Functional Programming and Interval Arithmetic

R. D. Lins, M. A. Campos, and M. de B. Correia

***Abstract*— The functional programming paradigm makes it easier to prove correctness of programs. However, if a program uses operations with real numbers, then, even if this program is correct, the use standard (not absolutely precise) computer operations with floating point numbers can make the result of this program incorrect.**

In this paper, we describe how to add interval arithmetic to functional programming. This addition enables us to prove correctness of the programs that process real numbers. Our goal is to design programs that are reliable and efficient.

I. Introduction

Each day, the demand for reliable software increases. Nowadays, software already controls nuclear plants, weapons, etc. Human lives, and even the existence of the whole planet depend (to some degree) on the reliability of these programs.

One way to guarantee reliability is to test a program for all possible cases. Usually, there are many possible inputs, so testing all of them is always difficult, and most of the time practically impossible. If we test a program on only some possible inputs, then the results of this testing are not 100% reliable. The only way to guarantee the program's reliability is to *prove* the program to be correct. The existing software design techniques, specification methods, programming style, languages, and system environments are not very supportive of the correctness proofs. To make them supportive, a long evolution is needed. Also, we need sophisticated tools to help us with proving program correctness.

In his 1978 Turing Award Lecture [4], John Backus drew the attention of the computer science community to *functional programming* as a possible solution to a series of problems of programming known as "the software crisis". Following this idea, the 1980's saw large advances in formal methods and functional programming. However, the low performance figures offered by all implementations of functional languages largely compromised their applicability. Only recently, implementors of functional languages have learned how to produce fast functional compilers [10, 11]. These compilers renewed the interest in functional programming.

Functional languages are, in essence, a nicer syntax to the λ-Calculus, a theory of functions [8, 3]. This makes proving program correctness a much easier task than in *imperative programming*, a more traditional programming paradigm in which programs reflect the features of the machine architecture.

Application programs that have been formally proved correct can only be reliable if they run on hardware that has been formally proved correct, under correct basic software, such as operating systems, environments and tools. This philosophy is inconsistent with the use of floating point arithmetic, because floating point arithmetic is inherently imprecise.

One of the features of "pure" functional languages, such as Miranda[1] [16], is *lazy* evaluation. This means that any expression will be evaluated only if needed, and at most once. Lazy evaluation allows the use of infinite data structures, such as infinite sets and lists, which are described by a formation law. These structures can be seen as intensionally defined.

The functional programming community has been aware of the fact that floating point arithmetic introduces errors to programs and thus, destroys their reliability. Lazy evaluation has been used to implement infinite precision arithmetic [6, 5]. Using this technique, real numbers are internally represented by a lazy list, and for this reason they are called *lazy reals*. In order to reduce the propagation of carries, Avizienis' numerical system is used [2], at a cost of losing uniqueness in representation. Operations on these numbers are also lazy. The whole evaluation process is driven by the printing routine, which "unfolds" the definition of the data structures generating a precision as good as required by the

The authors are with Dept. de Informática, Universidade Federal de Pernambuco, Recife, Brazil. Research reported herein has been sponsored by CNPq (Brazil) grant 530465/93-0.

[1] Miranda is a trademark of Research Software Ltd.

application. Unfortunately, this process involves far too much symbolic manipulation and is unbearably slow. Lazy reals also have an additional drawback: they cannot be described by the existing (ordered and metric) approaches to formalizing continuous data types [14].

We see interval arithmetic [1] as an alternative to infinite precision arithmetic, an alternative that can offer a much lower computational overhead.

In this paper, we briefly describe the main features of functional languages and advocate the idea that the combination of interval arithmetic and functional programming can make programs reliable and efficient.

II. Functional Programming

In this section we present a brief overview of the main features of lazy functional languages.

Three main characteristics define a "pure" functional language, such as Miranda [16], are:

- *higher-order functions*: functions are not only passed as parameters to other functions, but can also be the result of the evaluation of a given function.
- *referential transparency*: the value of any subexpression is static, there is no destructive assignment.
- *lazy evaluation*: any expression is evaluated at most once, and only when necessary.

Below we outline some aspects of the functional programming style.

A. The evaluation mechanism

Functional programs consist of definitions of functions and other objects. The execution of a program in a functional language consists of the evaluation of an expression; this evaluation can be performed as a successive rewriting of an expression until it takes a printable (*normal*) form (a normal form is the one that cannot be rewritten further.) For example, if we say

```
fac n = n * fac(n-1)   , n>0       (1)
      = 1              , otherwise
```

then

```
fac 2
```

is rewritten thus

$$\begin{aligned} \texttt{fac 2} &\Rightarrow 2*\texttt{fac}\ (2-1) \qquad (2)\\ &\Rightarrow 2*\texttt{fac}\ 1\\ &\Rightarrow 2*(1*\texttt{fac}\ (1-1))\\ &\Rightarrow 2*(1*\texttt{fac}\ 0)\\ &\Rightarrow 2*(1*1)\\ &\Rightarrow 2*1\\ &\Rightarrow 2 \end{aligned}$$

The definition (1) is used in the first line of (2) where we have to substitute the actual value 2 for the variable n; this process of parameter passing forms the major overhead in rewriting implementations of functional languages.

B. Pattern-matching

Pattern-matching is another interesting simplification to the syntax of expressions in functional languages. Using pattern-matching, one can place guards on the left hand side of a function definition. For instance, using pattern-matching, the above function definition can be expressed more neatly as

```
fac 0 = 1
fac n = n * fac(n-1)
```

In the general case pattern-matching can become quite complex. Miranda allows pattern-matching on general algebraic data types. In these cases, pattern-matching not only provides us with guards (as in the above example), but also with *selectors* which can "decompose" objects of composite (algebraic) data types.

C. Introducing new data types

The primitive, or *basic* types in Miranda are: `num`, `char`, and `bool`. The type `num` comprises both integers and floating point numbers. As elements of the type `bool`, we have the truth values `True` and `False`. The type `char` contains the characters of the ASCII character set. Miranda also contains data type constructors such as `list` and `tuple`.

A new algebraic type in Miranda can be is introduced by using the symbol `::=`. For instance, the type `tree` can be introduced as follows:

```
tree ::=  Leaf num | Node num tree tree
```

The tree-constructor `Leaf` has a type `num` → `tree`, and `Node` has a type

$$((\texttt{num} \rightarrow \texttt{tree}) \rightarrow \texttt{tree}) \rightarrow \texttt{tree}$$

(the → operator denotes curryfication and is left-associative). In Miranda, the definition mechanism uses pattern matching on constructors. For instance, we can define a function `wt` that computes the weight of a give tree as

```
wt (Leaf n) = n
wt (Node n t1 t2) = (wt t1)+(wt t2)+n
```

The type system of Miranda subsumes Pascal enumerated types, records, variant records, and some uses of pointers which are involved in the construction of recursive dynamic data structures.

D. ZF-expressions

The general form of a ZF-expression in Miranda is

$$\{exp;\ qualifiers\}$$

with an arbitrary (non-zero) number of *qualifiers*. Each qualifier is either a *generator* or a *filter*. A *generator* is a list of values which will be attributed to a specific variable in the expression *exp*. A *filter* is an arbitrary boolean expression that further restricts the range of a generator.

ZF-expressions are a convenient and elegant syntactic 'sugaring' of a class of expressions. For example, the list of all the squares of the integer numbers between 1 and 10 can be expressed in Miranda by the following ZF-expression:

```
{ x*x; x <- (1..10)}
```

E. Local definitions

The general form of a definition will be:

$$\begin{array}{lll} f\ x_0 \ldots x_n = & e_0 & \texttt{where} \\ & & a^1 v_1^1 \ldots v_p^1 = d^1 \\ & & \vdots \\ & & a^m v_1^m \ldots v_q^m = d^m \end{array}$$

The definitions of the functions $a^1, \ldots, a^m$ within the `where` block can be mutually recursive. Their scope is restricted to the expression e_0 (and to the right-hand sides of their definitions, since they are recursive). Local definitions are used both to define auxiliary functions and to hold values of intermediate computations which may be referenced a number of times in the expression e_0.

The definitions of $a^1, \ldots, a^m$ may themselves contain local definitions; a local definition is used even if a more global definition of the same symbol exists.

III. Programming language support for scientific computation - Prerequisites

In this section (based on [9, 15]), we present the features that a programming language should have to be suitable for numerical computation with the type `interval`. It is necessary to know these features if we want to enrich a functional language with this data type.

The data types occuring in numerical computation include real and complex numbers, vectors, and matrices.

Mathematicians use the same infix operators to denote the arithmetic operations is all corresponding spaces. Also identical function names identify the standard functions for different data types. Matrix or vector algorithms are usually formulated for arbitrary dimension.

Furthermore, the accuracy of the basic operations shall be optimal for all the spaces of the numerical computations.

Programming languages should enable the programmer to write and implement new arithmetic operations or even new expression evaluation in the language. For this purpose the following concepts are recommended:

- user-defined operators; at least, by overloading the standard operators, and ideally, by allowing new operator identifiers and/or symbols;
- operators and functions whose results are of composite (structured) data types;
- overloadable function identifiers;
- dynamic arrays;
- access to directed roundings for interval arithmetic;
- for the accuracy requirement three arithmetic properties are mandatory:
  - floating-point arithmetic must be optimally defined;
  - at least three roundings must be accessible;
  - an optimal scalar product must be available.

The following additional features are intended to make programming easier:

- A module concept is very helpful to facilitate the management of a lot of different arithmetic libraries.
- Generic, type-parameterized subroutines reduce the size of source code and thus enhance the readability of the program by combining routines which perform identical operations for different data types. The object code size may also decrease as a result.
- A subtype concept allows the automatic conversion of parameters to match the interface between the expressions of different types.
- Alternative expression concepts (e.g., for accurate evaluation or symbolic computation) may be supplied.
- Simultaneous polymorphism increases the information obtained:
  - a complex interval may be simultaneously treated as a circle and as a rectangle;
  - an expression can be both kept in its symbolic form, and evaluated as floating-point number and interval.

IV. Functional Programming for Scientific Computation

Analyzing the list of desirable features a language must have to support numerical computations one can see that many of the features above that will make programming easier either have already been implemented in functional languages (such as modules, polymorphism, etc.) or can be easily included in them.

Two major problems must be overcome in order to make functional languages suitable for scientific computation:

1. The introduction of the type array should not break referential transparency.
2. The inclusion of an explicit iteration operator.

The loss of referential transparency means that the user is interfering directly in the state of the execution of programs, which, in its turn, means not only losses to the potential parallelism, but also weakens the possibility of formal reasoning about programs. A neat solution has been recently proposed with *monads* [13].

The second problem is a much deeper one. Recursion is the natural way functional programmers express the repetition of parts of code, while iteration plays the same rôle in expressing numerical algorithms. In our experience, the translation of a iterative numerical algorithm into a recursive version is a *tour de force* and yields unnatural results.

In our implementation of ΓCMC with the primitive type interval [12], we are analyzing the possibility of including an explicit iteration operator, as well as implementing the following features, inspired by the PASCAL-XSC compiler [7]:

- The concept of a universal operator.
- Functions and operators with results of arbitrary data types.
- Overloading of procedures, functions and operators.
- The concept of a module.
- Dynamic arrays.
- Access to subarrays.
- The concept of a string.
- Controlled rounding.
- Optimal (exact) scalar product.
- Standard type dot precision.
- Additional arithmetic standard data types such as complex, interval, etc.
- Highly accurate arithmetic for standard data types.
- Highly accurate standard functions.
- Exact evaluation of expressions.

V. Conclusions

In this paper, we advocate the idea that the combination of interval arithmetic and functional programming can make programs reliable and efficient. Programs obtained by these means have the quality of a theorem in the sense of pure mathematics.

Many of the features that are desirable for the efficient scientific programming are already present in the implementation of modern functional programming languages. However, in order to make numerical algorithms easier to program in pure functional languages, some compromise is needed. This compromise must not damage the reliabilities of the functional programming paradigm and of interval arithmetic.

References

[1] G. Alefeld and J. Herzberger, **Introduction to Interval Computations.** Academic Press, N.Y., 1983.

[2] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic", *IRE Transactions on Electronic Computers*, 1961.

[3] H. P. Barendregt, **The Lambda Calculus, its Syntax and Semantics**, Studies in Logic and the Foundation of Mathematics, 2nd ed., North-Holland, 1984.

[4] J. Backus. "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Communications ACM*, 1978, Vol. 21, No. 8, pp. 613–641.

[5] H. Boehm and R. Cartwright, **Exact real arithmetic: Formulating real numbers as functions**, Technical report Rice COMP TR88-66, Department of Computer Science, Rice University, April 1988.

[6] H. Boehm, R. Cartwright, M. J.O'Donnell, and M. Riggle, "Exact real arithmetic: A case study in higher order programming", in *Proceedings of the Lisp and Functional Programming Conference*, 1986, pp. 162–173.

[7] R. Hammer, M. Neaga, and D. Ratz, "Pascal-XSC, New Concepts for Scientific Computation and Numerical Data Processing", In: R. S. Stepleman (ed.), **Scientific Computing**, North-Holland, 1993, pp. 15–44.

[8] A.Church, "A set of postulates for the foundation of logic", *Annals of Math.*, Vol. 33, No. 2, pp. 346–366.

[9] J. W. von Gudenberg. "Programming Language Support for Scientific Computation", *Interval Computations*, 1992, No. 4(6), pp. 116–126.

[10] R. D. Lins and B. O. Lira, "ΓCMC: A Novel Way of Implementing Functional Languages", *Journal of Programming Languages*, 1993, Vol. 1, pp. 19–39.

[11] R. D. Lins, G. G. Cruz Neto, and R. F. Lima, "Implementing and Optimising ΓCMC", *Proceedings of Euromicro'94*, IEEE Computer Society Press, Sep. 1994, pp. 353–361.

[12] R. D. Lins, M. A. Campos, M. B. Correia, G. G. Cruz Neto, and R. M. F. Lima, "The Implementation of the Type Interval in ΓCMC", *in preparation.*

[13] E. Moggi, "Computational lambda calculus and monads," In: *Logic in Computer Science*, California, IEEE Press, June 1989.

[14] W. R. de Oliveira, and M. B. Smyth, "Quasimetric Σ-Algebras", in *Proceedings of the XIV Congress of the Brazilian Computing Society*, Caxambú, August 1994.

[15] S. M. Rump, **Seminar Notes**, Universidade Federal de Pernambuco, Recife (Brazil), September 1994.

[16] D.A.Turner, "Functional Programming as executable Specifications", *Phil.Transactions of the Royal Society of London*, 1984, Vol. 312, pp. 363–388.