Techniques for Performance Tuning REXX Interpreters—A Case Study of Regina

Anders Christensen
Sintef Runit

Techniques for Performance Tuning Rexx Interpreters—A Case Study of Regina

Anders Christensen
<anders.christensen@runit.sintef.no>

The Rexx Symposium for Developers and Users
Boston, May 1–4, 1994

Abstract

This article describes some of the techniques and methods used for optimizing the Regina interpreter, a REXX interpreter written in C, originally for Unix systems. The methods described first may be regarded as optimalization techniques in isolation, but they are also prerequisites for the last technique described here: the creation and maintenence of shortcut pointers from the parse tree to the variable structure.

1 Introduction

When tuning a program like a REXX interpreter for improved speed, a number of general techniques are used. Some of these are interesting in themselves, but not very specific to REXX interpreters. The scope of this text is to present some of the techniques that are closely related to the datastructures and operations of REXX interpreters.

2 Datatyping Variables

The fact that REXX is a typeless language is often described as one of its major advantages. Thus, it might be a great surprise to learn that one of the techniques boosting the performance the most, was introducing typed variables. Another technique was introducing typed expressions, which is described in the next section.

Internally, a Regina variable can hold either a string value, a numeric value, or both. When setting a variable, either a string or numeric value is set, depending on the context. Whenever the value of a variable is retrieved, it can be retrieved as either a string or a number. If a string value is retrieved for a variable currently holding only a numeric value, that value is converted so the variable holds both data type formats and then the string value is returned.

To understand the difference between these two formats, it might be instructive to look at their definitions in Regina.

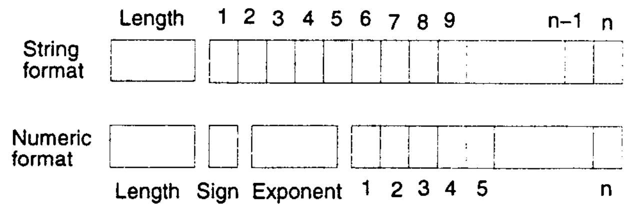

Figure 1: Storage formats for variables in Regina

For the string format, the values "2", " 2 ", and "2E0" are different, but for the numeric format, these are identical. The string is simply a sequence of characters, having a specific length. The numeric format is a sequence of decimal digits, to which there are connected three pieces of information: the length (number of digits), the sign, and the exponent (a native integer).

Consider the REXX statements:

	REXX statement	Numeric	String
1	`foo = 1+1`	2	N/A
2	`bar = foo || '.'`	2	'2'
3	`foo = (foo * 3) || ' '`	N/A	'3 '
4	`say foo*3`	3	'3 '

- After the first line, `foo` contains the numeric value 2, while its string value is not set. Note that it is not undefined, it can easily be converted from the numeric format, if necessary.
- In the second line, the string value of `foo` is retrieved, which means that the numeric value is converted. After the second line, both a numeric and a string value are stored for the `foo` variable.
- In the third line the numeric value of `foo` is retrieved and used in an expression which results in a string value. At the end of that statement, the `foo` variable is set to a new string value, and the numeric value becomes unset.
- In the fourth line the numeric value of `foo` is retrieved. However, at that point the `foo` variable have only a string value, so when retriving the value, the current string value is expanded to a numeric value. After the fourth line both a string and a numeric value are set.

Why maintain this double accounting? It turns out that variables set to a string value are very rarely used in numeric expressions . And vice versa, when a variable is set to a numeric value, it is seldom used in a string context; except for output statements, which tend to be slow anyway.

Based on these two observations, it makes sense to have two parallel, highly optimized sets of functionality for operating variables: one for numeric values and one for string values. Since the conversion between them are rather rare, the more time-consuming code for conversion between the two formats does not significantly increase the total execution time.

As a future extension, the scheme may be expanded to handle boolean variables too. However, it may turn out that the increased complexity this requires (six conversion types as opposed to only two above) may not justify the increase in speed. The use of boolean variables are much less widespread than string and numeric variables; and besides, boolean variables can be emulated through numeric variables.

In addition, the native floating point numbers could be used. It beats REXX numbers in speed, but it is difficult to avoid loosing accuracy wrt the definition of REXX arithmetics.

3 Construction of a Parse Tree

In order to explain what comes next, we need to know the format in which Regina stores a parsed REXX program. As an example, consider the following REXX code:

```
if ('xxx'/=bar) & (bar*foo>1000) then
   exit
```

Regina converts this sequence of tokens to a parse tree, the expression in the if-clause is shown in figure 2. The conversion between a sequence of tokens and a parse tree is described in most text books on compiler construction. As an aside note: parse trees are often considered to be incompatible with the customary way REXX programs are stored internally—a list of tokens. However, a static tree can easily be converted to a list of tokens. The difference lays in generating

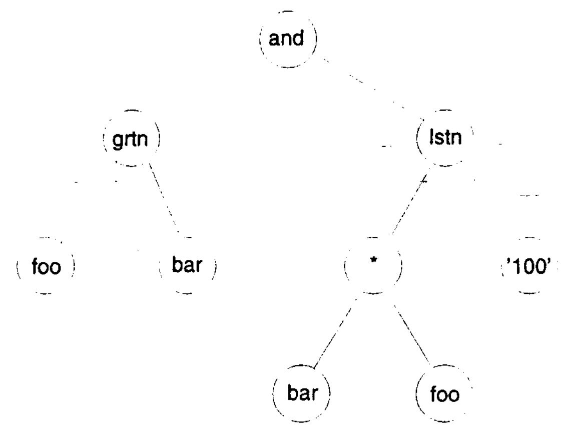

Figure 2: A parse tree built by Regina

a parse tree, which requires a more thorough analysis than a simple conversion of the source code to a list of tokens.

The most obvious approach for executing the code represented by the parse tree would be to traverse the parse tree, and for each binary operator ("=/", "&", "*", etc) first traverse the left subtree, then the right subtree, and in the end apply the operations to the two strings obtained from the traversals.

It is possible to add some optimalizations here:

`bar =/ 'foo'`
: We know that this must always be a non-numeric comparison, this there is no need to try anything but a normal string comparison straight away.

`2*bar`
: In this expression, we are only concerned with the numeric value of `bar`, so we retrieve its value in numeric "mode", as described in section 2.

`(a>b)&(c<d)`
: Here, each of the two pairs of parentheses can result in either "1" or "0". Thus, we use the native integer format of the computer to signify the values, rather than using the Regina string or numeric format.

4 Datatyping Expressions

Using these techniques, the dataformat of the data transmitted from a subtree to its parent node depends on the context. For instance, consider the parse tree shown in figure 2. After adding the datatypes, the new parse tree is shown in figure 3.

5 Hash Tables to Store Variables

Regina uses hash tables to store the variables defined at any given point during the execution of a REXX script. This technique can make the retrieval of a variable a constant-time operation, if given a well balanced hash table. However, once the hash table becomes full, the efficiency drops.

One of the key points with hash tables is to choose the correct size. If the size is too small, the handling of overflow adds a large overhead. If the table is too big, the extra work of initialization and deallocation adds unnecessary overhead. One solution is to have only one huge hash table for the whole interpreter, in which case the work of initialization and deallocation of the hashtable is done only once. However, this requires some extra overhead for insertion and deletion of variables.

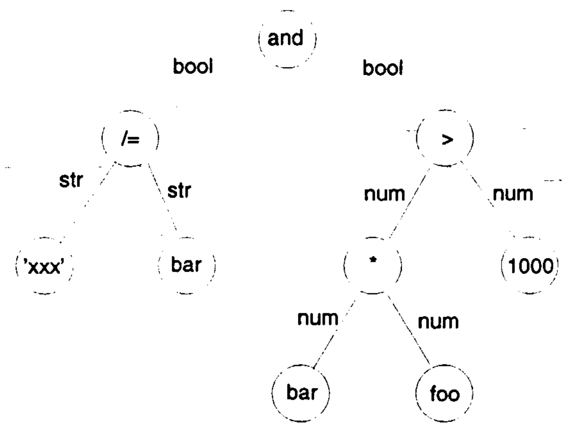

Figure 3: Parse tree with datatypes of transmitted results

Among other things, it makes the operations of deleting all tails of a particular stem a bit more complicated.

Another solution is to use dynamic hashing, where a small hash table is used initially, and the table is expanded when it is filled. The advantage of this technique is illustrated by the fact that the interpreter has no way of predicting the number of tails used by a routine at the entry of the routine. (Except that it may cache the number of tails used at earlier calls.)

Figure 4 shows how Regina stores its variables. There is one hash table for each subroutine having a PROCEDURE clause, and within each such hash table, there is another hash table for each stem in use.

6 Shortcut Pointers from Parse Tree

A well-known technique for optimizing computer code is to cache any value for which you may have need later. Regina makes use of this several places. For instance, whenever Regina executes a CALL clause or a function call for the first time, it must determine which routine to call. If the destination routine is an internal or built-in function, it is cached by setting to pointer in the parse tree to point to it.

7 Shortcut Pointers to the Variable Structure

Whenever a REXX clause refers to a variable name, the value of that variable must be retrieved from the variable structure. This involves some navigating, which can be time-consuming. However, it often turns out that multiple invocations of the same variable reference in a clause navigate through the variable structure only to end up at the same variable box. Thus, it may be advantageous to cache the result of the most recent navigation for each variable reference of the program. This means storing a pointer in the parse tree, pointing into the hash table of the variable structure.

Consider the following trivial code:

```
foo = 1
do 1000
   foo = foo + 1
   end
```

If we restrict the analysis to the contents of the loop, the variable `foo` is set 1000 times and its value is retrived 1000 times. I.e., navigating the variable structure 2000 times.

Then we add functionality for caching the result of each navigation. Neither retrieving nor changing the value of a variable are operations which change the identity of the box in the variable

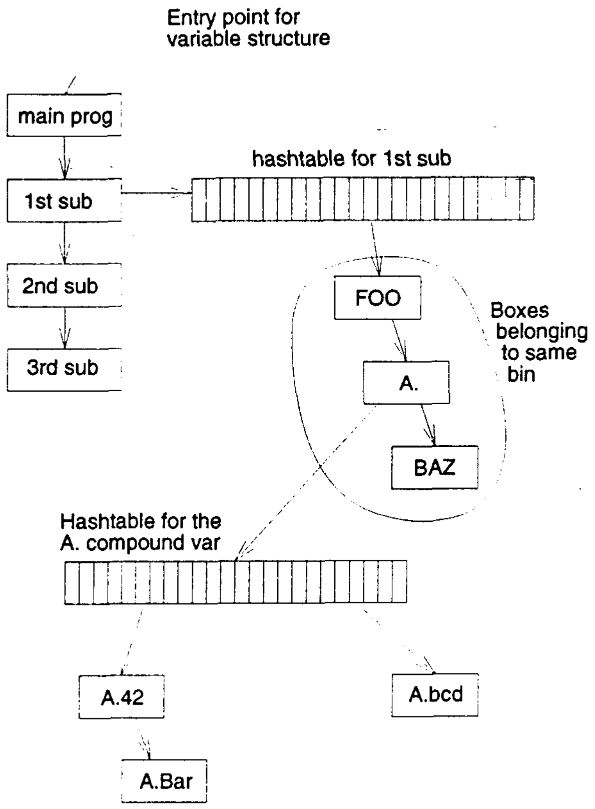
Entry point for variable structure
main prog
1st sub
2nd sub
3rd sub
hashtable for 1st sub
FOO
A.
BAZ
Boxes belonging to same bin
Hashtable for the A. compound var
A.42
A.Bar
A.bcd

Figure 4: The structure of variables in Regina

structure where the variable is stored. Thus, if we can cache a pointer to the variable, the number of walks through the hash table structure drops from 2000 to 2.

On the other hand, the cost of this is caching the pointer after each navigating walk, unless it was already cached. And the cost of verifying that the shortcut pointer is still valid. In particular the latter of these introduces a number of subtle points. Consider the following code:

```
/* first example */
call foo
exit

foo: do i=1 to 2
   say i
   if i=1 then do
      procedure
      i = 1
      end
   end
```

In this example (which is only allowed for TRL1—not TRL2) the variable `i` in the `SAY` clause refers to different variables during the first and the second iterations of the loop. This is due to the execution of the `PROCEDURE` clause during the loop, which changes the scope of the `i` variable. Thus, the shortcut pointers cached during the first iteration must be tested during the second iteration, and the fact that they are invalid must be detected.

This is achieved using a generation number, which is identical to the number of currently nested functions having executed the `PROCEDURE` clause. Whenever a new `PROCEDURE` clause is executed, the generation number is incremented, and whenever a `RETURN` clause is executed for a routine which have—during its course—executed a PROCEDURE clause, the generation number is decremented.

To verify the validity of a shortcut pointer, the current generation number is recorded in the box pointed to by the pointer. Whenever a recorded pointer is to be validated, it is considered invalid if the current generation number is greater than the number recorded in the box pointed to by the shortcut pointer (i.e. a PROCEDURE clause has been executed since this pointer was made, invalidating the pointer). In this case, the recorded shortcut pointer is attempted deallocated, and the variable is located using the standard procedure—the new location is of course cached if the current generation number is greater than the recorded number.

The next example shows a function.

```
/* second example */
say bar(3)
exit

bar: procedure
   parse arg i
   if i=1 then
      return 1
   else
      return bar(i-1)*i
```

Here, the last clause in the routine is executed twice, as a result of the recursion. However, due to the rules for evaluation of REXX expressions, the retrieval of the i variable at the end of the last clause is executed twice: first at end of the second invocation of bar, and then at the end of the first invocation of bar. (Note: i is referred to after the recursion itself.)

According to the rules outlined above, the shortcut pointer is cached at the end of the second call to bar (the first recursive call). Thus at the end of the first call to bar, this cached value is picked up, but the generation number does not match (the recorded generation number is greater than the current generation number), so the shortcut pointer is discarded and the variable is located using the standard procedure (i.e. since the pointer was made, the routines in which it was made has been terminated).

There is another, less subtle point here, too. All variables local to the second (recursive) call to bar are discarded when that routine returns. Thus the shortcut pointers appear to point to undefined memory! This is easily fixed by maintaining a counter with each variable box. Whenever a shortcut pointer is set to point to that box, the counter is incremented; and whenever a shortcut pointer is removed from pointing to a variable box, the counter is decremented. As soon as this counter mechanism is in place, a variable box can be marked for deletion, and retained until all shortcut pointers to it have been killed.

```
/* third example */
do i=1 to 2
   call bar
   end
exit

bar: procedure expose i
   if i=1 then
      foo = ''
   say foo
   return
```

Here, the second invocation of bar finds the cached pointer in the SAY clause, but the pointer is invalid, even though the generation number is correct. To handle this case, variables discarded during the execution of the RETURN clause are not immediately discarded if there are any shortcut pointers pointing to it (as recorded by the shortcut counter field). Instead, it is suspended until all shortcut pointers point elsewhere, at which time the box is deallocated. In the meantime a flag is set for the variable box, so that the interpreter can discover that the box is invalid if it tries to dereference the shortcut pointer.

```
/* fourth example */
do i=1 to 4
   if i=2 then do
      drop i
      i = 2
      end
   end
```

The code of the fourth example, as shown above, illustrates why the delete flag is necessary. The variable box created at the start of the loop is dropped during the loop, so a mechanism is necessary to detect that the box is invalid at the start of the next iteration.

8 Algorithms

The two algorithms shown are the the central for the correct operation of the shortcut pointers in Regina. The first algorithm is shown in figure 5, and describes how to access (retrieve or update) the value of a variable. To be effective, it requires that the code has been executed at least once before, so that shortcut pointers have been created.

```
foo is a variable reference to access
 if exists a shortcut pointer for var then
   if points to a variable not deleted then
     if the generation number is correct then
       retrieve/set the value
       return
     else
       decrement counter
       remove shortcut pointer
   else
     decrement counter
     remove shortcut pointer
 if counter=0 then
   delete/deallocate variable box
access variable "the hard way"
cache the found box in the shortcut pointer in the parse tree
increment counter
return
```

Figure 5: Retrieving/setting value of variable reference in the parse tree

The second algorithm is used to delete variables during the execution of the RETURN clause from a routine which had its own "private" PROCEDURE clause. It will always detach the variable boxes, but it will only deallocate the space if there are no shortcut pointers pointing to the box (as recorded by the counter in the box).

```
for each local variable
  disconnect it from variable system
   if counter is greater than 0 then
     mark variable box as deleted
   else
     deallocate variable box
```

Figure 6: Deleting local variables at return from routine

9 Why So Complicated?

Most computer languages keep track of their variables in much easier ways, so why introduce this complexity for REXX? Because of the enormous degree of freedom in REXX. REXX does not have compile-time routines, it has "only" run-time routine entry and exit points! Therefore, it is virtually impossible to bind a given clause to a particular "routine" at parse time. The possibilities of "`SIGNAL ON`" and "`INTERPRET`" ensure that control can pass from virtually any clause to virtually any label in a REXX program.

Thus, the techniques used for most compilers and some interpreters, which allow them to bind the variable references in the source code to specific locations at compile- or parse-time do not work for REXX, and more elaborate systems, like the one described above, are called for.