Improving the Reversible Programming Language R and its Supporting Tools

Christopher R. Clark
cclark@cise.ufl.edu

CIS 4914 Senior Project

Advisor:
Dr. Michael P. Frank
mpf@cise.ufl.edu

December 3, 2001

Abstract

This project involved improving the functionality and performance of a reversible programming environment developed previously by a team at MIT. Enhancements were made to the R reversible programming language, the R compiler, and the Pendulum reversible processor architecture simulator. Support for advanced condition testing, character strings, and data output formatting was added to R. The compiler was made easier to use and faster. Finally, the emulator was updated to support the new language features. The results of this project serve as a step in the evolution of the R programming tools. Suggestions for future work are presented.

1 Introduction

1.1 Background

Reversible computing means using only computational operations that can be exactly reversed, or undone. Reversibility can be applied to any or all levels of a computer system—circuits, architectures, programming languages, and algorithms. This project deals with a reversible programming language and a reversible processor architecture. At these levels, reversibility provides support for exploring interesting reversible algorithms.

1.2 Related Work

The foundation for this project is the reversible programming language R and the Pendulum reversible processor architecture, which were both originally developed at MIT by Dr. Frank, Carlin Vieri, and others. Dr. Frank also developed a compiler for R that targets the Pendulum architecture. Matt DeBergalis (MIT) developed a simulator for Pendulum assembly language programs, known as PendVM.

1.3 Motivation

The R language and compiler were developed as a proof-of-concept and to simplify writing programs for the Pendulum architecture. However, the functionality of R and the efficiency of the compiler were less than ideal. Improving these two areas is the main emphasis of this project. PendVM will also be modified to support the new functionality of R. The improved versions of R, the R compiler, and PendVM will provide future programmers with more capabilities and more productivity.

2 Research

2.1 Studying Existing Implementation

The first task of the project was to gain understand the current state of the R language, the R compiler, and the Pendulum virtual machine. This involved reading the relevant chapters of Dr.

Frank's manuscript [1], which include the specifications of the language and the compiler. After reading this material, I studied the source code for the compiler, which is written in Common Lisp. I wrote several simple R programs and compiled them using the R compiler in debug mode, which displays the output after each step in the iterative compilation process. By studying this output, I was able to understand the internal workings of the compiler. Next, I studied the source code of the Pendulum emulator, PendVM, which is written in C. I ran some sample programs to learn the emulator's interface.

2.2 Gathering Desired Features

The next phase of the project was to compile a list of desired features and determine which features to implement. I created an initial list though observations made while studying the existing documentation and source code. I also asked other members of the Reversible and Quantum Computing Group who use the R tools to contribute feature requests to the list. Finally, I presented the list to Dr. Frank, who provided some additional feature ideas and made some suggestions on the implementation of others. I prioritized the list based on each feature's benefit (usefulness) and cost (time to implement) and chose the top few to implement in this project.

2.3 Literature

Most of the literature used for this project consisted of reference material used while modifying the source code for the compiler. Dr. Frank's manuscript [1] was used extensively for its details on Pendulum, R, and the R compiler. Some Lisp references I used frequently were *Common Lisp, the Language* [2] and the *Common Lisp HyperSpec* [3]. I also read an overview of Lisp programming in *The Art of Lisp Programming* [4].

3 Design and Implementation

For the rest of this document, I will refer to the versions of the R specification, R compiler, and PendVM emulator that existed prior to this project as the *original* versions. I will refer to the modified versions that resulted from this project as the *new* or *enhanced* versions.

3.1 Compiler Output

The original compiler printed all output to the console. This required the user to redirect the output to a file and then manually edit the file to remove extraneous information and add a header line in order to run the program in PendVM. This is tedious and time-consuming. With the new compiler, the output is still written to the console, but the final assembly code is also automatically written to a file. The Pendulum assembly language (PAL) code file has the same name as the input file but with a .pal extension. This file contains the necessary header so that the file can be executed by PendVM without modification.

3.2 Conditional Execution Construct

The conditional execution statement (if statement) available in the original version of R was very primitive. The condition could only be a single relational comparison between two expressions. In addition, it did not support an else clause. In this project, the conditional execution construct was enhanced to support combining multiple comparisons using Boolean operators and an optional else clause. The enhanced conditional execution statement in R has the following syntax:

```
(if condition
        if-statement_1  if-statement_2  …  if-statement_n
else
        else-statement_1  else-statement_2  …  else-statement_n
)
```

The *condition* expression can be a Boolean combination of relational comparisons. The supported Boolean operators are AND (&&), OR (| |), and NOT (!). The supported relational operators are =, !=, <, >, <=, >=. Parentheses must be included such that each operator has exactly one parenthetical expression on each side (except for NOT, which must have exactly one expression on its right). The entire expression must also be surrounded by parentheses. The following is a valid example:

```
( ( (x < y) && (y != (z+1)) ) | | (! z) )
```

3.3 Character Strings

In the original version of the R language and compiler, the only supported data type was 32-bit signed integers. This limited the types of useful programs that could be written and made program output difficult to decipher. The programming potential was greatly increased by the addition of a character-based data type. The enhanced versions of R and the compiler now support the use of character strings in programs.

3.3.1 R Constructs

R now allows the programmer to define character strings as static memory variables. Since copying a string into a non-empty string variable cannot be done reversibly without generating garbage data, currently all string variables must be defined statically at compile-time. However, it is reversible to copy a string into an empty string variable by simply adding the value of the string to the empty string. This can be reversed by subtracting the value of the string. Conceivably, this could be used to read input from the user or from a file. As there are no input constructs in R, this functionality is not currently supported. However, the string declaration construct does allow the programmer to reserve an empty block of memory that could eventually be used for storing such input data.

A string variable can be declared using the defstring construct and the following syntax:

(defstring *name "string"* <*length*>)

The *length* is an optional parameter. If it is omitted, the compiler uses the length of *string*. If it is included and less than the length of *string*, then *string* will be truncated to *length*. If *length* is greater than the length of *string*, empty space will be allocated after the end of *string*. If *string* is empty (" ") then an empty string of *length* will be created.

3.3.2 Compilation

The compiler generates a list containing the 8-bit ASCII code of each character in the string and adds a null character (zero) at the end. It then packs each sequence of four characters into a 32-bit word with the first character in the low-order bit position. A list of these words is used to create a static array declaration, which is compiled to store the words in consecutive memory locations. Figure 1 illustrates this process.

String declaration:	(defstring h "hello")
Character value list (hexadecimal):	(48 65 6C 6C 6F 0)
Packed word list (hexadecimal):	(6C6C6548 0000006F)
Array declaration:	(defarray h #x6C6C6548 #x0000006F)
PAL memory allocation (decimal):	H: DATA 1819043176 DATA 111

Figure 1: String compilation process

3.4 Data Output

Previously, the output functionality of R consisted of two functions: printword and println. The printword function output a variable in signed integer representation and println simply output a newline representation.

3.4.1 New R Functions

A new output function called print is a general-purpose output function. It can be used to output the contents of a register, a static memory variable, a dynamic memory variable, a static string variable, or a literal string. It also supports options for specifying parameters of the output representation. The println function was enhanced to accept all the same data types and options as print. When println is used without any arguments, it outputs just a newline representation as in the original version. The printword function is still available for compatibility with existing programs.

The print and println functions will output any type of data and allow the programmer to specify options to convey to the run-time environment how the data should be displayed. For integer variables the display options consist of signed two's complement, unsigned two's complement, decimal, and hexadecimal formats. The R program and its output in Figure 2 below demonstrate the use of the output functions.

```
(defword x 16)
(defstring h "hello")
(defstring w " world")

(defmain main
  (print h) (println w)                          hello world
  (print "Base 10: ") (printword x) (println)    Base 10: 16
  (print "Base 16: ") (println x  :base 16)      Base 16: 10
)
```

Figure 2: R output functions

3.4.2 Output Convention

When compiling an output statement, the compiler automatically determines the data type of the item being output. To allow the run-time environment to determine how an output word should be represented, the compiler outputs a header word before each output data item. The header indicates the data type of the subsequent word and how it should be displayed. A newline is represented as a header word only.

PendVM was modified to support this new output convention. When the first output statement is encountered, it is interpreted as a header word. If it is the newline header, PendVM prints a newline to the display and the next output is interpreted as a header word. If it is another header type, the header is saved and the next output word is printed based on the type and options included in the saved header word.

3.5 Pendulum Assembly Code Optimizations

Currently, the compiler compiles each R instruction individually from source code form all the way to Pendulum assembly code form. Therefore, it cannot perform any optimizations across instructions. For example, an intermediate value that is used by two consecutive instructions will be computed twice. Additionally, to ensure reversibility, the computed value must be uncomputed after each use, leading to the following general sequence of events:

1. Computation of intermediate value
2. First use of intermediate value
3. Uncomputation of intermediate value
4. Computation of intermediate value
5. Second use of intermediate value
6. Uncomputation of intermediate value

Obviously, this is terribly inefficient, especially since the computations of items 3 and 4 exactly cancel each other out. This means that the entire sequence of instructions comprising the first uncomputation and the second computation perform no useful work. To reduce these types of inefficiencies, the new compiler now performs an optimization scan of the assembly code after the normal compilation process is completed. The compiler is able to recognize and eliminate unnecessary uncomputation and re-computation of values. The R fragment and its compiled PAL code (before optimization) in Figure 3 show the type of optimization performed by the post-compilation scan. All of the PAL instructions in bold print are removed by the optimizer. In this specific case, the instruction count is reduced by 33 percent in the optimized version.

```
(if (x) then
  (print 1)
else
  (print 0)
)
```

```
                ADDI $3 X
                EXCH $4 $3
                ADD $2 $4
                EXCH $4 $3
                ADDI $3 -X
_IFTOP:         BEQ $2 $0 _IFBOT
                OUTPUT $3
                ADDI $3 1
                OUTPUT $3
                ADDI $3 -1
_IFBOT:         BEQ $2 $0 _IFTOP
                ADDI $3 X
                EXCH $4 $3
                SUB $2 $4
                EXCH $4 $3
                ADDI $3 -X
                ADDI $3 X
                EXCH $4 $3
                ADD $2 $4
                EXCH $4 $3
                ADDI $3 -X
_ELSETOP:       BNE $2 $0 _ELSEBOT
                OUTPUT $3
                OUTPUT $3
_ELSEBOT:       BNE $2 $0 _ELSETOP
                ADDI $3 X
                EXCH $4 $3
                SUB $2 $4
                EXCH $4 $3
                ADDI $3 -X
```

Figure 3: Compiler optimization

4 Results

4.1 Verification

To ensure that the new compiler produced correct output, some sample programs were compiled using both compilers, and the output of each was compared. The Schrödinger wave equation simulation and multiplication algorithm, which were know to produce correct results with the original compiler were used for this test. For these programs, the PAL code produced by the original compiler and the PAL code produced by the new compile (before optimization) were identical. This verifies that none of the original functionality of the compiler was corrupted during the modification process.

4.2 Compile Time

One of the complaints from users of the R compiler was that the compilation process took too long. To improve performance, some tweaks were made to the main loop of the compiler that transforms the R source code into PAL code. To test the relative performance of the original and new compilers, the time to compile two sample R programs was measured for each compiler. In addition, the new compiler was compiled to a LISP binary executable and included in this test. The average of three trials was used for each combination of program and compiler. All tests were run on a system with an Intel Pentium 3-733 and 512MB RAM using CLISP 2.27. Figure 4 contains the results.

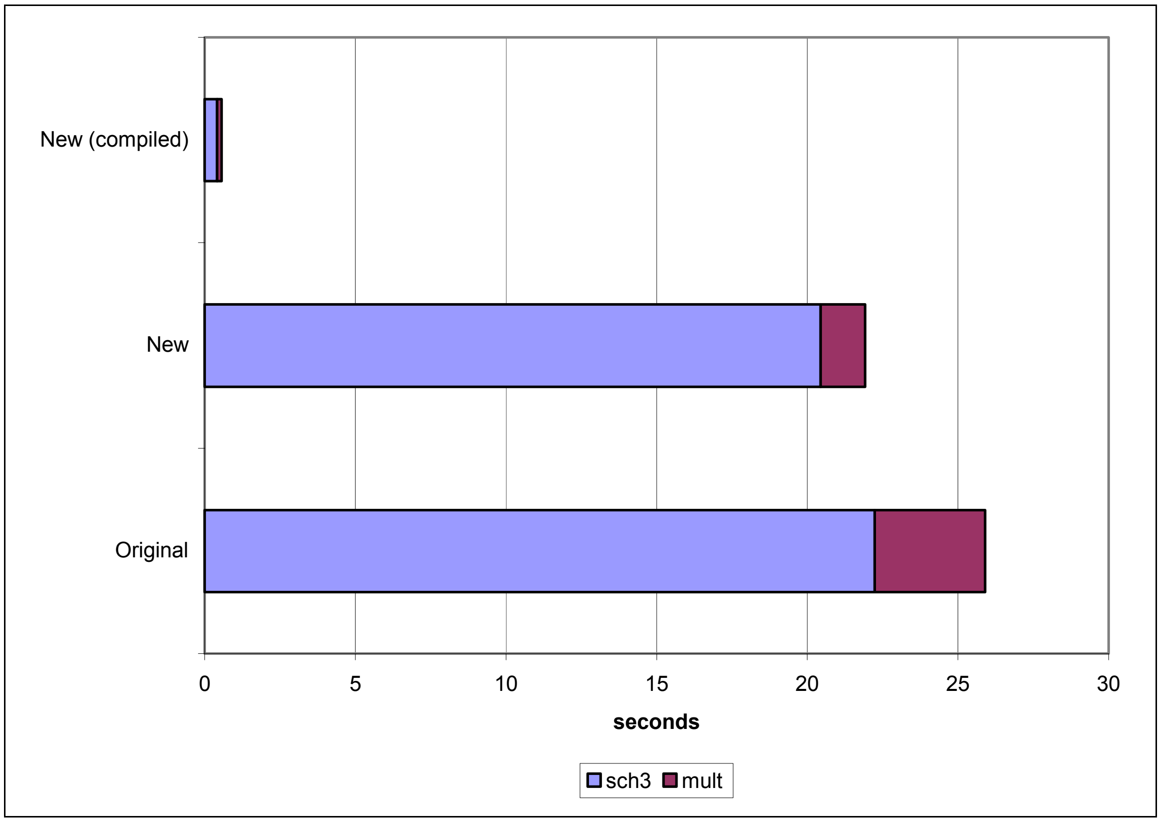

Figure 4: Compilation time comparison of the original and new compilers. The time is the CPU time spent executing the main loop of the compiler. The I/O time for reading the input file and writing the output is not included in this figure.

The new compiler is an average of 34 percent faster than the original compiler for the test programs. The compiled new compiler is an average of 94 percent faster than the interpreted version. The compiled compiler is able to compile both programs in about one-half of a second. This compiler should be quick enough to compile a complex program in a reasonable amount of time.

4.3 Code Efficiency

The types of post-compilation optimizations discussed in section 3.5 are only effective in certain limited situations. As shown in the example in there, when the if-else construct is used, a significant number of instructions can be eliminated. However, for the Schrödinger wave equation simulation, the optimizer only reduces the instruction count by two percent, and no reduction is made for the multiplication program. It is apparent that optimization may be more effective if done at an earlier stage in the compilation process.

5 Conclusion

I found this project to be a useful learning experience. I had only written a couple LISP programs previously, so, needless to say, I learned a lot about LISP while working on the compiler. I also learned to appreciate some things about developing a large program in general, such as the value of well-placed comments in the source code and modular design. I had never written a compiler before, so it was interesting to work on modifying one. I also learned about reversible computing theory and practical aspects for implementing reversible operations. I hope current and future users of R and its supporting tools find the enhancements of this project beneficial.

6 Future Work

6.1 Floating Point

One notable feature missing from R is the support of non-integer data and floating-point operations. Since the current Pendulum architecture does not contain floating-point hardware, floating-point arithmetic operations would have to be done in software. One possible approach to developing a floating-point library for R would be to translate a library from an open-source C implementation.

6.2 Evolution of R

If R is to become widely used, some changes should probably be made. Most programmers would find the current syntax unfamiliar, so it may be better to adopt a syntax more like C or Java. Support for function calls with return values, structures, and dynamic memory allocation are some other important missing features.

7 Acknowledgements

I would like to thank Dr. Frank for his willingness to support me on this project and for his assistance in completing the project. The detailed comments he included in the source code of the compiler were also greatly appreciated! I would also like to acknowledge Dr. Schmalz for his general support of the Senior Project class and for providing helpful tips throughout the semester.

8 References

[1] Frank, Michael P. “Reversibility for Efficient Computing”, unpublished ms. (1999)

[2] Steele, Guy L. *Common Lisp, the Language, Second Edition*, Woburn, MA: Digital Press (1990)

[3] Pitman, Kent, ed. “Common Lisp HyperSpec”, http://www.lisp.org/HyperSpec/FrontMatter, The Association of Lisp Users (as of 3 Dec 2001)

[4] Jones, Robin, Clive Maynard, and Ian Stewart. *The Art of Lisp Programming*, London: Springer-Verlag (1990).

9 Biography

I am graduating from the University of Florida in December 2001 with a degree in Computer Engineering. After graduation, I plan to pursue a master’s degree in Computer Engineering. During the past six summers, I have worked at Eglin Air Force Base on various computer-related projects including web design, database design, and automatic target recognition using image processing and artificial intelligence techniques. I have also done some general computer consulting for various small businesses and individuals performing tasks such as system building, troubleshooting, and network setup. I hope to eventually start my own business doing work in some computer-related area.