

## 1. Abstract

A tool is presented for research work on Turing machines and other abstract programming models. This tool is based on the use of the Recursive Function Algorithmic Language (Refal). A short background of this language is presented. Examples are given as to how this language may be useful for running simulations of Turing machines, and how it may be useful for the researcher into abstract machines.

## 2. Background

Refal (the Recursive Function Algorithmic Language) was developed in the late 1960's in the Soviet Union, by V. F. Turchin, et al (see [1] for a bibliography of early works). Its use since then has been primarily in the Soviet Union, with a number of implementations existing. Additionally, an implementation exists in the United States for IBM System 370 type computers, and a new implementation has been recently completed on a DEC VAX-11/780 at The City College of New York (the syntax used in this paper conforms to the VAX implementation -- the IBM version differs slightly).

Refal was developed as a language for use in AI research. It is modelled by the "Refal Machine", an abstract computing machine with computational power equivalent to a Turing Machine. Its strength lies in its excellent pattern matching capabilities, the ease with which rule-based systems may be designed in it, and in some sense its minimality as a model for computation. In addition, Refal has the ability to act as its own metalanguage, a feature which is useful in the specification of compilers and automatic programming systems. Two major applications will be discussed in this section.

### 2.1 Refal as an Algorithmic Description Language.

Refal can be used as a language for descriptions of algorithms. Although the allowable operations in Refal are few, it turns out that algorithms may be specified concisely and with great clarity. A brief description of some features of Refal appears in section 2.3. However, for purposes of illustration, a simple algorithm for determining whether a given string is a palindrome or not is given below:

```
(1) PAL(  
    s.edge e.middle s.edge = <PAL e.middle>;  
    s.left e.middle s.right = 'not a palindrome';  
    e.else = 'is a palindrome';  
);
```

In this example, a function called PAL is defined. When PAL receives an argument in which the first and last symbols are identical (matched by the two occurrences of the variable s.edge in the first sentence), then a recursive call to PAL is issued, with the first and last symbols removed from the original argument. If this is not the case, (i.e., the first and last symbols are not the same), then the second sentence is tested against the argument. If there are at least two symbols in the argument, then clearly the argument is not a palindrome, and the appropriate result is indicated. Finally, in the case where either one symbol or no symbol is left, the third sentence applies, and it is determined that the original argument was in fact a palindrome.

Note that each sentence has a "screening" effect on following sentences: each sentence has a pattern expression on the left side which defines a class of object expressions which it may match. Since matching of patterns against objects occurs in the order in which the sentences are written, set-differencing is obtained by



proper ordering of sentences within a function definition.

Of course, complex programming problems may require complex solutions. However, Refal's functional approach lends flexibility in the specification of algorithms, and encourages modularity and isolation of purpose to small sets of functions.

## 2.2 Refal in Automatic Programming - A Supercompiler

Although Refal could conceivably be used for string manipulation applications, perhaps its most striking use is in the design of Automatic Programming systems. Such a system has been built, and is undergoing further development. It is called the Refal Supercompiler system. Work is currently proceeding at the Department of Computer Sciences at The City College of New York, with funding from the National Science Foundation. Recent works on the Supercompiler may be found in [1,2,3].

In one mode, the Supercompiler acts as a producer of compilers. It takes as its input the description of an interpreting function for some language  $L_i$ , and produces a compiler for that language in  $L_m$ , the language of the machine on which  $L_i$  is to be compiled.

In addition to this "compiler-compiler" mode, the Refal Supercompiler can act as a general program optimizer. Coupled with an appropriate input transformer ( $L_i \rightarrow R$ ) and an appropriate output mapping ( $R \rightarrow L_m$ ), where  $R$  is Refal, the Supercompiler can perform various optimizations and transformations on programs in various languages, targeted for arbitrary machines. It is advantageous to choose Refal as the language in which the supercompiler and its auxiliary programs are written, as this minimizes the range of program structures on which optimizing functions must work.

## 2.3 Refal Concepts

In order to understand the use of Refal in simulating Turing Machines, it is necessary first to have a basic understanding of how Refal works. This section will serve to describe the various concepts needed to understand Refal and programs written in it.

### 2.3.1 Expressions

Expressions are the basic building blocks of Refal data and programs. Expressions consist of strings of symbols and certain metasympols. These metasympols include structure brackets (represented as properly paired parentheses), activation brackets ( $\langle$  and  $\rangle$ ), and variables. Symbols in Refal are any of the characters which may be typed at a terminal (surrounded by apostrophes), and compound symbols, new symbols written as strings of characters delimited by either blanks or periods. Compound symbols are generally used to indicate function names, integers, etc. Examples of Refal expressions:

(2) 'ABC'        '+34('XX')('YZ')  
      ('THIS') ('IS') (('AN EXPRESSION') 'I')

PAL, seen previously, is an example of a compound symbol, as is 34, above.

### 2.3.2 Variables and Patterns

Variables in Refal have a different meaning than those in most other programming languages. Since variables only get values in the process of pattern recognition, and retain no value after the process of pattern replacement, they are called free variables.

There are three kinds of variables in Refal. E-variables (expression variables) are used to match arbitrary expressions. That is, in the process of pattern recognition, an e-variable may take as its value any expression, including the empty expression. S-variables match any single symbol, including ordinary characters and compound symbols. An s-variable may not take an empty value. Finally, t-variables may match any term, that is, one symbol, or any expression surrounded by structure brackets. Again, a t-variable may not take an empty value.

Variables are written as a variable type indicator followed by an index, (which may be either a letter, a digit, or a compound symbol preceded by a period), e.g., e1, s.parse, tX. The indices have significance only within the bounds of one sentence -- multiple variables in the left side of a sentence with the same index must match against the same object.

Pattern expressions are expressions which contain no activation brackets, and are used to define a set of objects. Pattern expressions generally appear on the left sides of sentences in a Refal program. Patterns may include arbitrary instances of symbols, variables, and structure brackets. For example, a pattern which is to match any expression which contains at least one symbol on the left, and an A at the right end could be written as:

(3) s1 eX 'A'

A pattern which is to match any expression which starts and ends with the same symbol might be written as:

(4) s.same e0 s.same

Note that the symbol variables in (4) both have the same index, which insures that only objects which begin and end with the same symbol will be matched by this pattern.

Since any Refal function takes as input exactly one expression, multiple arguments are obtained by using structure brackets in the data and pattern expressions. For example:

(5) ('A' e.first) (e.second 'B') eQ

is a pattern expression which checks to see that the first pair of structure brackets in the argument has an A as the first element of the subexpression delimited by them, that the second pair of structure brackets has a B at the right end of the subexpression they surround, and that anything else may follow, including nothing.

### 2.3.3 Sentences

Sentences in Refal consist of a left side and a right side, separated by an equal sign =. The left side of a sentence consists of a pattern expression. The right side of a sentence may consist of any Refal expression, with the restriction that any variable appearing in the right side must also appear in the left side.

A sentence then is used to describe the action to be taken if the pattern in the left side matches against the object expression which the function receives as its argument. If a match occurs, then the active expression (i.e., the one in which the

function, and consequently, the sentence is called) is replaced by the right side, with the values that the variables received in the process of pattern recognition replacing each occurrence of those variables in the right side. For example, suppose that function PAL is called with the argument RADAR, thus: <PAL RADAR>. When this expression is activated, s.edge in the first sentence will get the value R, and e.middle will get the value ADA. Then, the original active expression will be replaced by: <PAL ADA>, that is, the right side of the sentence which matched, with e.middle replaced by the value it received in the pattern matching process.

### 2.3.4 Functions

Refal function definitions are comprised of a function name, followed by a number of sentences, which are separated by semicolons, and surrounded by braces. As was seen previously, the ordering of the sentences has an impact on the action taken by a function. A number of functions taken together comprise a program.

Refal programs are purely functional. As seen in (5), even the process of splitting an argument into separate subarguments must be specified. In addition, in the strictest version of Refal, there are no global variables as such.

Some of the more common functions expected in a programming language, such as I/O, arithmetic, etc., are added to the language through external functions: functions not written in Refal, but rather in the language of the underlying machine. For example, in order to cause an expression to be written to an external device, such as a terminal, one might use function PRINT:

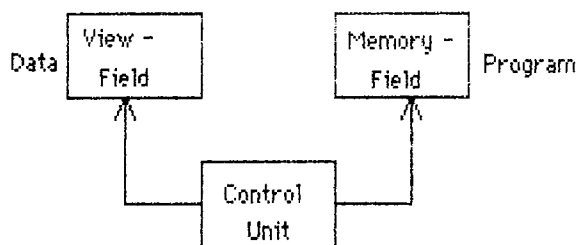
(6) <PRINT 'SOME VALUE' >

This function call, when activated, will cause SOME VALUE to be printed, and the argument, SOME VALUE, will be left in place of the call.

### 2.3.5 The Refal Machine

Finally, we arrive at the concept of a Refal Machine. This is a model of an abstract machine which executes Refal programs on Refal data. The Refal Machine consists of three parts:

(7)



The memory field contains a fixed Refal program. The view field contains an object (data) on which the Refal machine is to operate.

To start the Refal Machine, an active expression (i.e., one containing activation brackets) is inserted into the view field, and the control unit is started. The control unit finds the leading (leftmost, innermost) pair of activation brackets in the object (view field). Having found them, and the name of the function indicated following the left activation bracket, it locates the function definition in the memory field. The pattern in the first sentence is applied to the object expression, and if a match is found (according to a strict matching process described in [1]), the active expression

is replaced by the right side of the sentence, after replacing any variables with their values. If no match is found, the control unit applies the next sentence and so on. As soon as a match is found, the replacement is made, and the control unit locates the next leading activation bracket pair. If no match is found, the control unit stops, indicating an abnormal termination.

The efficiency of programs run in this fashion is on the same order as that of programs written and run in LISP. Greater efficiency, of course, may be achieved by using the supercompiler as an optimizer: in this mode, efficiencies approaching those of well-written assembler language programs have been achieved [3].

To illustrate the step-by-step execution of a Refal program, let us return to the definition of function PAL in (1). Suppose that the starting configuration of the view field is: <PAL RADAR>, as above, and the view field contains the definition of PAL as in (1). Then the contents of the view field would be modified, step-by-step, as follows:

(8)	<PAL RADAR >	
	<PAL ADA >	by the first sentence
	<PAL D >	by the first sentence
	is a palindrome	by the third sentence

At this point, the Refal Machine comes to a halt, as there are no activation brackets remaining in the view field: the work is completed. By using the interactive debugging system, it is possible to have the Refal system print out the result of each step.

### 3. Turing Machine Representation in Refal

A Turing machine  $M$  may be represented in the following manner (see [4]):

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

where

$Q$  is the finite set of states

$\Gamma$  is the finite set of allowable tape symbols

$B$ , an element of  $\Gamma$ , is the blank

$\Sigma$ , a proper subset of  $\Gamma$  not including  $B$ , is the set of input symbols

$\delta$  is the next move function, which maps  $Q \times \Gamma$  to  $Q \times \Gamma \times (L, R)$ , and may

be undefined for some values of  $Q \times \Gamma$

$q_0$ , an element of  $Q$ , is the start state

$F$ , a subset of  $Q$ , is the set of final states.

As an example of this representation, suppose we wish to design a Turing Machine which will accept a string of the form  $0^n 1^n$ . One way of doing this is as follows:

(10)  $Q = \{q_0, q_1, q_2, q_3, q_4\}$ ,  $\Sigma = \{0, 1\}$ ,  $\Gamma = \{0, 1, X, Y, B\}$ , and  $F = \{q_4\}$ .

The machine will be programmed in such a way that as it scans left-to-right through the input string, each 0 will be changed to an X. Upon changing a 0 to an X, the machine will scan to the right until it finds the first 1, changing it to a Y, and will return to the left looking for the next 0. When no more 0's are found, the machine scans to the right to check that no more 1's are to be found. If none are, the machine accepts the string and stops.

The definition of  $\delta$ , the transition function, is usually given as a table:

(11)

<u>State</u>	<u>Symbol</u>				
	0	1	X	Y	B
q <sub>0</sub>	(q <sub>1</sub> , X, R)	--	--	(q <sub>3</sub> , Y, R)	--
q <sub>1</sub>	(q <sub>1</sub> , 0, R)	(q <sub>2</sub> , Y, L)	--	(q <sub>1</sub> , Y, R)	--
q <sub>2</sub>	(q <sub>2</sub> , 0, L)	--	(q <sub>0</sub> , X, R)	(q <sub>2</sub> , Y, L)	--
q <sub>3</sub>	--	--	--	(q <sub>3</sub> , Y, R)	(q <sub>4</sub> , B, R)
q <sub>4</sub>	--	--	--	--	--

In table (11) above, function  $\delta$  is described as a number of triples, each consisting of the next state, the symbol to be written, and the direction in which the tape head moves.

There are a number of ways of representing this Turing Machine (TM) in Refal. As a first attempt, let us describe the format of the TM as follows:

(12)  $\langle \text{TM1 S.STATE (E.LEFT) S.HEAD E.RIGHT} \rangle$ , where:

E.LEFT represents the string of tape symbols to the left of the read/write head,

S.HEAD represents the symbol at which the tape head has stopped,

E.RIGHT represents the string of symbols to the right of S.HEAD,

S.STATE is the current state,

and the right structure bracket (parenthesis) is positioned just to the left of the symbol being read. Upper case letters are used here to indicate metasympols which will represent elements of the modelled TM configuration.

In this example, one sentence is written for each element of  $Q \times \Gamma$  which has a value, and the collection of all the sentences into one Refal function represents the entire TM. Then we can define TM1 as:

(13.0) TM1 {

(13.1)  $q_0 (e.left) 0 e.right = \langle \text{TM1 } q_1 (e.left X) e.right \rangle$ ;

(13.2)  $q_0 (e.left) Y e.right = \langle \text{TM1 } q_3 (e.left Y) e.right \rangle$ ;

(13.3)  $q_1 (e.left) 0 e.right = \langle \text{TM1 } q_1 (e.left 0) e.right \rangle$ ;

(13.4)  $q_1 (e.left sa) 1 e.right = \langle \text{TM1 } q_1 (e.left 0) e.right \rangle$ ;

(13.5)  $q_1 (e.left) Y e.right = \langle \text{TM1 } q_1 (e.left Y) e.right \rangle$ ;

```

(13.6)    q2 (e.left sa) 0 e.right  = <TM1 q2 (e.left) sa 0 e.right >;
(13.7)    q2 (e.left) X e.right > = <TM1 q0 (e.left X) e.right >;
(13.8)    q2 (e.left sa) Y e.right > = <TM1 q2 (e.left) sa Y e.right >;
(13.9)    q3 (e.left) Y e.right > = <TM1 q3 (e.left Y) e.right >;
(13.10)   q3 (e.left) B e.right > = <TM1 q4 (e.left B) e.right >;
(13.11)   q4 (e.left B) e.right > = 'ACCEPT'
(13.12)   );

```

Notice that for each state in which the TM must move its head left (e.g., 13.4, 13.6, and 13.8), an extra symbol variable, sa, is inserted, to represent that symbol which is just to the left of the tape head. This is necessary because TM1 is written in such a way that in general only the tape head symbol is shown explicitly, and e-variables are used to represent the left and right ends of the tape. Since on moving left, a symbol must be moved out of the structure brackets, an extra s-variable must be used in these cases. It is, of course, possible to always specify both symbols adjoining the tape head symbol explicitly, but this simply complicates the writing of the definition of TM1.

Suppose that we wish to determine whether 0011 is in the set of strings defined by  $0^n 1^n$ . In the version described in (13), the TM is started by loading the definition of TM1 into the memory field of the Refal machine, and inserting

```
(14) <TM1 q0 () 0 0 1 1 B>
```

into the view field. Note that here the B is the TM blank symbol. Of course it is possible to rewrite TM1 so that the empty string delimits the right end of the semi-infinite TM tape, and in Refal programming this is a general method. However, for purposes of illustration, let us assume that the B represents an infinite number of blanks to the right of the TM tape.

Upon startup, the Refal machine will simulate this TM in the following steps, each step representing one configuration of the TM:

```

(15) <TM1 q0 () 0 0 1 1 B>
      <TM1 q1 (X) 0 1 1 B>
      <TM1 q1 (X 0) 1 1 B>
      <TM1 q2 (X) 0 Y 1 B>
      <TM1 q2 () X 0 Y 1 B>
      <TM1 q0 (X) 0 Y 1 B>
      <TM1 q1 (X X) Y 1 B>
      <TM1 q1 (X X Y) 1 B>
      <TM1 q2 (X X) Y Y B>
      <TM1 q2 (X) X Y Y B>
      <TM1 q0 (X X) Y Y B>
      <TM1 q3 (X X Y) Y B>
      <TM1 q3 (X X Y Y) B>
      <TM1 q4 (X X Y Y B) >
      'ACCEPT'

```

Notice that by reading the step-by-step execution of the Refal machine, it is

possible to determine what each move, and the corresponding configuration, of the TM was. For example, in making the third step, the TM goes from state  $q_1$  to  $q_2$ , moves the tape head one symbol to the left, and writes a Y where a 1 was previously. Again, the right structure bracket is interpreted to be just to the left of the current symbol.

Of course, if it is only necessary to check set membership of  $0011$  in  $0^n1^n$ , for non-negative  $n$ , without a trace of each step, then a much more straightforward Refal function can be written. For example:

```
(16) TM2(
      0 e1 1 e2 - <TM2 e1 e2>;
      = 'ACCEPT' ;
      ex - 'REJECT' ;
      );
```

TM2 works for the following reason. Note that two e-variables,  $e_1$  and  $e_2$  exist on the same level in the first sentence. In this case, since either one may take an arbitrary expression as its value, the leftmost e-variable,  $e_1$ , is initially given an empty value. Then, the pattern recognition process proceeds left-to-right thus: the argument is checked to see if it begins with a 0. If it does,  $e_1$  is given an empty value, and the next symbol in the argument is checked to see if it is a 1. If it is, then  $e_1$  retains its empty value, and  $e_2$  gets the remainder of the argument as its value. If the next symbol is not a 1, then  $e_1$  is *lengthened* by one symbol, and the check for the next symbol being a 1 is repeated. This lengthening process continues until either a 1 is found, or the end of the argument is reached. In the latter case, the first sentence fails. As a result, the first sentence in the definition of TM2 matches its argument only if the first symbol in the argument is a 0, and there is a 1 to its right. When this happens, one 0 and one 1 are deleted, and TM2 is called again recursively.

The second sentence comes into play only when the argument string is empty, and so it may be implied that the original string was in fact a member of the desired set. Finally, the third sentence was added to take care of the case where either the original string contained something other than a 0 or 1, or the string was not in  $0^n1^n$ . Such a final sentence could have been added to the definition of TM2, in order to exclude strings which fail for these reasons.

Returning to the earlier version, it is apparent that by tracing the execution of the Refal machine, it is possible to "see" each step that the modelled TM takes. This feature itself should prove Refal to be invaluable as an aid to the investigator into the behaviour of finite state machines.

For a third method, suppose we decide to design the TM in the style of a finite state machine, so that one Refal function represents one state of the machine, and one sentence is written in each function to represent each transition out of that state. Let us assume that the format of the Refal representation of this machine will be:

(17)  $\langle F.STATENAME (E.LEFT) S.HEAD E.RIGHT \rangle$ , where:

$F.STATENAME$  is the name of the state-function, and

$E.LEFT$ ,  $S.HEAD$ , and  $E.RIGHT$  are as in (12) above.

Then, the writing of the new representation of the TM progresses in a straightforward manner: for each state in  $\delta$  (table 11, above), we create a Refal function, with one sentence for each allowable transition. For each state in  $F$ , the set of final states, we create a Refal sentence, similar to sentence (13.11) in TM1. In this example, let us assume that the TM tape is bounded on the right, allowing us to delete



the blank symbol B as used in TM1. Then, accepting states will have the empty expression on the right, to indicate the remaining infinite tape.

This new representation then will consist of the following Refal program:

(18)

```
STATE0( (e.left) 0 e.right = <STATE1 (e.left X) e.right>;
        (e.left) Y e.right = <STATE3 (e.left Y) e.right>; );
STATE1( (e.left) 0 e.right = <STATE1 (e.left 0) e.right>;
        (e.left sa) 1 e.right = <STATE2 (e.left) sa Y e.right>;
        (e.left) Y e.right = <STATE1 (e.left Y) e.right>; );
STATE2( (e.left sa) 0 e.right = <STATE2 (e.left) sa 0 e.right>;
        (e.left) X e.right = <STATE0 (e.left X) e.right>;
        (e.left sa) Y e.right = <STATE2 (e.left) sa Y e.right>; );
STATE3( (e.left) Y e.right = <STATE3 (e.left Y) e.right>;
        (e.left) = <STATE4 (e.left) >; );
STATE4( (e.final) = 'ACCEPT' ; );
```

This last example should indicate the ease with which lexical analyzers and parsers may be written in Refal -- a state transition table is designed, and then one sentence for each transition is written. If it is necessary to debug the scanner or parser, an indication may be made to the Refal interpreter to show the step-by-step transitions leading to the bug.

#### 4. Conclusions

The language Refal has been presented for use as a tool for investigating the operation of abstract machines. In this application, the simplicity of its design helps make such operation quite clear. A brief history of the language has been given, along with a description of methods of programming which may shed light on problems in the study of finite state machines. Examples of these methods have been given.

Implementations of Refal exist now on IBM 370 type machines, as well as on DEC VAX-11 type machines. The most recent version, completed in the summer of 1985, is written mostly in C, so as to allow for portability. Work is progressing currently on transporting the new implementation to both the IBM PC, and the Apple Macintosh computers.

### 5. References

- [1] Turchin, V.F., The Language Refal, The Theory of Compilation, and Metasystem Analysis, Courant Institute Technical Report No. 18, New York, Jan. 1980.
- [2] Turchin, V.F., Nirenberg, R.M., Turchin, D.V., Experiments with a Supercompiler, Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh, Aug. 1982
- [3] Nirenberg, R.M., The Mapping of the Refal Machine onto a Von Neumann Machine, Ph.D. dissertation, City University of New York, New York, 1983
- [4] Hopcroft, John E., and Ullman, Jeffrey D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.