Development of an Interpreter for LRT using the Exact Real Number Paradigm

J. Leonardo González-Ruiz, J. A. Hernández-Servín, J. Raymundo Marcial-Romero

Facultad de Ingeniería, Universidad Autónoma del Estado de México, Toluca, México
leon.g.ruiz@gmail.com, xoseahernandez@uaemex.mx, jrmarcialr@uaemex.mx

Abstract. The exact real number representation can grow arbitrarily so it does not truncate or rounds up as opposed to floating point number representation. The advantage of this representation is that not rounding errors are generated and any operation can be achieved with the desired accuracy. The LRT is a proposal that implements exact real number. This paper develops an interpreter for LRT using this paradigm whose operational semantics is based on sixteen rules so the programs based on LRT libraries are less complicated to debug. One of the main problems in implementing LRT has been memory consumption. The main contribution of this work is that LRT has its own administrator for infinite lists as well as its own lazy evaluation. The performance of programs written in LRT interpreter is shown to be superior to the libraries of functions in both execution time and use of memory.

Keywords: Exact real number, Interpreter, lazy evaluation, LRT, interval arithmetic.

1 Introduction

The paradigm of exact real numbers arises from the need to represent real numbers on a computer, compared to traditional floating point representation [6]. The real numbers may be described and represented in various forms for formal purposes in mathematics, for example, represented by the intersection of intervals [15].

For practical purposes, the common representation of real numbers is performed by finite strings of decimal digits$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and a dot $\{.\}$.This representation is commonly used and is called "floating point". The set of digits can represent only a subset of the real numbers exactly, this means that most real numbers must be represented by other real numbers that are close to, or, for a interval of rational numbers which delimit the actual number requested, causing rounding errors. Despite this, floating point representation is acceptable for a wide range of applications, however, the accumulation of these rounding errors produced by a large number of computations produces inaccurate results. For example, programming languages such as C, which uses floating point arithmetic causes rounding errors in complex operations that require precision of significant digits, as an alternative of floating point we have used the interval analysis[15]. Interval analysis involves expressing a real as a pair of numbers which represent an interval containing the number. Interval arithmetic operations compute new upper and lower bounds on the result after the operation has been performed. One way to represent a real number using interval analysis is digit strings of potentially infinite size, this representation is known as exact real numbers paradigm[3]. There are exact real number implementations like IRRAM [12], IC-Reals [4], RealLib [7], that expect to become the standard for use in program languages.

Another option for calculating real numbers is the language called LRT (Language for Redundant Test) [11]. The language LRT is a variant of PCF [5] and an extension of PCF (Programming Computable Functions) with a ground type for real numbers and suitable primitive functions for real-number computation. LRT efficiency is given by the choice of linear functions, handling of potentially infinite lists and type of language evaluation. An implementation of this language was developed by Lucatero [10], that developed a library for handling exact real numbers in the programming language `C++`.

In this paper an interpreter for the LRT language is described, this interpreter has the ability to handle infinite lists and *lazy* evaluation. The development of this interpreter merge two languages, LRT (for the real number calculation) and `C-` [9] (for the basic grammar in a language) producing the new language LGR. The scripts written for the interpreter will be compared with developed implementations of Lucatero's library because it is considered the continuation of her research.

2 Implementations

A real number is computable if there is an algorithm that can produce all its digits. The representation of real number in a computer can be handled by different approaches ranging from libraries of functions in the C programming language to applications in a functional language [10].

2.1 LRT Language

The LRT language (Language for Redundant Test), its a variant of *Real PCF* [5] developed by Marcial et al. in 2004. LRT is a nondeterministic language, eliminates the parallel conditional of *Real PCF* that is a parallel language and add a nondeterministic operator `Rtest` [8].

A real number x can be represented in the LRT language as a tuple:

$$(y, 2^e)$$

where $y \in [-1, 1]$ y $e \in \mathbb{Z}$.

LRT uses the representation (mantissa, exponent), where the mantissa is represented by a infinite list and the exponent as a integer number. This representation allows LRT extends its rank to the entire real number line, with the restriction that the mantissa must be in the interval $[-1, 1]$.

The constructors `Cons`, `Tail` y `Rtest`, are the base of the real number construction in the LRT language.

- *Constructor* ***`Cons`***: It is a function that accepts as input two intervals and returns an interval, is the one that performs a reduction intervals and is defined as follows:

$$\texttt{Cons}([\underline{x},\overline{x}]) = [\underline{x}\overline{a} + \underline{b}, \overline{x}\overline{a} + \underline{a}]$$

  where$[\underline{a},\overline{a}]$ y $[\underline{x},\overline{x}]$ are two intervals of rational numbers in the interval$[-1,1]$.

- *Constructor* ***`Tail`*** *:* It is a function that uses as input two intervals of rational numbers belonging to the interval $[-1,1]$. It is considered the inverse of the constructor `Cons` only when the interval $[\underline{x},\overline{x}]$ is contained in $[\underline{a},\overline{a}]$. In this case the following equation is considered:

$$\texttt{Tail}_{[\underline{a},\overline{a}]}[\underline{x},\overline{x}] = \frac{x - (\frac{\overline{a}+\underline{a}}{2})}{(\frac{\overline{a}-\underline{a}}{2})}$$

When $[\underline{x},\overline{x}]$ is not within $[\underline{a},\overline{a}]$, has to guarantee that the result is within the interval $[-1,1]$, since this is the range of the function.

- *Constructor* `Rtest`: This function determines which side the rational number lies within the interval $[-1,1]$, that is the `Rtest` returns $True$ if the number to be calculated lies in the positive side and $False$ otherwise. LRT language is called "redundancy check language" because the constructor `Rtest` is an operator of redundant verification, ie that there are numbers for which the operator can return both true and false [10].

2.2 LRT Implementations

Two calculators were developed based on LRT language. One developed by Villanueva as described in [16], and the second one is a modified version developed by Linares [8]. Villanueva implements the basic operation using the functional language Haskell by considering real number in the interval $[-1,1]$, his calculator gives accurate results but it is limited to basic operations. Specifically, he implemented addition, subtraction, multiplication and division only. Linares, on the other hand takes Villanueva's implementation and manages to add improved algorithms to compute transcendental functions such as sin, cos, tan, tan^1, e, and log. fundamentally based on algorithms developed in [14] by adapting the type of data used in the LRT language.

The calculator developed by Linares is more efficient than the one by Villanueva and also the range operation is expanded to the entire line of real numbers; in spite of that neither of them improve implementations developed in the programming language `C++` such as iRRAM [10].

To improve these calculators, Lucatero [10] developed a library for handling exact real numbers in the programming language `C++` [8]. The algorithms and the operational semantics of LRT language are implemented in `C++` language using `FC++` which allows the use of calls by need (*lazy* evaluation) and the definition of infinite lists on an imperative language as is `C++`, both necessary for the operation of the LRT semantics; in addition to use the floating point number of GMP library for greater accuracy in the LRT constructors.

The comparisons made by Lucatero, show significantly improved runtime operations compared with the implemented in a strictly functional language, which also holds the operational semantics of the LRT language. The resulting time is longer compared to the one developed in `C++` that also uses the exact real computation, as iRRAM. Lucatero concluded that memory consumption increases as more precision digits are required in the calculations; this is due of the recursive call used in the algorithms, because it must be stored in memory of the new state of function call. The library developed by Lucatero has been compared to commercial implementations as mentioned above, so, this research is considered a continuation of her previous work in search of an interpreter implementation more efficient at runtime.

3 Methodology

To implement the interpreter a methodology based on a standard implementation [1] of a compiler is used, which consists of the following stages of development: Lexical Analyzer, Syntactic analyzer, Semantic Analyzer, generate intermediate code and optimize code.

The implementation of this methodology begins with two basic languages, LRT and `C-` [9], the merge of these languages forms LGR language, having the main characteristics of the two languages. The first one can compute exact real numbers, and the second one, is a reduced programming language which grammar is used as the basis for the development of interpreter in this research, its semantics can be consulted in [9]. The interpreter is developed using the programming language C, alongside with the grammar of LRT and `C-`.

3.1 Design the Lexical Analyzer

At this stage the source code of LGR is read, which is composed by a character string. The analyzer collects character sequences into meaningful units called *tokens* to be used for the following stages. The lexical units used in this interpreter are integers, floats, booleans, real, rational comments, reserved words and blanks. For design the lexical analyzer we use Flex[13], that is a tool for generating lexical analyzers used for the development of the interpreter. This program reads a text file which contains C code that shows rules of regular expressions, this file can be compiled, generating an executable file, which is capable of analyze an input code in order to find regular expressions and execute its corresponding code in C.

For the develop of an lexical analyzer we must define some rules called regular expressions which represent patterns of strings. A regular expression is defined by a set of strings that matches [1]. This language made by a pattern of characters, is used to define different lexical elements of the interpreter, for a example, a regular expression of a real number is shown as follow:

A real number can be represented as an interval succession of the form: $[(((digit+/digit+),(digit+/digit+))),((digit+/digit+),(digit+/digit+)))*,...]$.

An other example of regular expressions are the reserved words of LRT language:

- *cons*: `Cons` function.
- *tail:* `Tail` function.
- *rtest:* `Rtest` function.
- *succ:* `Succ` function.
- *pred:* `Pred` function.
- *iszero:* `Iszero` function.

3.2 Design the Syntactic Analyzer

The second stage of an interpreter is the syntactic analyzer. In order to develop it, we use GNU bison [2] that is a general-purpose parser generator available for almost all operating systems, normally used in conjunction with Flex. Bison converts the formal description of a language, written in BNF (a formal way to describe formal languages), in a program written in C that performs parsing.

The parser receives the LGR source code in a token form from the lexical analyzer and performs the parsing call, which determines the structure of the program.

A part of the LGR grammar in its real part, is described below:

$\langle$*type specifier*$\rangle$	::=	‘INT’|‘FLOAT’|‘REAL’|‘BOOL’|‘RATIONAL’|‘VOID’
$\langle$*statement*$\rangle$	::=	$\langle$*expressionstmt*$\rangle$ | $\langle$*compoundstmt*$\rangle$ | $\langle$*selectionstmt*$\rangle$
	|	$\langle$ *iterationstmt* $\rangle$ | $\langle$*returnstmt*$\rangle$ | $\langle$*natstmt*$\rangle$ | $\langle$*realstmt*$\rangle$
$\langle$*natstmt*$\rangle$	::=	‘SUCC’ ‘(’$\langle$*statement*$\rangle$‘)’ | ‘PRED’ ‘(’$\langle$*statement*$\rangle$‘)’
	|	‘ISZERO’ ‘(’$\langle$*statement*$\rangle$‘)’
$\langle$*realstmt*$\rangle$	::=	‘TAIL’ ‘[’ ‘RATIONAL’ ‘,’ ‘RATIONAL’ ‘]’ ‘(’$\langle$*statement*$\rangle$‘)’
	|	‘RTEST’ ‘NUM’ ‘NUM’ ‘(’$\langle$*statement*$\rangle$‘)’
$\langle$*factor*$\rangle$	::=	‘(’$\langle$*expression*$\rangle$‘)’ | $\langle$*var*$\rangle$ | $\langle$*call*$\rangle$ | ‘NUM’ | ‘NFLOAT’ | ‘FALSE’ | ‘TRUE’ |‘RATIONAL’ | ‘CONS’ | ‘[’‘RATIONAL’‘,’‘RATIONAL’‘]’ ‘(’$\langle$*statement*$\rangle$‘)’

3.3 Design the Semantic Analyzer

The next stage of development is the semantic analyzer, which is responsible for giving meaning to the *tokens* and the structures formed on a user entered code. LGR is a typed language and a type declaration has the form $C\ x$ where x is a variable and C is a type. Each variable must have a type declaration before is used, the types of the attributes of a function must also be declared. Types used for this interpreter are:

- int, Integer type.
- float, Float type.
- real, Real type.
- bool, Bool type.
- rational, Rational type.

When a function is invoked in the interpreter, the parameters join arguments and the expression is evaluated; the value obtained is the meaning or reason for the invocation of the function. In other hand LGR allows recursion, adding elements such as lazy evaluation that is discussed below. An structure that the interpreter uses for general purpose is a symbol table, which is a data structure that uses the translation process of a programming language, where each symbol in the source code of a program is associated with information such as location, type of data and the scope of each variable, constant or function. A common implementation of a symbol table can be a dynamic table, which will be maintained throughout all stages of the compilation process.

The lexical analyzer in the previous step forms a parse tree, which is a representation of the code structure of a string tokens, the tokens appear as leaves of the parse tree from left to right and internal nodes tree depicting the steps of a derivation [1]. However, a parse tree contains information that is not necessary for the interpreter to be able to produce an executable code. Therefore, a new tree is generated using only the information needed for compilation and interpretation.These trees represent abstractions of token sequences of source code that the user entered, and sequences of tokens can not be recovered from them. However, it contains all the information needed for making more efficient the grammatical trees translation, these are known as abstract syntax trees, or AST [9]. A parser will cover all the steps represented in a parse tree, in order to obtaining a correct syntax, then an abstract syntax tree is constructed.

An other major task of the interpreter is to keep the information integrated and updated of the data types, the use of this information ensures that each part of the program makes sense and consistency under the rules of language, in this case LGR, ie , type checking. This test is developed in the syntactic part. When the interpreter produce the grammar tree, also performs a verification type. This operation is based on finding a variable, looking for the variable in the symbol table to know its type which it was declared. Subsequent to solve the expression, either an assignment, an operation or an invocation to a function, the interpreter verifies that the data type in each of them is the same, otherwise sends an error in the output of the interpreter. In this way ensures that the program does just what it must do, to have uniformity in the type of data, their operations and their return values.

3.4 Generate Intermediate Code

At this stage an interpreter that takes as input the AST generated by the above stage. The functions described below are responsible for interpreting the user-written code, including its resolution using lazy evaluation.

– *reduce* function*:* The purpose of the interpreter is to solve the operations described in the AST, reductions are performed in order to leave a code in its minimal form to achieve the intended operation specified by the user.
– *lazyreduction* function*:* To solve a source program in a *lazy* way, functions are implemented that allow code execution using this form of evaluation.

To achieve *lazy* evaluation, specific functions are implemented to solve the parameters of a function in a lazy way, in other hand functions that allow handling of recursion, detailed below are implemented.

– *ftocall* function*:* It is performed whenever there is a call to a function. A stack that contains two columns is created, the first stores the variable that invoked the call or on which the result will be reflected back . In the second column, the name of the function that was invoked is stored. In this way, the interpreter is able to be solving the recursive calls and go freeing up memory once the evaluation of functions is finished.
– *fparameters* function*:* The lazy evaluation can decide when to solve a parameter, this is that if you need it, it will be evaluated. For this, the *fparameters* function creates a stack, which stores in its first column a copy of the symbol table of the function being invoked, in the second column pointers to AST where the unresolved parameters are stored.

The above two functions are invoked at the same time whenever there is a call to a function in the source code. In this way joins a call to a function with its unresolved parameters. Once a recursive function is evaluated, the line in both stacks is removed and freed from memory.

– *interpret_parameters* function*:* The moment that a function requires a parameter to solve their instructions, *interpret_parameters* function is responsible for searching in the parameter table the AST code that solves it. In this way, the function makes necessary operations to solve a function, arithmetic expression, or stored variable, in order to return its resolved value and enable the function that needs the value to continue with its operations.

When solving a parameter, the parameter table is updated with the value obtained, avoiding solve each time the parameter required.

Functions for calculating real numbers are implemented, which use the functions described above for *lazy* evaluation and recursion. The interpreter has predefined functions where the LRT language is integrated.

- *evaluate* function*:* This function is predefined by the interpreter. Receives two parameters to calculate the actual number and the desired accuracy. The function *evaluate*, work as any function in the interpreter in a *lazy* way.
- *evaluate_ cons* function*:* This function receives an AST pointer, the received node has the return instruction from a `Cons`. Is the main function of LRT, because performs the reduction of two `Cons` and evaluate if has come to the accuracy desired by the user. This function stores a list of results, which is the actual number determined at the end of the program.
- *fcons* function*:* The reduction of two intervals is performed by this function. Receives two intervals of type `Cons` and returns the reduction. Since operations are performed between rational separately (numerator, denominator), the integers in the numerator and denominator tend to grow quickly, so a factorization function that factorize the rational numbers to store smaller values within the INTERVAL structure is implemented.
- *frtest* function*:* This function receives an interval where are stored the delimited numbers l and r and an interval. The function returns an integer that indicates whether the input range is or not, between the delimiting content.
- *ftail* function*:* The opposite process of determining the reduction of two intervals, is made by the *ftail* function, this function receives an interval of type `Tail`and returns an interval of type `Cons`, this as a result of the operational semantics rules of LRT.
- *evaluate_ intervals* function*:* Because *ftail* function returns an interval of type `Cons`, an evaluation function is implemented which together with the above, determine the reduction of two intervals using the rules of the operational semantics of LRT, recursively calling the *ftail* function . This function receives a pointer of INTERVAL type, which contains the first position of the doubly linked list of generated intervals.
- *split_ head* function*:* This function receives a pointer to a node in the AST and returns an interval generated type `Cons`. The primary function of *split_ - head* is to obtain an interval type `Cons`. This function is called whenever it is necessary to calculate an other interval in different functions of the LRT language.
- *asignParamAst* function*:* For proper operation and code and memory optimization, a specific function is implemented to store the resulting values in the calculation of real numbers. The *asignParamAST* function receives a pointer to an AST node, and returns a pointer to a node of the same type. Its primary function is to store the resulting intervals in the parameter stack to prevent calculate again.

3.5 Optimize Code

The handling of pointers along the implementation of the interpreter is essential for fast and efficient memory usage. By not having to store each value in memory space, we opt for using pointers to reference calculated values, resolve the *lazy* evaluation and recursion over interpretation. Similarly, for optimal usage, calls and parameter stacks are built in such a way that when a recursive call is finished memory is released. One of the main advantages of the interpreter, despite using infinite lists scheme, the list that stores the result does not grow more than two elements, unlike other implementations that store a list of intervals calculated as the final result.

4 Results

The tests of the interpreter are performed on a computer with a processor at 2.66 GHz Intel Core 2 Duo, 4GB of DDR3 RAM. For testing and validation of the interpreter ten different scripts written in LGR are executed. The results of the performance are shown below.

4.1 Calculation of Real Numbers with LGR

The *faverage* function receives two real numbers and returns the average of them. Ten different scripts are written and ran to calculate the arithmetic expressions of the first column of table 1, in these scripts *faverage* and *fractolist* functions are implemented in LGR, the last one converts a rational to a intervals list to do later basic operation of average. The comparative with the `GMP-FC++` library for LRT is shown in Table 1.

The results in the Table 1, show that the runtime is lower than Lucatero's, Likewise the memory consumption is minimal compared to her implementation; Regarding the arithmetic expressions 9 and 10, interpreter results surpass those of `FC++` due to inefficient factorization function used by the interpreter, showing that the implementation of the program takes due to the number of operations performed to factor and not the overflow of RAM memory. In cases where takes longer time is only consuming $348K$ of available memory.

The development of this work, follows a standard methodology for creating compilers and interpreters. This allows the LRT language perform more autonomously and not depend on functional languages like Haskell or as GMP library functions for handling infinite lists. Since, the interpreter is written in the C programming language, is possible to run on different operating systems and computer architectures. One problematic factor found in the development of this work is the handling integers of variable lenght due to buffer overflow in the *long long* data type that handles the C programming language; because of that, the interpreter fails to finish on time because rational numbers are variably growing, causing instability in the interpreter. To partly cope with this situation a factoring algorithm that reduces the rational number is implemented to store it appropriately in the type of data set; considering that an algorithm for prime factorization is located within a problem of type NP, their performance at runtime effects the output of the interpreter making it particularly slow. Because of the difficulty of this problem we propose some guidelines in future work section to remedy the situation. Making a comparison with the library developed by Lucatero shows that the memory usage for recursion and efficiency of *lazy* evaluation solves the problem encountered in the previous research because it

Table 1. Comparison between implementations

No	Arithmetic Expression	Runtime (seconds)				
		GMP-FC++	**LGR + factorization**	**LGR without factorization**	RAM used by `GMP-FC++`(K)	**RAM used by LGR (K)**
1	$\sum_{1}^{5} \frac{(\frac{1}{4}+\frac{1}{4})}{2}$	0.342	0.000693	0.000659	740	328
2	$\sum_{1}^{5} \frac{(\frac{1}{3}+\frac{1}{3})}{2}$	0.342	0.000923	0.000885	796	324
3	$\sum_{1}^{10} \frac{(\frac{1}{2}+\frac{1}{2})}{2}$	0.685	0.001929	0.000899	820	344
4	$\sum_{1}^{10} \frac{(\frac{2}{5}+\frac{2}{5})}{2}$	0.685	0.001234	0.000813	$1,020$	340
5	$\sum_{1}^{15} \frac{(\frac{1}{3}+\frac{1}{3})}{2}$	1.028	0.009086	0.008132	$1,252$	332
6	$\sum_{1}^{15} \frac{(\frac{1}{4}+\frac{1}{4})}{2}$	1.028	0.009270	0.001066	904	356
7	$\sum_{1}^{20} \frac{(\frac{1}{2}+\frac{1}{2})}{2}$	1.41	0.135286	0.001587	984	372
8	$\sum_{1}^{20} \frac{(\frac{2}{5}+\frac{2}{5})}{2}$	1.41	0.406574	0.005135	$1,544$	372
9	$\sum_{1}^{25} \frac{(\frac{1}{4}+\frac{1}{4})}{2}$	1.7625	8.527817	0.011391	$1,076$	384
10	$\sum_{1}^{25} \frac{(\frac{2}{5}+\frac{2}{5})}{2}$	1.7625	12.953400	0.025865	$1,868$	380

could not control how memory is freed in `FC++` runtime, unlike this interpreter that performs releasing memory when finished using a recursive call or parameter. Making a comparative in runtime this interpreter surpass the reported by Lucatero, due to handling infinite lists.

5 Conclusions

In this paper an interpreter using the methodology shown in Section 3 is developed, for calculating exact real numbers in the programming language C. The functions necessary to implement recursion, *lazy* evaluation and infinite lists are implemented, using Flex tools for building a lexical analyzer and Bison for the construction of a parser. The operational semantics of the programming language LRT and `C-` language are merged in order to obtain the language LGR. The results show the efficiency of the management of recursion and *lazy* evaluation. In a matter of precision and accuracy of the results of operations that can be performed with the interpreter, the results show that the implementation provides an accurate result using the specified accuracy. The development of the interpreter shows evidence of having a standalone product and controlled for LRT language .

The problem of memory management in the library created by Lucatero has been resolved in this implementation, however, problems encountered in this development are the following:

An interval consists of two rational, which are treated separately as the numerator and denominator, performing operations using these integers. The problem arises when these integers start growing. Integers use the type *long long*, the larger type that stores integers in C. Since the numbers are growing rapidly, a factorization function is used to decrease the size of the resulting integer. When making an interval reduction by *fcons* function, function *factorize* is invoked. When trying to store these numbers in variables of type *long long*, at reach its maximum storage capacity, overflow variables, avoiding the interpreter to finish its execution.

The development of this interpreter leaves open lines for improvement in a matter of implementation, to solve the problems encountered in this development and add more grammar to the language to make a more robust LGR language. A further work may be the following:

– Development of a data structure capable of storing a variable length integer.
– Implementation of the trigonometric functions for LRT.
– Perform the implementation of three-address code.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques. Addison wesley (1986)
2. Corbett, R., Stallman, R.: Bison: Gnu parser generator. Texinfo documentation, Free Software Foundation, Cambridge, Mass (1991)
3. Edalat, A.: Exact real number computation using linear fractional transformations. `http://www.doc.ic.ac.uk/exact-computation/exactarithmeticfinal.ps.gz`
4. Errington, L., Heckmann, R.: Using the ic reals library. `http://www.doc.ic.ac.uk/exact-computation`
5. Escardó, M.H.: PCF extended with real numbers. A domain-theoretic approach to higher-order exact real number computation. Ph.D. thesis (1997)
6. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR) 23(1), 5–48 (1991)
7. Lambov, B.: Reallib: An efficient implementation of exact real arithmetic. Mathematical Structures in Computer Science 17(01), 81–98 (2007)
8. Linares, D.: Diseño e implementación de una calculadora científica con el lenguaje de programación lrt. Licentiate thesis (2009)
9. Louden, K.C.: Construcción de compiladores. Principios y prácticas (2005)
10. Lucatero, A.: Desarrollo e implementación de una librería para el cómputo real exacto basado en lrt. Licentiate thesis (2013)
11. Marcial-Romero, J.R., Escardó, M.H.: Semantics of a sequential language for exact real-number computation. In: Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on. pp. 426–435 (2004)

12. Müller, N.T.: The irram: Exact arithmetic in c++. In: Blanck et al. (eds.). LNCS, vol. 2064, pp. 222–252. Springer Verlag Heidelberg (2001)
13. Paxson, V., Estes, W., Millaway, J.: Lexical analysis with flex. `http://flex.sourceforge.net/`
14. Plume, D.: A calculator for exact real number computation. Ph.D. thesis (1998)
15. Potts, P.J., Edalat, A.: Exact real computer arithmetic. Draft report, Imperial College, London (1977)
16. Villanueva, G.: Diseño e implementación de una calculadora para cómputo con números reales exactos empleando información redundante. Licentiate thesis (2007)