Enhancing imperative exact real arithmetic with functions and logic

Norbert Th. Müller

Abteilung Informatik, FB IV, Universität Trier,
D-54286 Trier, Germany
www.uni-trier.de/~mueller
mueller@uni-trier.de

Abstract. In the near past, there have been many approaches to 'implement' constructive analysis, many of them with the use of functional programmig languages. In this paper we present a similar approach based on the iRRAM-package which has been acknowledged as a fast package for exact real arithmetic in C++.

1 Introduction

Choosing the 'right' programming language for real arithmetic is not easy: There are many arguments that speak for functional languages here, as a real number x can easily be represented by sequences $\Phi : \mathbb{N} \to \mathbb{Q}$, and as functions $f : \mathbb{R} \to \mathbb{R}$ then correspond to $\Psi : (\mathbb{N} \to \mathbb{Q}) \to (\mathbb{N} \to \mathbb{Q})$; both structures can be easily expressed in functional languages, see e.g. (Mül95; O'C05; BK08). However, in the past the usual lazy evaluation in functional languages often lead to an abysmal performance of the implementations. Today, performance is still the most important argument against functional approaches. But parallel to improvements on functional programming itself, also more and more experience from the object-oriented approaches is transferred into the functional implementations. Additionally, exact real arithmetic based on functional programming may now internally use imperative programming as well, e.g. in languages like OCAML(BK08).

Object oriented languages like C++ make it easier to exploit all the possibilities of current CPUs, simply because CPUs are working in an imperative way. In the end, the optimal solution might be a library that incorporates the best parts of both progamming schemes. Nevertheless, similar to the many existing programming languages, many flavors of exact real arithmetic might survive. Software packages are always only an offer: Their users might simply prefer imperative approaches over functional ones, or vice-versa.

Furthermore, the imperative approaches are also improving in their expressiveness. In this short paper, we want to address functional aspects that can be used in packages like the iRRAM. We have the strong belief that even for the iRRAM, that has been presented for the first time in 1996, still many further important improvements can be implemented. A big question is where to make the cut between the elegance of functional programming and the efficiency of imperative (or object-oriented) languages in a mixed approach. In our opinion, the cut has to be done at a very high level; the imperative approach should be followed as long as possible.

Our main aim is an implementation that can be used for numerical computations and that is competitive with ordinary arithmetic using double precision numbers without any rigorous error control. So we work even below the level where usual questions from computational complexity are considered. We believe that in this area, where bit complexity is the issue, Turing machines are an appropriate model and differences between thinking in constructive and or in classical logic are not yet important.

The best fitting theoretical background to the implementation is TTE (Wei00). This is reflected in many implementational details: We may use arbitrary representations for $\mathbb{R}$, as long as they are admissible; computability and complexity can be based on Type-2-machines or oracle machines; continuity restrictions can be weakened using multi-valued functions; I/O operations can model infinite computations with Type-2-machines; metric spaces with dense subsets can be used, etc.

Main focus is on the area of low-precision computations, let us say up to about 1000 bits per number. Of course, in the cases where we need higher precision, the implementation should provide this almost seamlessly. A further important aspect is that we should be able to deal with billions of dependant operations. An illustrative example is a naive implementation of the harmonic series (or rather an initial part thereof), where e.g. $\sum_{n \leq 10^9} 1/n$ can be computed (better: approximated) with double precision using a simple loop in far less then 10 seconds today. If, for each of the 10^9 partial sums, we would allocate an own object in an object-oriented language (in order to get arbitrary precision later) and we would not release them again very soon, we would already require a lot more main memory than standard computers provide today: Each object takes about 50 bytes, so we would need around 50GB! Functional approaches might suffer from the same problem, even if they are enhanced with object-oriented parts.

In the following we want to show that (basically) object-oriented implementations are able to provide a lot of the expressiveness of (basically) functional implementations. To do this, we extend the `iRRAM` package by a functional component. For a better understanding of this extension, we briefly present a few internals of the `iRRAM` package that have been added since 2000, the date of the last publication on the development of the `iRRAM` (Mül01). First versions of the functional approach presented here have been developed in 2005; since the end of 2008 most of it is already part of the published version of the iRRAM package.

2 Basic structure of the iRRAM

In order to understand the ideas behind the `iRRAM`, we should have a look at other approaches to exact real arithmetic first.

A very natural view on real numbers is that they are (equivalence classes of) sequences of rational numbers, so computable real *numbers* can simply be implemented as procedures in imperative languages. But the need for the implementation of real *functions*, i.e. for computations with real numbers, implies that we need a datatype for sequences and that we must be able to construct new sequences during a computation. This can be achieved using functional extensions to an imperative language, e.g. like `FC++` used by (Bri06).

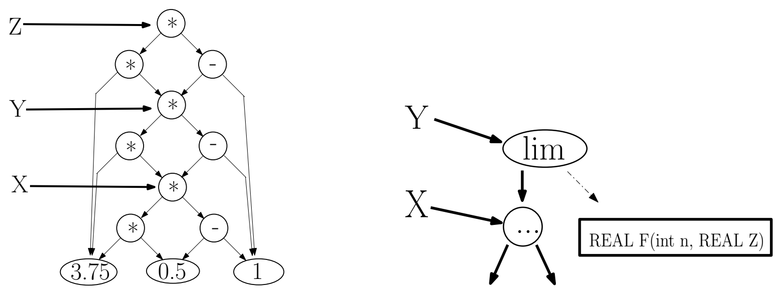


Fig. 1. An 'algebraic' DAG and a DAG with a limit node

Another possibility is to use computation diagrams to describe the computations, see e.g. (Lam07). These essentially are directed acyclic graphs, DAGs, labeled with operator symbols. Suppose e.g. we compute $x_{i+1} = 3.75 \cdot x_i \cdot (1 - x_i)$ iteratively starting with $x_0 = 0.5$. Then three steps `S=0.5; X=3.75*S*(1-S); Y=3.75*X*(1-X); Z=3.75*Y*(1-Y)` of this iteration would lead to the first DAG in figure 2, shared between `X`, `Y` and `Z`.

The construction of such a DAG is a simple programming example in object-oriented languages, but it is already quite memory intensive: Each node in such a DAG will easily occupy about 50 byte. So for current computers, the total number of DAG nodes existing concurrently during a computation must be smaller than about 100 million. This amount of nodes might already be reached if we just want to invert a matrix of dimension 500×500 using Gaussian elimination. The construction of DAGs can roughly be compared to the 'lazy evaluation' used in functional languages; essentially it is just a question of the programming language.

At a first look, DAGs seem to be quite restricted: They can only be constructed from the small number of node types implemented by the programmer, i.e. they have a rather algebraic structure. But special nodes in such a DAG may be used for limits of sequences: Given an algorithm for a sequence F of functions ($F : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$ for simplicity), that converges with known speed to a limit f (e.g. $|f(x) - F(n,x)| \le 2^{-n}$), an assignment like `Y=limit(F,X)` would simply construct the second DAG in figure 2 for $Y = \lim_{n\to\infty} F(n, X)$, i.e. for $Y = f(X)$:

A very important property of DAGs is that they can be 'evaluated', i.e. there are conversion operators to other datatypes (like decimal numbers with finite length). If such a conversion leads to a countable set and is not a trivial constant function, then it has to be either a partial function or, more important, a multivalued function. Usually the conversions will have an additional parameter for the intended accuracy (like the length of the resulting string). It is quite easy to see that with a combination of variably long conversions of reals to decimal strings, (exact) conversions from decimal strings to reals, and the limit operator, we are already able to compute all computable real functions (in the sense of TTE (Wei00)). So the main issue is not the computational power of the implementation, but efficiency and usability.

The efficient evaluation of a DAG is not a trivial task (see e.g. (Lam07)): Suppose we want to get, with a certain accuracy, an approximation d to a number x represented by a given DAG. One approach (A) for this is to try to compute the necessary precisions for the operations top-down on the way to the leaves and then compute values from the bottom up with the computed precisions. Unfortunately, this will very often lead to gross overestimations of the necessary precisions and hence to bad efficiency.

The other possibility (B) is to try an iterative evaluation of the DAG from the leaves to the root: Using interval arithmetic, we start with an initial guess for a suitable precision and eventually get a first approximating interval I containing x. If I is too large for the wanted accuracy, we iteratively repeat the evaluation with higher precisions. The efficiency of this approach is much better than one might expect: If the involved precisions grow exponentially from iteration to iteration, the computational complexity is usually dominated by the last iteration.

Special care has to be devoted to multivalued functions that naturally arise during computations with real numbers: Suppose that in the mid of a computation with real numbers a multivalued function is called. Then there are two possibilities: (I) The returned value might be from a countable set (like e.g. rounding to an integer or computing a decimal approximation). In this uncomplicated case, the value simply might influence the shape of DAGs in the further computation or become a leaf in a later DAG again. In this way, the value is encoded into the DAG. The other possibility (II) is that the returned value is from an uncountable set again (like taking the square root of a complex number or like computing a lower real approximation to a left-computable number): In this case, the multivalued function itself would have to be a node in the DAG. As a consequence, in the iterative evaluation of the DAG, each iteration might compute a different result. This could lead to a failure of the iterative evaluation. As far as we know, this problem has not yet been considered at all for the DAG approach, implementations like `Reallib` (Lam07) only consider single valued functions in the nodes.

After these preparations, the structure of the iRRAM can now be explained very easily: The `iRRAM` package takes the approach (B) to the extreme; its base is iterations on intervals, but the DAGs are not constructed at all! Instead of the DAGs, just enough data is saved to ensure that the whole computation can be redone with a higher precision, if necessary. This immediately cuts memory consumption to a very small amount: It is almost as small as for a program using double precision numbers, just that multiple precision numbers have to be used instead. This efficient way of operation had been present in the `iRRAM` from the very beginning in 1996. The `Realib` in contrast, being solely based on DAGs in the beginning, had to be enhanced with a mode copying the `iRRAM` behavior in order to be applicable to larger sized problems at all.

It is also quite easy to see that the data to be saved from one iterative evaluation to the next simply consists of all the outcomes of the used multivalued functions: So dealing with multivalued functions in case (I) is now a bit more complicated: Different answers on different iterations could now lead to inconsistent results. So when a computation is repeated, the saved outcomes of the multivalued functions from the previous iteration have to be ‘replayed’ (in their original order). This can be done using simple linear lists that need only marginally more space than the values stored in them. So usually even a very frequent use of multivalued functions is handled with substantially less memory in the `iRRAM` than in the DAG approach. For case (II), there exists a special limit operator that has been introduced already in (Mül98).

In the following we briefly address important additions to the `iRRAM` since 2000.

Iterations: The basic concept of the `iRRAM` is the computation in iterations. Whenever the current approximation is unable to continue because of the lack of precision, the computation is repeated. In the version from 2000, this was still done using `longjmp`, but has shortly after been replaced by exception handling. So in the `iRRAM`, exceptions are the rule.

Additionally, the exception handling is able to promote information about the amount of accuracy missing when an iteration is initiated. Using the fact that in applications almost all computed functions are locally Lipschitz-continuous, a heuristic is used to estimate what precision will be appropriate in the next iteration. This can often reduce the number of iterations to just two: The first, initial one that implicitly computes a Lipschitz constant, and then the second one that gets the sufficiently precise result.

Input and Output: While in the older versions input and output used `C`-style (e.g. `printf`), this has been changed to own versions of `cin`, `cout`, `cerr` and even of file-I/O, that work correctly despite the iterative nature of the `iRRAM`.

Underlying Intervals: The iterative evaluation in the `iRRAM` is based on interval arithmetic. Already in the first `iRRAM` version from 1996 (Mül96), simplified centered intervals I were used: They are stored as a multiple precision floation point number d as center, together with a fixed precision floating point number r as radius, i.e. $I = \{x : d-r \leq x \leq d+r\}$. This is of advantage if the radius does not need to be as large and precise as the center d, which is often the case if we attempt to compute functions on single real numbers. But if d and r need a similar amount of memory, the traditional representation using the endpoints l, u and with $I = \{x : l \leq x \leq u\}$ is competitive. This is true especially, if it is sufficient to use double precision numbers for l and u. So in parallel to an extension in the `Reallib` package (Lam06), in 2004 intervals using double precision numbers have been added also to the `iRRAM`. Operations on these (quite restricted) intervals can be executed very fast, as the corresponding arithmetic is built into the hardware. Their main use is in the initial iteration of `iRRAM` computations. So if a computation is possible without necessity of more precise intervals, then it will be amazingly fast, almost as fast as unverified arithmetic.

At lot of effort has been put into the optimization of this initial iteration:

- All operations have been implemented as `inline` functions, avoiding expensive function calls.
- An object of type `REAL` is basically just a double precision interval plus a pointer to a possible simplified centered interval. This structure is so small that constructing an object of this type during the evaluation of expressions does not need to call functions, especially `malloc` will not be called.
- Interval arithmetic needs a proper outward rounding of the borders: l must be rounded to $-\infty$, while u must be rounded to ∞. As switching the rounding mode for double precision numbers in current CPUs is a quite expensive operation, the right border is inverted: The interval $[l, u]$ is stored as the pair $(l, -u)$, such that the rounding mode does not have to be changed during a computation.

– In contrast to `Reallib`, where switching between double precision intervals and simplified centered intervals is done essentially at compile time via `templates` in `C++`, the `iRRAM` uses a (slightly slower but dynamic) approach with a check whether a pointer is `NULL`. This easily allows a mixture of both types of intervals during one (usually the first) iteration.
– To utilize the CPUs further, the computation with double precision intervals is done with the `SSE2` extensions of (Intel or AMD) CPUs with a 64bit architecture. This allows to compute the left and right border in parallel. Unfortunately, the `gcc` compiler is still erroneous using the `SSE2` extensions: On 32-bit-machines, the data is not always properly assigned to a 16-byte-boundary on the stack, leading to segmentation faults. This is announced to be changed from `GCC` version 4.4 onwards, so soon also on 32-bit-machines the `iRRAM` will benefit from `SSE2`.

As a benchmark, a naive computation of the harmonic sequence is usefull: If we compute $\sum_{i=1}^{n} \frac{1}{i}$ using a loop, i.e. `x=0; for (i=1;i<=n;i++) x=x+REAL(1)/i` (ignoring better ways to compute this value), we get the following results for $n = 10^9$:

– Using the DAG-version of the `Reallib`, already $n = 10^7$ needs about 1.7GB of main memory just to construct the final DAG. So clearly the value for $n = 10^9$ would be impossible to be computed at all.
– Using the `iRRAM`, so avoiding DAGs at all, 5 leading decimals of the sum for $n = 10^9$ can be computed within 15s (using `SSE2`) or 18s (without `SSE2`) on a processor of type Athlon 64 X2 4600+. Here the first iteration with its double precision intervals is already sufficient.
– Using the iterative mode of the `Reallib` (that copies many features of the older `iRRAM`, but has the `template` approach to double precision intervals), those 5 decimals of the sum for $n = 10^9$ can be computed within 12.5s (without using `SSE2`). So the template approach is faster in this special case. Unfortunately, we were unable to compile the `Reallib` to use `SSE2`.
  We want to emphasize here that we knew that the dynamic interval approach of the iRRAM (where each operation has to test for the type of intervals used) must sometimes be slower than the `template` approach of the `Reallib`, where the type of intervals is essentially fixed at compile time. It has been a design decision to use the dynamic approach, as it easily allows to mix low precision computations with parts that need higher precision.
– Computing 20 decimals of the sum for $n = 10^9$ takes between 300s (`iRRAM`) and 550s (`Reallib`). Obviously, double precision doesn't suffice here. `iRRAM` uses a second iteration with simplified centered intervals and a precision of 2^{-136}.
– As a comparison: Using simple double precision numbers (i.e. no intervals and no error control at all), $n = 10^9$ takes about 7.5s, delivering 10 correct decimals. So for low precision results, the `iRRAM` is slower only by a factor of 2.

Lazy Booleans: As all computable real functions have to be continuous, comparsion operators like < or <= have to be undefined if their two arguments are equal and if their values are just allowed to be $\{T, F\}$. This sometimes makes it quite hard to translate ordinary programs to the `iRRAM`. The datatype `LAZY_BOOLEAN`, introduced in 2003 and based on a similar datatype in (EH02), is an extension of the usual boolean data type, consisting of the three values $\{T, F, \perp\}$. One possible view is that these values form a DCPO, where the logical operators have non-strict extensions:

a||b	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

,

a&&b	T	F	$\perp$
T	T	F	$\perp$
F	F	F	F
$\perp$	$\perp$	F	$\perp$

,

!a	
T	F
F	T
$\perp$	$\perp$

The comparison operators can be easily extended continuously to this DCPO. In the `iRRAM`, where we work with interval approximations to the real numbers, we are able to compute the values of such a comparison as soon as the intervals are disjoint. If they are not disjoint, the computation may continue with the 'approximation' $\perp$ until we need a 'higher precision', for example in a conversion to the ordinary type `bool`.

$$\texttt{(x<y)} = \begin{cases} T, \ x < y \\ F, \ x > y \\ \perp, x = y \end{cases}$$

b	bool(b)
T	T
F	F
$\perp$	undefined

Conversion to `bool` will mainly happen implicitly: E.g., '`if ( a<b && c>d ) {...}`' will use `LAZY_BOOLEAN`s for the three operations <, > and `&&`, but a final conversion to `bool` for the `if`.

Trying to convert $\perp$ to a boolean value will force a new iteration of the `iRRAM`, leading to smaller intervals that might allow to actually determine the right truth value.

The purpose of the `LAZY_BOOLEAN`s is to avoid discontinuities: Expressions like '`x<1 || x>0`' will now be total (although this example is trivial), whereas ordinary boolean evaluation would lead to at least one undefined value (at $x = 1$ or $x = 0$). Non-trivial examples can be constructed with the special n-ary (and multivalued) function `choose(b_1,...b_n)` that returns the index of one of the parameters with a true boolean value.

3 Function objects

Functional approaches in `C++` have already been used for exact real arithmetic for about ten years, e.g. by Briggs (Bri06) using the functional extension `FC++`. Strangely, the functional approach stops at the level where real numbers are defined, it does not consider real *functions* as objects. Even without the use of `FC++`, we may use a combination of templates, computation diagrams, and overloading of the `()` operator to get a reasonable interface for real functions in `C++`.

For the implementation of functions, we could also have used representations like those defined in (Wei00): We could e.g. use a metric on functions spaces, define some dense subset, and then implement a function as the limit of a sequence of approximations from this dense subset. (This is the core of the implementation of the real *numbers* in the `iRRAM`.) Unfortunately, these approximations would require too much memory.

Instead, we took an approach similar to the DAGs used for real numbers. The use of DAGs avoids the necessity to construct approximations to functions explicitly, we only have to copy pointers and manage simple reference counters. In contrast to numbers, the main evaluation procedure for these function-DAGs will not aim at 'full' approximations to the functions, but usually only give functional values at points, so we avoid to store the expensive approximations to the functions. Of course, the limitations from the previous section still apply here: The size of the DAGs is limited to a few million nodes that exist concurrently.

We use a two step construction to the function-DAGs: The basic class uses the concept of `templates` in `C++` to define a generic function type:

```
template<class PARAM,class RESULT> class FUNCTION;
```

The objects of this class are functions from a `PARAM` data type to a `RESULT` data type. Using overloading of the `()` operator, a `FUNCTION<PARAM,RESULT> f` can almost be used like an ordinary function of type `RESULT g(const PARAM& x)`, i.e. we may use expressions like `y=f(x)` for the evaluation of a `FUNCTION`. The implementation essentially uses a single pointer to the following class:

```
template<class PARAM,class RESULT> class FUNCTIONAL_object
```

The objects of `FUNCTIONAL_object<PARAM,RESULT>` now are nodes in DAGs. But in contrast to the previous sections, these DAGs represent functions and not numbers. As tools for these function-DAGs we implemented an initial subset of the operators given in (Bra98). The selection of the currently implemented operators is neither minimal nor meant to be complete, further operators can be easily added at need.

General constructors: The most general constructor just transforms an implemented algorithm into a function with the corresponding types. Additionally, there is a constructor for constant functions with a given value:

```
FUNCTION<PARAM,RESULT> from_algorithm (RESULT (evalp)(const PARAM &z))
FUNCTION<PARAM,RESULT> from_value     (const RESULT &value)
```

As an example, we might use an implementation of the trigonometric function sin to define a corresponding `FUNCTION`:

```
REAL sin (const REAL& x) { ... }
FUNCTION<REAL,REAL> _sin_ = from_algorithm(sin);
...
REAL x  = ...
REAL fx = _sin_(x);
```

An unexpected restriction with the naming of such functions that arises in `linux` systems should be mentioned here, as it slightly influences the use of the `iRRAM` library: Internally, the `iRRAM` uses routines from `cmath` for a few very fast operations on double precision numbers. Unfortunately, using `cmath` leads to a pollution of the global namespace: Many usefull names as `sqrt`, `abs`, or `sin`, can no longer be used for function objects. In consequence, users of the library will be unable to compile applications if they try to use such a common name for their own functions.

Structural operators: The following operators build the basis for the construction of larger function-DAGs:

– Given two functions $f : \subseteq M_1 \to M_2$, $g : \subseteq M_2 \to M_3$, then $h = compose(g, f)$ is the function $h : \subseteq M_1 \to M_3$ with $h(x) = g(f(x))$.
– Given two functions $f_1 : \subseteq M_1 \to M_1'$, $f_2 : \subseteq M_2 \to M_2'$, then the product h of f_1 and f_2 is the function $h : \subseteq M_1 \times M_2 \to M_1' \times M_2'$ with $h(x_1, x_2) = (f_1(x_1), f_2(x_2))$.
– Given two functions $f_1 : \subseteq M \to M_1$, $f_2 : \subseteq M \to M_2$, then the juxtaposition h of f_1 and f_2 is the function $h : \subseteq M \to M_1' \times M_2'$ with $h(x) = (f_1(x), f_2(x))$.

In C++, the prototypes look as follows:

```
FUNCTION<PARAM,RESULT> compose
 (const FUNCTION<INTERMED,RESULT>&g, const FUNCTION<PARAM,INTERMED> &f)
FUNCTION<std::pair<PARAM1,PARAM2>,std::pair<RESULT1,RESULT2> > product
 (const FUNCTION<PARAM1,RESULT1> &f_1, const FUNCTION<PARAM2,RESULT2> &f_2)
FUNCTION<PARAM,std::pair<RESULT1,RESULT2> > juxtaposition
 (const FUNCTION<PARAM,RESULT1> &f_1,  const FUNCTION<PARAM,RESULT2> &f_2)
```

Here, the compostion operator can even be written as simply as `g(f)`, i.e. as an application of the outer function `g` on the inner function `f`.

Projections: Although our interface provides only unary functions, the use of functions with several variables can easily be achieved using `pair`s or `vector`s offered by the standard C++ libraries. The numerical operations that we intend to use are mainly binary, so we implemented two projections on `pair`s and a general projection on the `i`th component of a `vector`:

```
FUNCTION<PARAM,RESULT1 > first
 (const FUNCTION<PARAM,std::pair<RESULT1,RESULT2> > &f)
FUNCTION<PARAM,RESULT2 > second
 (const FUNCTION<PARAM,std::pair<RESULT1,RESULT2> > &f)
FUNCTION<PARAM,RESULT> projection
 (const FUNCTION<PARAM,std::vector<RESULT> > &f, const int i)
```

Partial Application: The following operators allow to bind parameters to values:

```
FUNCTION<PARAM2,RESULT> bind_first
 (const FUNCTION<std::pair<PARAM1,PARAM2>,RESULT > &f, const PARAM1& z)
FUNCTION<PARAM1,RESULT> bind_second
 (const FUNCTION<std::pair<PARAM1,PARAM2>,RESULT > &f, const PARAM2& z)
```

Special arithmetic operators: If arithmetic operators can be applied to the `RESULT` datatype, then we may use operators like the following to define new functions:

```
FUNCTION<PARAM,RESULT> operator +
 (const FUNCTION<PARAM,RESULT> &f, const FUNCTION<PARAM,RESULT> &g)
```

For example, given two functions $f : \subseteq M_1 \to M_2$, $g : \subseteq M_1 \to M_2$, then $h = f+g$ is the function $h : \subseteq M_1 \to M_2$ with $h(x) = f(x) + g(x)$. In the implementation, this is essentially mainly the juxtaposition of f and g, where after an evaluation we add the components of the result.

Similar operators exist for the other basic arithmetic operations '$-$', '$\star$', '/' as well.

Special arithmetic constructors: Using the arithmetic operations together with constant functions, we are obviously already able to construct (multivariate) polynomials. Nevertheless, it is more efficient to have a direct access e.g. to polynomials.

```
FUNCTION<PARAM,PARAM> polynomial
  (  const std::vector<PARAM> &coeff  )
```

In a similar way, we might easily use sequences of coefficients in order to define Taylor series.

Comparison operators: For the case that the `RESULT` type has comparison operators, we also implemented a template for componentwise comparison of a function with another function or with a constant value.

For example, given two functions $f : \subseteq M_1 \to M_2$, $g : \subseteq M_1 \to M_2$, then $h = f < g$ is the function $h : \subseteq M \to \{T, F, \perp\}$ with $h(x) =$ '$f(x) < g(x)$', so for $M_2 = \mathbb{R}$ we get

$$h(x) = \begin{cases} T, f(x) < g(x) \\ F, f(x) > g(x) \\ \perp, f(x) = g(x) \end{cases}$$

```
FUNCTION<PARAM,LAZY_BOOLEAN> operator <
 (const FUNCTION<PARAM,RESULT> &f, const FUNCTION<PARAM,RESULT> &g)
FUNCTION<PARAM,LAZY_BOOLEAN> operator <
 (const FUNCTION<PARAM,RESULT> &f,  const RESULT &b)
FUNCTION<PARAM,LAZY_BOOLEAN> operator <
 (const RESULT &a,  const FUNCTION<PARAM,RESULT> &g)
```

Logical operators: Also the logical operators `||` (disjunction), `&&` (conjunction) and `!` (negation) have been extended to functions:

```
FUNCTION<PARAM,LAZY_BOOLEAN> operator &&
 (const FUNCTION<PARAM,LAZY_BOOLEAN> &a, const FUNCTION<PARAM,LAZY_BOOLEAN> &b)
FUNCTION<PARAM,LAZY_BOOLEAN> operator ||
 (const FUNCTION<PARAM,LAZY_BOOLEAN> &a, const FUNCTION<PARAM,LAZY_BOOLEAN> &b)
FUNCTION<PARAM,LAZY_BOOLEAN> operator !
 (const FUNCTION<PARAM,LAZY_BOOLEAN> &a)
```

Miscellaneous further operators: For the ease of formulation function like $g : x \mapsto f(x) + 3$, we added the possibility to mix functions and constant values (that are interpreted as functions of the correct arity). For example:

```
FUNCTION<PARAM,REAL> operator +
 (const FUNCTION<PARAM,REAL> &f,  const REAL &b)
FUNCTION<PARAM,REAL> operator +
 (const REAL &a,  const FUNCTION<PARAM,REAL> &g)
```

Similar operators exist for the other arithmetic operations '$-$', '$\star$', '/' as well.

4 Application Examples

Simple functions: As an example, the following is the complete source code for the implementation of the logistic iteration function $x_{i+1} = 3.75 \cdot x_i \cdot (1 - x_i)$ using a function-DAG. It shows that the iterative structure of the iRRAM reals can be mixed without any problem with the function-DAGs.

```
#include "iRRAM.h"
using namespace iRRAM;
REAL id_REAL( const REAL& x ){ return x; };
void compute(){
  FUNCTION<REAL,REAL> X = from_algorithm(id_REAL);
  FUNCTION<REAL,REAL> LF = REAL(3.75) * X * ( REAL(1) - X );
  int count;  cin >> count;  REAL x = 0.5;
  for ( int i=1; i<=count; i++ )  x = LF(x);
  cout << x << "\n";
}
```

At the moment, the evaluation of the function-DAGs in the published version of the `iRRAM` is not as efficient as those of the (number-)DAGs in `Reallib`: We still use a recursive traversal through the DAG, so the DAG is essentially rather treated as a tree. In consequence, the evaluation of a DAG might take time exponential in the number of its nodes. Caching strategies that resolve this problem are currently under consideration.

Dedekind cuts: Most often, Cauchy sequences are used as the basis of the real numbers used in exact real arithmetic. As an alternative, the (equivalent) Dedekind cuts are of growing popularity, see e.g. (Bau08): There a real number x is represented by two nonempty and disjoint sets L, U of rational numbers, that additionally are located; i.e. if $q < r$, then $q \in L$ or $r \in U$. In classical logic, we could then write that $(\forall q \in L)(\forall r \in U) q < r$ and define $\sup\{q \in L\} = \inf\{r \in U\} =: x$.

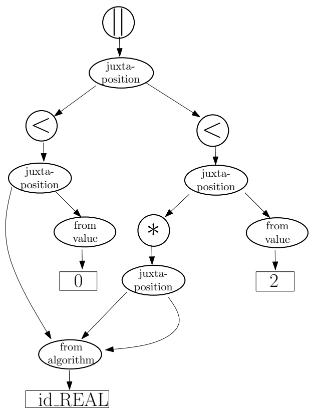
In the following, we want to present how this concept could be integrated into the `iRRAM`. For L and U we used two functions with result type `LAZY_BOOLEAN`, where we chose `REAL` (instead of `RATIONAL`) as argument type. Additionally (and essentially for efficiency), we also added two rational numbers $l \in L$ and $u \in U$ that help to find an initial approximation for the represented real number. So like in (Bau08), $\sqrt{2}$ can be defined by $l = 1, u = 2, L = \{x \mid x < 0 \vee x^2 < 2\}, U = \{x \mid 0 < x \wedge 2 < x^2\}$ Then in the `iRRAM` we could easily implement a corresponding *constructor* for a real number as follows:

```
REAL cut(  const RATIONAL left_bound, right_bound,
           FUNCTION<REAL,LAZY_BOOLEAN> smaller, larger  )
```

As an example, the following program produces the first 100 decimals of $\sqrt{2}$ using two function-DAGs that are constructed on-the-fly.

```
REAL id_REAL( const REAL& x ){ return x; };
void compute(){
   FUNCTION<REAL,REAL>  X=from_algorithm(id_REAL);
   REAL x = cut(1,2, X<REAL(0) || X*X<REAL(2), REAL(0)<X && REAL(2)<X*X);
   cout << setRwidth(100) << x << "\n";
}
```

The function-DAG for `X<REAL(0) || X*X < REAL(2)` is depicted in the following graph:



There are still many rough ends in the implementation The conversion to the usual `iRRAM` reals currently just uses a trisection to find the value. This could be improved by a heuristic to speed up the search, or perhaps even a Newton method.

Additionally, the cuts are implemented using the most simple limit operator of the `iRRAM`. So even after we found suitable good left and right boundaries, after each iteration we have to compute them again. A better way would be to remember the already known values and restart the trisection from there with improved precision. One possibility to do this would be to extend the existing `limit_mv` to remember choices more complicated than 32-bit-ints.

Of course, this will always have higher overhead than the already existing algorithm for the square root, but the cuts form a quite elegant way e.g. to define algebraic numbers. This of course can also be achieved in many other ways. So at the time, the Dedekind cuts in the `iRRAM` are only a proof of concept: Whether it is worth implementing improvements mainly depends on whether there will be applications using Dedekind cuts that can not elegantly be written using the already implemented concepts.

Predicates on real numbers: We may interpret a predicate on the real numbers as a function from vectors of reals to the `LAZY_BOOLEAN` datatype. So our functional extension can also be used as a first step into implemented logic over the real numbers in `C++`. There already exists a vaguely related approach called `LC++` extending `C++` via `FC++` to a `PROLOG`-like language. As an example, we present a few lines showing how expressions and predicates with several variables can be implemented on top of the `iRRAM`:

```
typedef std::vector<REAL> UNIVERSE;
typedef FUNCTION<UNIVERSE,REAL> REAL_EXPRESSION;
typedef FUNCTION<UNIVERSE,LAZY_BOOLEAN> REAL_PREDICATE;

UNIVERSE id_vector_REAL( const UNIVERSE & x ){ return x; };
FUNCTION<UNIVERSE,UNIVERSE> variables=from_algorithm(id_vector_REAL);
```

```
void compute(){
  REAL_EXPRESSION X = projection(variables,0);
  REAL_EXPRESSION Y = projection(variables,1);
  REAL_EXPRESSION Z = X*Y+X/Y;
  REAL_PREDICATE  P =  Z > X  ||  X < REAL(1) ;
  UNIVERSE param(2);
  param[0] = 1; param[1] = 2;
  cout << setRwidth(20) << Z(param) << " "  << P(param)  << "\n";
};
```

This shows that many parts of the approach taken in (Bau08) can already be expressed in object-oriented languages. So for purposes of constructive analysis, a hypothetical approach mixing object-oriented and genuinely functional languages might switch between the paradigms on a quite high level.

5 Final remarks

The user interface to function-DAGs currently allows only the computation of the represented function at a single point. Internally, the interval representation of the iRRAM is used, which is highly optimized for those evaluations. Of course, a deeper look into the implementation would also allow to intercept these underlying intervals and e.g. deduce a modulus of continuity. Additionally we have the special DAG node that allows to use any implemented real function as a leaf in a DAG. So using function-DAGs, viewed as a representation in the sense of (Wei00), is equivalent to the standard representations for functions (on metric spaces). Similar to the number-DAGs, the basic concerns are again efficiency and usability of this special data structure.

The functions-DAGs also enables us to define alternative evaluation strategies, like an interval evaluation. However, a good set-valued evaluation should be preferred. Unfortunately, the representations of sets that can e.g. be found in (Wei00) have not been checked from a practical viewpoint before.

The logical extension to the `iRRAM` is quite basic at the moment, as it does not allow quantification. As soon as there is a good set-valued extension, introducing quantifiers as special logical operators will be quite straightforward: As a basis, the approach taken in (Bau08) can be used.

As far as the author knows, space complexity has hardly been investigated in exact real arithmetic. The enormous reduction using only iterations instead of DAGs (for the real number case) vaguely looks like the difference between solving reachability problems in graphs in logarithmic space or in polynomial time. This could be a starting point for further theoretical research.

Another important question (both in theory and in practice) is how far computations can be parallelized. This is again connected to space complexity, as an efficient parallel evaluation might use temporarily created DAGs to distribute computational tasks.

Bibliography

[Bau08] Andrej Bauer. Efficient computation with dedekind reals. In *CCA*, 2008.

[BK08] Andrej Bauer and Iztok Kavkler. Implementing real numbers with rz. *Electron. Notes Theor. Comput. Sci.*, 202:365–384, 2008.

[Bra98] Vasco Brattka. *Recursive and Computable Operations over Topological Structures*. PhD thesis, Fachbereich Informatik, FernUniversität Hagen, 1998.

[Bri06] Keith Briggs. Implementing exact real arithmetic in c++, python, and c. *Theor. Comput. Sci.*, 351:374–81, 2006.

[EH02] L. Errington and R. Heckmann. Using the IC Reals library. Technical report, Imperial College, 2002.

[Lam06] Branimir Lambov. Interval arithmetic using sse-2. In Peter Hertling, Christoph M. Hoffmann, Wolfram Luther, and Nathalie Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, number 06021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/714> [date of citation: 2006-01-01].

[Lam07] Branimir Lambov. Reallib: An efficient implementation of exact real arithmetic. *Mathematical Structures in Computer Science*, 17(1):81–98, 2007.

[Mül95] Norbert Th. Müller. Constructive aspects of analytic functions. In Ker-I Ko and Klaus Weihrauch, editors, *Computability and Complexity in Analysis*, volume 190 of *Informatik Berichte*, pages 105–114. FernUniversität Hagen, September 1995. CCA Workshop, Hagen, August 19–20, 1995.

[Mül96] Norbert Th. Müller. Towards a real Real RAM: a prototype using C++. In Ker-I Ko, Norbert Müller, and Klaus Weihrauch, editors, *Computability and Complexity in Analysis*, pages 59–66. Universität Trier, 1996. Second CCA Workshop, Trier, August 22–23, 1996.

[Mül98] Norbert Th. Müller. Implementing limits in an interactive RealRAM. In J.-M. Chesneaux, F. Jézéquel, J.-L. Lamotte, and J. Vignes, editors, *Third Real Numbers and Computers Conference*, pages 59–66. Université Pierre et Marie Curie, Paris, 1998. Paris, France, April 27-29, 1998.

[Mül01] Norbert Müller. The iRRAM: Exact Arithmetic in C++. *Lecture notes in computer science*, 2991:222–252, 2001.

[O'C05] Russell O'Connor. Few Digits. 2005.

[Wei00] Klaus Weihrauch. *Computable analysis: an introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.