

3.2 The Regular Expressions

XML provides a mechanism, the document type declaration (DTD), to define constraints on the logical structure and to support the use of predefined storage units. An XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it. An XSLT transformer uses the declarations for checking documents. We implemented the following variant of the language DTD. (The double "or"-sign || is a meta-symbol, while the single | is just a regular character.)

Table 5. Variant of the language DTD

ElementDec ::= <!ELEMENT Name C*ontentSpec >*

```
ContentSpec  ::= EMPTY || Children*
Children ::= Choice || Seq
Choice ::= (Cp OrCp* )
Seq      ::= (Cp AndCp* )
Cp       ::= #PCDATA || CpAlt*
CpAlt  ::= Name || Choice || Seq
OrCp  ::= |Cp
AndCp  ::= ,Cp

AttlistDecl   ::= <!ATTLIST Name AttDef* >
AttlistDecl   -- can be encountered just one time.
AttDef        ::= Name AttType DefaultDecl
AttType       ::= CDATA
DefaultDecl ::= #IMPLIED || IgnoredDecl
IgnoredDecl – Any declaration. That is interpreted as #IMPLIED.

EntityDecl   ::= <!ENTITY % Name EntityValue >
EntityValue     ::= Dquote PEReference* Dquote
                    || Quote PEReference* Quote
Dquote :: = "
Quote :: = '
PEReference ::= %Name ;
```

For the undefined conceptions we refer the reader to the specifications of the language DTD (see [33]). We use the following DTD defining the structure of the tape and programs for the below-defined TM written in XSLT.

Table 6. Document type declaration for the TM

```
<!ELEMENT Go (Instruction, State, LeftTape, Symbol,
                                                RightTape)>
<!ELEMENT LeftTape  (Nod) >
<!ELEMENT RightTape (Nod) >
<!ELEMENT Nod ((Cell, Nod) | Cell) >
<!ELEMENT Symbol    (#PCDATA) >
<!ELEMENT Cell      (#PCDATA) >
<!ELEMENT State     (#PCDATA) >
<!ELEMENT Instruction (Instruction | EMPTY) >
<!ATTLIST Instruction
                  CurrState     CDATA     #IMPLIED
                  CurrSymb      CDATA     #IMPLIED
                  NextSymb      CDATA     #IMPLIED
                  NextState     CDATA     #IMPLIED
                  Movement      CDATA     #IMPLIED >
```

Here *Instruction* is a sequence of the instructions for a concrete TM, *State* is a current inner state, *LeftTape* is the left part of the tape followed by the current cell, *Symbol* denotes a symbol written in the current cell, *RightTape* denotes the right part of the tape.

It is naturally to define a tape for a TM just as a sequence of the cells (*Cells*). We refused such representation by the following two reasons. By definition, all TM's actions take place locally around a current cell. The language XSLT does not provide a simple possibility to rewrite a tail of the tape; one is forced to rewrite the tail by symbol-wise, that leads to inefficiency. This is one of the reasons why we write out (in the XML-document) one symbol and a reference to the tail. The second reason is, by the essence of the interpretation, it is frequently necessary to compare contents of the cells from different places. The instruction **<xsl:for-each . . . >** moves us inside a sub-tree and the other parts of the whole tree become either unavailable or available with a complicated way. By the same reasons, the TM's instructions are written out analogically.

Above we mentioned the DTD is used for checking of the logical structure of the input XML-document. On the other hand, this knowledge can help to develop more efficient programs. In one-level programming (without usage of a program transformer), it is not in the least obviously and simply to use this information for decreasing of running time of an algorithm. It is a paradox, but the supercompiler is able to implement that easy and simple. That is caused with that the supercompiler successfully processes composition of functions and, by definition, uses its own discretion to extend domains of the partial functions. We use a program-filter, which is the identity partial function on the documents corresponding to a given DTD and forcing an abnormal stop of the machine on the other documents.

3.3 An Interpreter of a Fragment of XSLT Written in Refal and Supercompilation of It

This paper concerns supercompilation of double interpretation, i.e. when one interpreter interprets another interpreter interpreting a given program. In this section we consider the first interpreter, the second will be considered in the following section.

An interpreter **VT.ref** (see [11]) of a fragment of XSLT was written in the language Refal-5 [26],[31]. The aim of our work is a demonstration of the possibility of the program transformer Scp4 [17],[18] to work as a compiler from some language into Refal-5 (see [1],[3]) and to generate an efficient program in Refal-5. The speedup of the generated program w.r.t. interpretation of its pro-image is quite remarkable.

The interpreter **VT.ref** accepts (as its input) programs in a small fragment of XSLT closed with the following syntactic constructions

Table 7. Fragment of the language XSLT.

```
<xsl:apply-templates  select = qexpression >
</xsl:apply-templates>
<xsl:call-template   name = NAME>
  <!-- Content: xsl:with-param? -->
</xsl:call-template>
<xsl:with-param name = NAME >  <!-- Content: template -->
</xsl:with-param>
<xsl:choose>  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
<xsl:when   test = boolean-expression> <!-- Content: template -->
</xsl:when>
<xsl:otherwise>  <!-- Content: template --> </xsl:otherwise>
<xsl:copy-of select = qexpression />
<xsl:element name = NAME >  <!—Content: template -->
</xsl:element>
<xsl:attribute name = NAME >  <!-- Content: template -->
</xsl:attribute>
<xsl:for-each   select = qexpression>  <!-- Content: template -->
</xsl:for-each>
<xsl:if   test = boolean-expression>  <!-- Content: template -->
</xsl:if>
```

<xsl:param   name = **NAME** > </xsl:param>
<xsl:text > <!-- Content: #PCDATA --> </xsl:text>
<xsl:value-of select = *qexpression* />

qexpression ::= dquote expression dquote
expression ::= expr || *expr / expression* || *quote* **char-string** *quote*
expr ::= xt:built-in(expr) || *$Name* || *@Name* || *dot*
built-in ::= **node-set**
dquote ::= **"**
quote ::= **'**
dot ::= **.**
Name ::= **NAME**
qboolean-expression ::= dquote boolean-expression dquote
boolean-expression ::= expression Boolean-operator expression
boolean-operator ::= **=** || **!=**

For the undefined conceptions we refer the reader to the specifications of XSLT (see [34]). Our main aim is demonstration of Scp4, but not a literal implementation of XSLT.

The built-in function **xt:node-set** was implemented to provide algorithmic universality to the fragment of XSLT. It is impossible to define repeated transformations of the XML-data by the XSLT-program, if this built-in function is not used. That is the main reason why the language XSLT is becoming an admirable practical object for supercompilation. Namely, that makes easy to specialize the interpreter w.r.t. a given program, when the given program does not contain a call of **xt:node-set** (see [1]). Emphasize, this property also is very meaningful for partial evaluators [5],[6],[7],[21].

3.4 An Interpreter of the Turing Machine in the Language XSLT

While the interpreter VT.ref ([11]) defines the semantics of the considered fragment of XSLT in terms of the language Refal, the second interpreter considered in this section is an interpreter of the Turing Machine written in the fragment of XSLT.

The existence of such interpreter of the Turing Machine proves the algorithmic universality of the chosen subset of XSLT. The sources of the interpreter are available for immediate download (see [11]).

The following documents were created for our comparative experiments:

- TM.dtd - the description of the domains of the programs TM.xsl , TMPQ.xsl and TMDoublePQ.xsl,

- TM.xml – the input data for the programs TM.xsl , TMPQ.xsl and TMDoublePQ.xsl,

- TM.xsl - the interpreter of the Turing Machine, which does not take care of the tape's borders. The experiments (supercompilation) were done for the concrete TM (the program to be evaluated with this interpreter) from the example DoublePQ [Table 1],

- TMPQ.xsl - a variant of the previous program, which evaluates a given TM for replacement all the characters **P** with the **Q**s.

- TMDoublePQ.xsl - a variant of the program TM.xsl evaluating a given TM for doubling of the characters **P** and replacing them with the **Q**s.

- TMN.dtd - the description of the domains of the programs TMNPQ.xsl and TMNDoublePQ.xsl,

- TMN.xml – the input data to the programs TMNPQ.xsl and TMNDoublePQ.xsl,

- TMNPQ.xsl, TMNDoublePQ.xsl - variants of TMPQ.xsl, TMDoublePQ.xsl , where the tape's borders are being worked on: the empty cells are added on the left hand and the right hand of the tape, if that is necessary.

These documents are available for immediate download (see [11]) and some of them can be found in the attachment No. 1.

3.5 The Experiments with Supercompilation of Double Interpretation

3.5.1 On adequate mapping of results of supercompilation

Before considering of the results of the experiments, it is necessary to say that the real internal language defining programs to be transformed by the supercompiler SCP4 is not the language Refal-5. But it is a Refal-graph language (see [24],[23],[18]), in which the steps of the Refal Machine are decomposed into more elementary actions and the pattern-matching takes into account not the only pattern from the left-side of the Refal-sentence, but, in a sense, all the left sides from the Refal-definition "simultaneously". I.e. in the Refal-graph language we can describe the process of pattern-matching as a non-trivial tree, where the failures during the pattern-matching backtrack us to the closest branching. In this sense, the branchings are "functional", i.e. they behave like the Refal-5 functions, the failure inside of which leads to an abnormal stop. Here one can refer to a rough analogy with the blocks in Refal-5 (see [26]).

SCP4 translates a program into the Refal-graph language, and only after that it starts the transforming. When the residual Refal-graph program is ready, then SCP4 translates it in a Refal-program. From point of view of a user, Refal-5 is the input (as well as the output) language for this supercompiler.

A program in the Refal-graph language often contains simple syntactic information, which allows optimization of the process of interpreting, if this process is running directly. Refal-5 does not contain some of such syntactic entities. That can be lead to a non-adequate translation (in Refal-5) of the results of the transformations with SCP4 [20]. Therefore, from a practical point of view, the direct interpretation of the Refal-graph language makes meaningful interest (we would like to point to the work by A.P. Konyshev [12])

3.5.2 *The testing and the time analysis*

We collected the running times and running steps of the examples described in the previous section in two tables. In each of the examples the number of the non-empty cells on the TM's tape is equal to 16. Such choice was motivated by the proportion of the times, which takes place: that is hours and seconds. Else either one of the times is huge or the other is very little. The first columns stay for the names of the examples. The second columns show information about the running of the original interpreter VT.ref (of the subset of XSLT), while the third columns are responsible for residual programs after the repeated supercompilation. The names of the columns are clear. The number *TimeSpeedup* (it is a quotient of the contents from the second column by the third one) is not indicated, if dividing by zero happens.

Table 8. Step speedup

	Original prg.	Residual prg.	StepSpeedup
TM	1960210	3195	613
TMPQ	48789	40	1219
TMDoublePQ	1960475	519	3777
TMNPQ	49512	58	853
TMNDoublePQ	1988750	1018	1953

Table 9. Time speedup (seconds)

	Original prg.	Residual prg.	TimeSpeedup
TM	14.741	0.070	210
TMPQ	0.350	0.000	--------------
TMDoublePQ	14.761	0.000	--------------
TMNPQ	0.300	0.000	--------------
TMNDoublePQ	13.830	0.020	691

Let us make some explanations to the tables.

Consider the third column in [Table 8]. The residual replacement of 16 characters **P** with 16 characters **Q** takes 40 Refal-steps, when the tape borders are not taken into account, and that takes 58 ones, when the borders are treated. Here the number of Refal-steps is grater than the expected 16, because a non-adequate translation of the program from the Refal-graph language in Refal-5 (see the section 2.5.1) and the representation of the character's sequence as a tree structure. The residual doubling of 16 characters **P** takes 519 Refal-steps, when the tape borders are not taken into account, and that takes 1018 steps, when the borders are treated (the number of the actions grows like **2*n^2**). During the interpretation this example takes 3195 Refal-steps; the reason is the looking for a needful TM's instruction each time. These numbers are quite clear: they demonstrate that seriously transformations happened during the supercompilation.

Consider the second column in [Table 8]. The numbers from it are grater: that is normal, because the interpreting of the interpreter. The growth of the number of the elementary actions (w.r.t. single interpretation) could be roughly viewed as "squaring".

The small numbers for the residual programs causes the very big *StepSpeedup*. That is expected because the properties of the XSLT-programs from our examples. The coefficient *TimeSpeedup* cannot be seen as a reflection of the real world: too big and too small times are not believable, they are just some properties of the hardware, the RAM's configuration determined with the external software environment and so on.

We have tried indirectly to determine the time speedup obtained by the two supercompilations (the one after the other) of the double interpretation. The problem to determinate this coefficient is caused by either the memory limit (swapping takes place) during the original interpretation (before supercompilation) or the zero-time of running of the residual program. The question is really interesting, because the very big step speedup still does not mean a similar time speedup. While all the TM's steps are uniform and their running time is approximately the same, then running time of the Refal-steps can be dispersed along a large scale.

The result of our thinking is the time speedup is equal to the step speedup (approximately).

Consider the third row *TMDoublePQ* from the table [Table 8], where the step speedup is equal to 3777. Let us increase the number of **P**s on the tape: namely, write two **P**s on the tape, after that write four **P**s and so on, each time doubling of this number. By the sense of the algorithm, the depending of the steps on the number of **P**s looks like the square, i.e. the doubling of the **P**s has to cause multiplication of the steps by four. Our experimental observation showed the same effect, while swapping (during the original interpretation) did not happen. Further we were launching only the residual programs while their running time became comparable with one second. Elementary arithmetical operations show that the number *StepSpeedup* non-remarkable differs from the number *TimeSpeedup*.

To be height-lighted: the time speedup *TimeSpeedup* turns one hour to one second.

As an example we consider the residual Refal-program, which is a result of supercompilation of TMDoublePQ.

Table 10. The result of supercompilation of TMDoublePQ

```
$ENTRY Go {
 ((Instruction e.2) e.0) ((State) e.4)
 ((LeftTape) e.6) ((Symbol) s.8)
 ((RightTape) e.7) e.1  = <F16 (e.6) (e.7) s.8>;
 }

F69 {
 (((Nod) ((Cell) Q) ((Nod) e.7) e.5) e.6) ((Nod) e.4) e.1
  = <F69 (((Nod) e.7)) ((Nod) ((Cell) Q) ((Nod) e.4))>;

 (((Nod) ((Cell) B) ((Nod) e.7) e.5) e.6) ((Nod) ((Cell) s.8)) e.1
  = <F16 (((Nod) ((Cell) Q) ((Nod) ((Cell) Q)
                                       ((Nod) e.7)))) ( ) s.8>;

 (((Nod) ((Cell) B) ((Nod) e.7) e.5) e.6)
 ((Nod) ((Cell) s.8) ((Nod) e.9) e.4) e.1
  = <F16 (((Nod) ((Cell) Q) ((Nod) ((Cell) Q) ((Nod) e.7))))
           (((Nod) e.9)) s.8>;
 }
```

```
F16 {
 (((Nod) e.5) e.6) (((Nod) ((Cell) s.9))) B
   = ((tm) ((LeftTape) ((Nod) ((Cell) B ) ((Nod) e.5)))
          ((Symbol) s.9) ((RightTape)));

 (((Nod) e.5) e.6) (((Nod) ((Cell) s.9) ((Nod) e.2) e.3) e.1) B
   = ((tm) ((LeftTape) ((Nod) ((Cell) B) ((Nod) e.5)))
          ((Symbol) s.9) ((RightTape) ((Nod) e.2)));

 (((Nod) e.5) e.6) (((Nod) ((Cell) s.9))) Q
   = <F16 (((Nod) ((Cell) Q) ((Nod) e.5))) ( ) s.9>;

 (((Nod) e.5) e.6) (((Nod) ((Cell) s.9) ((Nod) e.2) e.3) e.1) Q
   = <F16 (((Nod) ((Cell) Q) ((Nod) e.5))) (((Nod) e.2)) s.9>;

 (e.6) (e.1) P = <F69 (e.6) e.1>;
 }
```

Note that the XSLT-construction **<tag . . . > . . . </tag>** is encoded as **((tag . . .) . . .)** in the Refal-data by the parser.

That residual program is perfect: it contains no redundancies. The two functions **F69** and **F16** correspond to the internal states of the given program for the Turing Machine. The number of the Refal-steps of the residual program does not differ from the number of the steps of the input TM. The terminology defining the conceptions of the XSLT-interpreter has vanished at all. Moreover, the terminology, defining the conceptions of the TM-interpreter written in XSLT, has vanished at all. Running times of the all Refal-steps are uniformly bounded on input data. All recursions in the residual program are "tail"-recursions, i.e. (in the essence) there exist no recursive calls but only transformation of their arguments. One can translate this SCP4's result in a low-level language enough efficiently: all the function calls can be formed into jumps to labels: no stack's operations are needed (see, for example,[12]).

We would like once more to call the reader for paying attention to this residual program. This program clearly demonstrates our main result. We input (to the supercompiler SCP4) an interpreter of an algorithmically universal language, which has to interpret another interpreter of another algorithmically universal language, which in its turn has to interpret a simple program. After that we input the SCP4's output again to the supercompiler. The supercompiler SCP4 (by these two running) cleaned this double interpretation and constructed an algorithm in Refal substantially coinciding with the algorithm written in the TM-language.

The large names of the structures describing the TM's tape cause a little increasing of the length of this program w.r.t. the analogical program from the section 2.

4. SYNTACTIC ANALYSIS

The size of this paper does not allow make a precise syntactic analysis of the objective programs from our experiments: these programs are large. The sources of all these programs partially are given in the text of this paper and in the attachment No. 1, their complete versions are available by the internet (see [11]). Furthermore, from our point of view, it is more usable that one regards to a program transformer as an enough complicated physical system (rather than a mathematical abstraction). Such approach, on the one hand, allows predict behavior of the system on the level of the ideas, disregarding some of its properties ("a frictional force") and throwing away irrelevant technical details; on the other hand, it assumes that the system is living and is being developed, so understanding of its new properties can be possible just after specification of a concrete its model being considered ("a frictional force depends only on the properties of the surfaces contacting one with the other, but does not depend on the speed of the slipping"). Uncomputability of more or less serious problems of the automatic program transformation, approximation of these problems, apparently, make any attempt of complete formalization of the transformation non-perspective, but nevertheless such investigations are interesting.

The most complicate "physical" characteristics of our instrument for experimentation (SCP4) are the algorithms of: the generalization, the structuring of the function stack [25], the condition for folding of the driving tree and the order for passing along the driving tree [17], [27], [18]. It is technically difficult to follow behavior of the composition of the indicated characteristics with other tools of the transformations.

4.1.1 The semi-compositionality

Following after N. D. Jones [7], we have to relate our interpreter VT.ref to the semi-compositional ones. The classification of the parameters into syntactic and non-syntactic (see [7]) took place automatically during the supercompiling process: the indispensable condition for a simplifying ordering (see [9],[13]) to fold of the driving tree did not allow to lose the information about the static nature of the syntactic parameters, because during the interpretation the length of the syntactic structures of the being interpreted XSLT-program monotonously decreases within one logical step of the XSLT-Machine, and the XSLT-program is being restored in the beginning of each XSLT-step. It is necessary to emphasize that the indicated classification took place automatically (on-line) with one of the general tools of supercompilation; it happened logically -- the supercompiler SCP4 has not a special conception of a "syntactic parameter" (in the considered context).

Let us denote as **<FunctionName ... >** an abstract function call in a meta-language, which allows reasonings ([28]) about the whole set of the programming languages. Here, the word "function" is used enough broadly and a corresponding conception can be called (in a concrete language) some other word (for instance, in XSLT it is "template"), but the "function" always reflects an input point into a syntactic loop.

The Refal-syntax of the XSLT-interpreter (that is a property of our interpreter) fixes an injective mapping of the function stack's conceptions of the language XSLT into the function stack's conceptions of the language Refal (i.e. into its corresponding linguistic structures), preserving their stack's structure. We could be saying about a morphism if there exists a precise definition of the stack's structure, which does not depend on the concrete programming language. The below-following scheme illustrates this injective mapping:

Table 11. The "morphism" of the stacks.

<IntXSLT_refal ... >

```
                         <F_xslt ... <G_xsl ... > >
```

The XSLT-stack is moved to: ↓

```
<IntXSLT_refal …     <IntXSLT_refal                  >        >

                <F_xsl ...                <G_xsl ... >        >
```

The one-level denotation:

```
<IntXSLT^F_xslt_refal ... <IntXSLT^G_xslt_refal ... >>
```

In the scheme the nesting of the calls corresponds to their composition. The lower line is an object to be transformed with the upper line, the arguments of the upper calls either are disposed directly on the upper line or are enclosed with the projection of the corresponding upper angle brackets to the lower line.

Therefore, the information about the function stack's properties of a part of the being interpreted data becomes indirectly available for SCP4 and, as a consequence of that, this information is processed with the tools for analyzing namely the function stack's structures (see [25],[27]), but not with the general tools for treating arbitrary data of an unknown nature.

The input point for an XSLT-program is not indicated in its syntax, but is defined inside the body of an XML-document [34],[33]. That causes the partial function to be evaluated in run-time is not determined during the specialization of the XSLT-interpreter w.r.t. a given XSLT-program, while the XML-document is unknown. A logical consequence of this circumstance: in supercompile-time all possible variants of the partial functions, which can appear in run-time, have to be considered (by definition of the interpreter). And, hence, more non-informative (general) variants of the generalization [25],[24] of the data of the XSLT-interpreter are created. The essence of our repeated supercompilation (see the section [3.5.2]) is the cleaning of the result of the first supercompilation, when one declares that the input point should be a call of the first (defined in the XSLT program) template. It is interesting to note that after the first phase of the transformations the "garbage" inside the residual programs is trivial (see the residual programs in [11]), even though, naturally, they contain some XSLT-terms. Apparently, this is associated with the global logic of the considered XSLT-programs (and, certainly, with the logic of the XSLT-semantics), which contradicts a more or lesser substantial definition of the partial functions, which the author of the XSLT-programs did not develop as self-dependent semantic entities.

Repeated supercompilation does matter. There exists another evidently reason for that: the simplest elementary transformations being performed with the supercompiler SCP4, sure, do not commute, hence, the repeated supercompilation can produce more effective result than the first supercompilation.

Example: Let the following Refal program be an input to Scp4.

```
$ENTRY Go { e.number = <UnarySum (I) (e.number)>; }
UnarySum {
(e.numb1) (I e.numb2) = I <UnarySum (e.numb1) (e.numb2)>;
(e.numb1) () = e.numb1;
}
```

This definition of the unary addition (being viewed separately from the context of the function call) does not provide any information about the output structure of the function *UnarySum*. After the specialization w.r.t. the call context, a non-trivial output format **I e.out** will be derived. The specialization w.r.t. this output information takes place later than the global analyzing for the property of identity (see [19]), so the residual recursive component cannot be recognized as a syntactic identity function.

The result of the first supercompilation is:

```
$ENTRY Go { e.number = I <F7 e.number >; }
F7 {
I e.1 = I <F7 e.1 >;
     = ;
}
```

Now we again input the result program to Scp4 and obtain the following perfect residual program

```
$ENTRY Go { e.number = I  e.number; }
```

5. CONCLUSION

We have described a number of experiments with the supercompiler Scp4 (see [17],[18]). We have implemented an interpreter of an algorithmically universal fragment of the language XSLT. The interpreter is written in a functional programming language Refal. We have specialized the interpreter with respect to a Turing Machine interpreter written in XSLT, when the last interpreter has to evaluate the given programs for the TM. After that we input the SCP4's outputs again to the supercompiler. The supercompiler SCP4 cleaned this double interpretation and constructed algorithms in Refal substantially coinciding with the algorithms written in the TM-language. The examples we presented show considerable speedup after the two supercompilations.

6. ACKNOWLEDGMENTS

This work could not have been carried out without support of Prof. V.F. Turchin. The second author is grateful to the Wuhan University (China) and Prof. He Keqing for the partial support of this work. Our thanks to A.P. Konyshev, who supported Refal-5.

7. REFERENCES

[1] Clark J., XT Version 19991105 http://www.jclark.com/xml/xt.html , 1999.

[2] Futamura Y., Nogi K.,Generalized partial computation.,In: Partial Evaluation and Mixed Computation, Proceedings. Eds: D. Bjorner. A.P.Ershov and N.D. Jones, North-Holland, Amsterdam, pp.133-151, 1988.

[3] Futamura Y., Nogi, K. and Takano, A. Essence of generalized partial computation, Theoretical Computer Science 90(1991), pp.61-79, North-Holland, Amsterdam.

[4] Johnson M., XML for the Absolute Beginner. http://www.javaworld.com/javaworld/jw-o4-1999/jw-04-xml_p.htm , 1999.

[5] Jones N., Sestoft P., Sondergaard H., An experiment in partial evaluation: the generation of a compiler generator. In: Rewriting Techniques and Applications, Proceedings. Ed: Jouannaud J.-P., LNCS vol. 202, pp.153-166, Springer, 1988.

[6] Jones N.D., Gomard C.K., Sestoft P., Partial Evaluation and Automatic Program Generation, Prentice Hall International, June 1993.

[7] Jones N.D., What Not to Do When Writing an Interpreter for Specialization. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation,* volume 1110 of *Lecture Notes in Computer Science*, pages 216-237. Springer-Verlag, 1996.

[8] Jones N.D., Computability and Complexity from a Programming Perspective. Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997. ISBN number 0-262-10064-9.

[9] Higman G. Ordering by divisibility in abstract algebras, Proc. London Math. Soc. (3) 2(7) , 1952, pp.326-336.

[10] Karliukou (Korlyukov) A.V., User manual on the Supercompiler Scp4. (in Russian) http://www.refal.net/supercom.htm , 1999.

[11] Karliukou (Korlyukov) A.V., Nemytykh A.P. , Supercompilation of double interpretation: sources , demonstration. www.refal.net/~korlukov/demo_scp4xslt.zip

[12] Konyshev A.P, The translator from the Refal-graph language to the language C: sources and demonstration. http://www.botik.ru/pub/local/scp/refal5/ , 2000.

[13] Kruskal J.B., Well-quasi-ordering, the Tree theorem, and Vazsonyi's conjecture, Trans. Amer. Math. Society 95, 1960 pp.210-225.

[14] Levin L., Fundamentals of Computing., http://www.cs.bu.edu/fac/lnd/toc/ , 1996.

[15] Nemytykh A.P., Pinchuk V.A., Turchin V.F. A Self-Applicable Supercompiler. Partial Evaluation, Lecture Notes in Computer Science, 1110 (1996) pp: 232-237.

[16] Nemytykh A.P., Pinchuk V.A., Program Transformation with Metasystem Transitions: Experiments with a Supercompiler. In: Perspectives of Systems Informatics, Proceedings. Eds: D. Bjorner et al., LNCS vol. 1181, pp.249-260, Springer, 1996.

[17] Nemytykh A.P., Turchin V.F.,The Supercompiler Scp4: sources , on-line demonstration. http://www.botik.ru/pub/local/scp/refal5/ , 2000.

[18] Nemytykh A.P., Supercompiler Scp4: Use of Quasi-distributive Laws in Program Transformation. , In: Proceedings of International Software Engineering Symposium, Wuhan University Journal of Natural Sciences, Vol. 6, No. 1-2, pp:375-382, March 2001, Wuhan, China.

[19] Nemytykh A.P., The Supercompiler Scp4: Online transformations after the folding procedure. (submitted to Journal of Software, Chinese Academy of Sciences, Beijing, China), 2001.

[20] Romanenko, S.A. Refal-4 - an extension of Refal-2, in which the driving can be expressed. , Moscow , M.V. Keldysh Institute for Applied Mathematics, Russian Academy of Sciences, preprint #147, Moscow, 1987.

[21] Romanenko, S.A. A compiler generator produced by a self - applicable specializer can have a surprisingly natural and understandable structure. In: Partial Evaluation and Mixed Computation, Proceedings. Eds: D. Bjorner. A.P.Ershov and N.D. Jones, North-Holland, Amsterdam, pp.445-463, 1988.

[22] Salomaa A., Jewels of Formal Language Theory. Computer Science Press, 1981.

[23] Turchin V.F., The Language Refal, the Theory of Compilation and Metasystem Analysis, Courant Computer Science Report #20, New York University, 1980.

[24] Turchin V.F., The concept of a supercompiler, ACM Transaction on Programming languages and Systems, 8(3), pp:292-325, 1986.

[25] Turchin, V. F. The algorithm of generalization in the supercompiler. Proceedings of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation, Amsterdam: North-Holland Publishing Co., 1988.p. 531-549.

[26] Turchin V.F., Refal-5, Programming Guide & Reference Manual. Holyoke, Massachusetts: New England Publishing Co., 1989 (electronic version: http://www.botik.ru/pub/local/scp/refal5/ , 2000.)

[27] Turchin V.F., Metacomputation in the language Refal. (unpublished, private communication), 1990.

[28] Turchin V.F. , Nemytykh A.P., Metavariables: Their implementation and use in Program Transformation, Technical Report CSc. TR 95-012, City College of the City University of New York, 1995.

[29] Turchin V.F., Metacomputation: Metasystem Transition plus Supercompilation. In: Partial Evaluation, Proceedings. Eds: O. Danvy, R.Glueck and P.Thiemann, LNCS vol 1110, pp.481-509, Springer, 1996.

[30] Turchin V.F., Supercompilation: Techniques and Results. Perspectives of System Informatics, LNCS, vol. 1181 (1996) pp.227-248.

[31] Turchin V.F., Turchin D.F., Konyshev A.P., Nemytykh A.P., Refal-5: sources and executable modules. http://www.botik.ru/pub/local/scp/refal5/ , 2000.

[32] Turchin V.F. Refal: The Language for Processing XML Documents. http://www.refal.net/english/xmlref_1.htm , 2000.

[33] XML: http://www.w3.org/TR/1998/REC-xml-19980210 , 1998

[34] XSL Transformations (XSLT) Version 1.0 W3C. http://www.w3.org/TR/1999/REC-xslt-19991116 , 1999

ATTACHMENT 1

TMNDoublePQ.xsl

```
<!--   ==========================================================================              -->
<!--  TMNDoublePQ.xsl                                                                          -->
<!--  An interpreter of the Turing Machine.                                                    -->
<!--   ==========================================================================              -->
<!-- Example of the program for the Turing Machine.                                            -->
<!-- The program replaces an array of the characters P with another array, where each of the Ps was changed with two Qs   -->
<!-- (one per cell). If, for instance, in the beginning of the TM's job there are exactly ten of the characters P on the tape   -->
<!-- (followed one by another) and the pointer indicates to the first of them, then, when the machine will stop, the tape will   -->
<!-- contain twenty Qs followed one by another. The tape's borders are being worked on: the empty cells are added on   -->
<!-- the left hand and the right hand of the tape, if that is necessary.                       -->
<!--   ==========================================================================              -->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xt="http://www.jclark.com/xt" version="1.0">

<!--  TMNDoublePQ.xsl
*******************************************************************************************************          *
*   The interpreter of Turing Machine interprets a concrete Turing Machine.                                      *
*    Initially, the concrete machine is put inside a program (i.e. the concrete machine is just a part of this XSLT-program).   *
*                                                                                                                *
*    The instruction has the form: (CurrState   CurrSymb   NextSymb   NextState   Move).                         *
*    The instruction is executed if the machine is in the configuration (CurrState   CurrSymb). That is to say, its internal state   *
*    is the CurrState and the symbol scanned under the head is the CurrSymb. The execution consists in overwriting   *
*    the cell's contents under the head with the symbol NextSymb, in moving the head from the current cell to its adjacent   *
*    neighbor in the direction indicated by the Move and in  changing the internal current state with the NextState.   *
*    A program consists of a finite number of the instructions. There exists at least one instruction with the start state "start".   *
*    There can be instructions with the finish state "stop". The program starts with an instruction with the start state.   *
*    The execution of the program consists in stepwise executing of the instructions.                            *
*    The input to the TM is a finite array of rightward cells.                                                   *
*******************************************************************************************************          *
-->

<!-- The concrete Turing Machine: replacement of Ps with double Qs on the tape. -->

<!--  **************************************************************************    -->
<!--   The interpreter itself.                                                      -->
<!--  **************************************************************************    -->

<xsl:template match="Go">
    <xsl:call-template name="Go2">
    <xsl:with-param name="TMS">
    <TMState>
    <Instruction CurrState="start" CurrSymb="B" NextSymb="B" NextState="stop" Movement="right">
    <Instruction CurrState="start" CurrSymb="Q" NextSymb="Q" NextState="start" Movement="right">
    <Instruction CurrState="start" CurrSymb="P" NextSymb="Q" NextState="moveleft" Movement="left" >
    <Instruction CurrState="moveleft" CurrSymb="Q" NextSymb="Q" NextState="moveleft" Movement="left" >
    <Instruction CurrState="moveleft" CurrSymb="B" NextSymb="Q" NextState="start" Movement="right">
    </Instruction>
    </Instruction>
    </Instruction>
    </Instruction>
    </Instruction>
    <!-- The end of the concrete Turing Machine. -->
```

```
                <State>start</State>
                <LeftTape>
                <Nod><Cell>B</Cell><Nod><Cell>B</Cell>
                <Nod><N/></Nod>
                </Nod></Nod>
                </LeftTape>
                <xsl:copy-of select="Symbol"/>
                <xsl:copy-of select="RightTape"/>
        </TMState>
        </xsl:with-param>
        </xsl:call-template>

</xsl:template>


<!--   ===============================================================              -->
<!--   The head template.                                                           -->
<!--   ===============================================================              -->


<xsl:template name="Go2">
<xsl:param name="TMS"/>
<xsl:for-each select="xt:node-set($TMS)/TMState">

<xsl:choose>

<xsl:when test="State='stop'">
     <tm>
              <xsl:copy-of select="LeftTape"/>
              <xsl:copy-of select="Symbol"/>
              <xsl:copy-of select="RightTape"/>
     </tm>
</xsl:when>

<xsl:otherwise>
              <xsl:call-template name="Go2">
              <xsl:with-param name="TMS">
              <TMState>
              <xsl:copy-of select="Instruction"/>
                          <xsl:call-template name="Step">
                          <xsl:with-param name="TMS">
                          <TMState>
                                      <xsl:copy-of select="Instruction"/>
                                      <xsl:copy-of select="State"/>
                                      <xsl:copy-of select="LeftTape"/>
                                      <xsl:copy-of select="Symbol"/>
                                      <xsl:copy-of select="RightTape"/>
                          </TMState>
                          </xsl:with-param>
                          </xsl:call-template>
              </TMState>
              </xsl:with-param>
              </xsl:call-template>
</xsl:otherwise>

</xsl:choose>
</xsl:for-each>
</xsl:template>


<!--   ===============================================================              -->
<!--   One step of the Turing Machine.                                              -->
```

```
<!--   ===============================================================                  -->

<xsl:template name="Step">
     <xsl:param name="TMS"/>
<xsl:for-each select="xt:node-set($TMS)/TMState">

<xsl:choose>

<xsl:when test="Instruction/@CurrState=State">
            <xsl:choose>
            <xsl:when test="Instruction/@CurrSymb=Symbol">
                        <xsl:choose>
                        <xsl:when test="Instruction/@Movement='right'">
                                    <State>
                                                <xsl:value-of select="Instruction/@NextState"/>
                                    </State>
                                    <LeftTape>
                                                <Nod>
                                                <Cell>
                                                <xsl:value-of select="Instruction/@NextSymb"/>
                                                </Cell>
                                                <xsl:copy-of select="LeftTape/Nod"/>
                                                </Nod>
                                    </LeftTape>
                                    <Symbol>
                                                <xsl:value-of select="RightTape/Nod/Cell"/>
                                    </Symbol>
                                    <RightTape>
                                                <Nod>
                                                <xsl:for-each select="RightTape/Nod/Nod">
                                                <xsl:apply-templates/>
                                                </xsl:for-each>
                                                </Nod>
                                    </RightTape>
                        </xsl:when>

                        <xsl:when test="Instruction/@Movement='left'">
                                    <State>
                                                <xsl:value-of select="Instruction/@NextState"/>
                                    </State>
                                    <LeftTape>
                                                <Nod>
                                                <xsl:for-each select="LeftTape/Nod/Nod">
                                                <xsl:apply-templates/>
                                                </xsl:for-each>
                                                </Nod>
                                    </LeftTape>
                                    <Symbol>
                                                <xsl:value-of select="LeftTape/Nod/Cell"/>
                                    </Symbol>
                                    <RightTape>
                                                <Nod>
                                                <Cell>
                                                <xsl:value-of select="Instruction/@NextSymb"/>
                                                </Cell>
                                                <xsl:copy-of select="RightTape/Nod"/>
                                                </Nod>
                                    </RightTape>
                        </xsl:when>
                        </xsl:choose>
            </xsl:when>

            <xsl:otherwise>
                        <xsl:call-template name="Step">
                        <xsl:with-param name="TMS">
                        <TMState>
                                    <xsl:copy-of select="Instruction/Instruction"/>
                                    <xsl:copy-of select="State"/>
                                    <xsl:copy-of select="LeftTape"/>
                                    <xsl:copy-of select="Symbol"/>
                                    <xsl:copy-of select="RightTape"/>
                        </TMState>
                        </xsl:with-param>
                        </xsl:call-template>
            </xsl:otherwise>
            </xsl:choose>
</xsl:when>

<xsl:otherwise>
            <xsl:call-template name="Step">
            <xsl:with-param name="TMS">
            <TMState>
                        <xsl:copy-of select="Instruction/Instruction"/>
                        <xsl:copy-of select="State"/>
                        <xsl:copy-of select="LeftTape"/>
                        <xsl:copy-of select="Symbol"/>
                        <xsl:copy-of select="RightTape"/>
            </TMState>
            </xsl:with-param>
            </xsl:call-template>
</xsl:otherwise>

</xsl:choose>
</xsl:for-each>
</xsl:template>


<!--   ==============================================================              -->
<!--   Processing of the end of the tape.                                          -->
<!--   ==============================================================              -->


<xsl:template match="N">
      <Cell>B</Cell>
      <Nod><Cell>B</Cell>
      <Nod><N/>
      </Nod></Nod>
</xsl:template>

<xsl:template match="Nod">
      <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="Cell">
      <xsl:copy-of select="."/>
</xsl:template>


<!--   ****************************************************************************        -->
<!--   The end.                                                                            -->
<!--   ****************************************************************************        -->
```

```
</xsl:stylesheet>
```

TMN.dtd

```
<!--  ==============================================================================  -->
<!--  TMN.dtd                                                                          -->
<!--  The description of the domains of the programs:   TMNPQ.xsl and TMNDoublePQ.xsl.  -->
<!--  ==============================================================================  -->

<!ELEMENT Go (Symbol, RightTape)>
<!ELEMENT TMState (Instruction, State, LeftTape, Symbol, RightTape) >
<!ELEMENT LeftTape  (Nod) >
<!ELEMENT RightTape (Nod) >
<!ELEMENT Nod ((Cell, Nod) | N) >
<!ELEMENT Symbol    (#PCDATA) >
<!ELEMENT Cell      (#PCDATA) >
<!ELEMENT State     (#PCDATA) >
<!ELEMENT N         (#PCDATA) >
<!ELEMENT Instruction (Instruction | EMPTY) >
<!ATTLIST Instruction
                CurrState     CDATA     #IMPLIED
                CurrSymb      CDATA     #IMPLIED
                NextSymb      CDATA     #IMPLIED
                NextState     CDATA     #IMPLIED
                Movement      CDATA     #IMPLIED >
```

TMN.xml

```
<!--  ==============================================================================  -->
<!--  TMN.xml                                                                          -->
<!--  The input tape to the programs:   TMNPQ.xsl and TMNDoublePQ.xsl.                 -->
<!--  ==============================================================================  -->

<!DOCTYPE Go SYSTEM "TMN.dtd">

<Go>

<Symbol>P</Symbol>

<RightTape>
<Nod><Cell>P</Cell><Nod><Cell>P</Cell><Nod><Cell>P</Cell>
<Nod><Cell>P</Cell><Nod><Cell>P</Cell><Nod><Cell>P</Cell>
<Nod><Cell>P</Cell>
<Nod><N/></Nod>
</Nod></Nod></Nod></Nod></Nod></Nod></Nod>
</RightTape>

</Go>
```

The other documents can be found in www.refal.net/~korlukov/demo_scp4xslt.zip .