The iRRAM : Exact Arithmetic in C++

Norbert Th. Müller

Abteilung Informatik — Universität Trier
D-54286 Trier, Germany

Abstract. Abstract: The iRRAM is a very efficient `C++` package for error-free real arithmetic based on the concept of a Real-RAM. Its capabilities range from ordinary arithmetic over trigonometric functions to linear algebra even with sparse matrices. We discuss the concepts and some highlights of the implementation.

Keywords: Computable Real Analysis, Random Access Machines, Limits, Interval Arithmetic, Multi-valued functions, `C++`

1 Introduction

In the last decade, several implementations of exact real arithmetic based on different theoretical approaches or programming languages have been discussed, see e.g. [BoCa90] (essentially based on decimal representation), [EdPo97] (linear fractional transformations), or [GL00] (using multiple precision arithmetic). All these approaches lack the possibility of full imperative programming that would facilitate the implementation of the usual numerical algorithms.

In [BrHe95], Brattka and Hertling considered a different approach: a Turing machine based simulation of random access machines working with reals. The basic idea was to iterate a finite precision approximation of the RAM, but to increase the precision from iteration to iteration.

In [Mu97], the author presented a prototype implementation based on this simulator: an interactive Real-RAM (*iRRAM*). A preliminary version had already been presented at the Second Workshop on Computability and Complexity in Analysis, August, 24/25th 1996, Trier, Germany. Since then, the package has been constantly enhanced and improved. Its stability has been proven by computations taking several weeks of time, producing just a few numbers for a small table in [HM00]. The source of the software package can be found at `http://www.informatik.uni-trier.de/~mueller/`.

A program for the iRRAM is coded in ordinary `C++` but may use a special class `REAL`, that behaves like real numbers without any error. A quite small set of *intrinsic* operations is allowed to be used directly with variables of this type: usual arithmetic operations, tests (with a special semantic defined below), output, and conversion to/from integer or other types. The programmer may use (almost) all programming methods from `C++`, like defining own data types (like real matrices or complex numbers, which are already implemented in this way) and functions (even returning real values).

The most important extension to the original simulator from [BrHe95] was the implementation of a few very powerful operators for the computation of limits of user-defined sequences. The operators even allow to compute multi-valued functional limits like the complex square root [Mu98]. With their use, e.g. the full set of trigonometric functions from *sin* to *acosech* could be implemented easily using a simple Taylor series approach. The powerful AGM methods [Bo87,Bt76,Sch90] have been used for special values like π or $\log(2)$ or for very efficient implementations of $\log(x)$.

This ability of computing limits seems to be a unique feature of the iRRAM. The limited scope of algebraic or symbolic computations on real numbers is left as soon as these operators are used. From this point on, Type 2 Theory of Effectivity (TTE) [Ko91,We95,We97] is the best fitting theoretical model. This implies e.g. that computable functions must essentially be continuous, that it is no longer possible to check whether two real numbers are equal, etc.

A certain relaxation of the 'law' that computability implies continuity has been discussed e.g. in [BrHe94,BrHe95,Br96,Br99]: Instead of computing ordinary functions we may also compute set-valued functions F, e.g. $F : \mathbb{R} \to 2^{\mathbb{R}}$. Computing such a function F for an argument x means that the result of the computation of F on x must be one of the values in the set $F(x)$.

In the iRRAM the set-valued F appears like an ordinary (but *multi-valued*) function f, where we use the notation $f : \mathbb{R} \rightsquigarrow \mathbb{R}$ to express the multi-valuedness. Here the value $f(x)$ must be one of the elements of $F(x)$. The actual choice of the value is hidden from the user and appears to be nondeterministic: The same x may lead to different results $f(x)$ at different points of a computation.

The source code for the iRRAM is ordinary `C++`, as already said. However, some restrictions are necessary: The user should *not* use the normal IO operations, as this could lead to surprising and annoying results. Dynamic allocation of memory with `malloc` should be used as rarely as possible, while the use of `alloca` should be possible without problems. The use of external libraries has to be handled with care, as they will normally not be suited for the special semantics of the iRRAM.

With these restrictions, `C++` looses a lot of its universal character, but more than enough is left to do numerical work. The following simple example contains some of the most important features of the implementation and should give a rough impression of the implementation:

```
#include "REALLIB.h"

REAL maxapprox (long k, const REAL& x, const REAL& y){
   if ( positive(x-y,k) ) return x;
   else return y;
};

REAL max (const REAL& x, const REAL& y){
   return limit(maxapprox,x,y);
};

void compute() {

REAL x1 =  3.14159;
REAL x2 = "3.14159";
REAL x3 =  pi();

```

```
18  rwrite(x1,-200,65);           rprintf("\n");
19  rwrite(x2,-200,65);           rprintf("\n");
20  rwrite(max(x3,x2),-200,65);   rprintf("\n");
21  }
22
23  int main (int argc,char **argv) {
24  iRRAM_initialize(argc,argv);
25  iRRAM_exec(compute);
26  }
```

We will briefly explain this program in the following:

- Line 1 is the import of the necessary definitions.
- Lines 3-6 show an example of a simple self-defined real-valued function.
- Line 4 shows applications of the usual control structures in `C++` and of the overloading of operators that allow to build arithmetic expressions, here simply `x-y`. The function `positive(x-y,k)` is an 'intrinsic' (see below) multi-valued test whether `x-y` is positive. In the interval $x-y \in (-2^k, 2^k)$, the result may be wrong, i.e it may be `true` as well as `false`. In consequence, `maxapprox` returns a value that differs from the maximum of x and y by at most 2^k.
- Lines 8-10 are an example for the limit computing capabilities of the iRRAM: The maximum of `x` and `y` is the limit of `maxapprox(k,x,y)` for `k` $\to \infty$.
- Lines 12-21 define the core procedure `compute()` of this iRRAM example.
- Lines 14-16 show three of the possible initializations of `REAL` variables: from a double, from a string, or from a function (returning π).
- Lines 18-20 show I/O that has to be done through special functions. In this example values are printed with an error of at most $2^{-200} \approx 10^{-61}$ and with a width of exactly 65 characters (including the exponent), so each shown decimal has to be correct and the resulting output is as expected:

  ```
  +.31415899999999999882618340052431449294090270996093750000000E+0001
  +.31415900000000000000000000000000000000000000000000000000000E+0001
  +.31415926535897932384626433832795028841971693993751058209 7E+0001
  ```

  The first line of the output shows the result of the initialization from a `C++` double with a mantissa of 53 bits leading to about 15 correct decimals. However, the full conversion to a decimal form gives one nonzero decimal for each of the 53 bits. The second line comes from the (much slower) initialization from a string, where we get the precise intended value of the string constant. The third line shows the value of π (which is computed using an AGM iteration, see e.g. [Ba88,Bo87]).
- Lines 23-26 define `main`, which is optional. If `main` is missing, exactly this four line definition of `main` is used as default. This allows to use the iRRAM in two ways: (i) As a package for exact arithmetic with a special semantics explained in the next section, and (ii) as a tool for approximate arithmetic with 'usual' semantics explained in section 14.

The paper is structured as follows: In the following two sections we give a sketch of the semantics and of the corresponding simulation. In section 4 to 9 we will explain this simulation in more detail and with emphasis on those implementational details that have great influence the efficiency. Section 10 is devoted to the limit computing capabilities of the iRRAM. The rest of the paper aims at applications and gives an overview of additional features of the implementation.

2 Semantics of the iRRAM

The iRRAM implements an imperative programming language for real numbers, as said in the previous section. We will give a rough idea of the theoretical background by sketching a corresponding operational semantics (without going into all the details). This greatly simplifies the explanation of the concepts of the iRRAM. The later sections will show how the semantics is realized by the implementation.

In the following, let $\mathcal{P}$ be a program for the iRRAM, or more precisely: a program $\mathcal{P}$ that is called via `iRRAM_exec(`$\mathcal{P}$`)`.

- We assume that input and output of $\mathcal{P}$ may each consist of two vectors of natural and real numbers, i.e. we have an *input and output space* $\mathbb{M} = \mathbb{N}^* \times \mathbb{R}^*$. Here the natural numbers might represent encoded values of other sets as long as these are denumerable.
- For simplicity, we assume that $\mathbb{S} := \mathbb{N}^* \times \mathbb{R}^* \times \mathbb{M}$ is the set of possible *configurations* during the computation of $\mathcal{P}$. We further assume that appropriate encodings are used to represent the state of the program execution as well as the values of all discrete valued variables (including integers, floating point numbers, strings, pointers etc.) of $\mathcal{P}$ as natural numbers, which are given as the first component m_d of a configuration $s = (m_d, m_r, (o_d, o_r))$. The second component m_r represents the real-valued variables within the program, and the third component consists of the discrete output o_d and the real output o_r produced during the computation. The input to $\mathcal{P}$ is not included in these configurations.
- On the configurations, the progress of the computation can be expressed via a *transition relation* $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} \subseteq \mathbb{S} \times \mathbb{S}$ that depends on the program $\mathcal{P}$ and a given input vector $\mathcal{I} \in \mathbb{M}$, or equivalently as a (multi-valued!) transition function $\mathcal{T}_{\mathcal{I},\mathcal{P}} : \mathbb{S} \rightsquigarrow \mathbb{S}$.
  The finite iteration $\overset{n}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}}$ and the transitive closure $\overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}}$ of $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$ are defined as usual, e.g. $s \overset{n}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} s'$ iff $s = s_0 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_1 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} \cdots \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_n = s'$ for some states s_i.
  We assume that the output is append-only, i.e. it can neither be read or rewritten. Using $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$, this property can be expressed via:

$$\begin{gathered}(m_d, m_r, (o_d, o_r)) \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} (m'_d, m'_r, (o'_d, o'_r)) \\ \text{implies } o'_d = o_d \circ o''_d, o'_r = o_r \circ o''_r \text{ for some } o''_d, o''_r\end{gathered}$$

  and

$$\begin{gathered}(m_d, m_r, (o_d, o_r)) \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} (m'_d, m'_r, (o_d \circ o''_d, o_r \circ o''_r)) \\ \text{iff } (m_d, m_r, (\varepsilon, \varepsilon)) \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} (m'_d, m'_r, (o''_d, o''_r))\end{gathered}$$

  Here ε denotes the empty string and $\circ$ denotes the concatenation of strings.
  Because of the enormous complexity of `C++` and the encodings necessary to achieve the simple configuration set $\mathbb{S}$, we will *not* explicitly define $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$ here, but it should correspond to 'elementary' operations within the program, like entering or leaving loops, evaluation of (sub-)expressions, assignments, etc.

Basic operations of the iRRAM on the data type `REAL` like elementary arithmetic are also assumed to be executed within one single transition! These operations will be called *intrinsic* in the following. It is the appropriate choice of these intrinsics that determines the computational power and as well as the implementability of the iRRAM. We will only use intrinsics that are computable multi-valued function in the sense of TTE, which implies certain restrictions: For example, there is no test for equality of real numbers.

- We assume there is a distinct initial configuration $s_0 = (\varepsilon, \varepsilon, (\varepsilon, \varepsilon))$. Furthermore there are a set $\mathbb{S}^+$ of configurations, where the program terminates successfully, and a set $\mathbb{S}^-$, where the program fails with an error, for example due to division of a real by zero. So for $s \in \mathbb{S}^+ \cup \mathbb{S}^-$, there is no s' with $s \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s'$.

  A *computation* is either a finite sequence $s_0 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_1 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_2 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} \dots s_n$ such that $s_n \in \mathbb{S}^+ \cup \mathbb{S}^-$ (called terminating or failing computations) or an infinite sequence of states $s_0 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_1 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_2 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} \dots$ with $s_i \notin \mathbb{S}^+ \cup \mathbb{S}^-$ for all i.

  Please note that due to the multi-valued nature of $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$ (and similar to nondeterministic machines), there may be many distinct computations and there may even exist computations of different types for the same input $\mathcal{I}$!
- Finally, the computed multi-valued function $\mathcal{S}_\mathcal{P} \colon \mathbb{M} \rightsquigarrow \mathbb{M}$ is defined by

  $$\mathcal{S}_\mathcal{P}(\mathcal{I}) = \{\mathcal{O} \mid s_0 \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} (m_d, m_r, \mathcal{O}) \in \mathbb{S}^+\}$$

  for any admissible $\mathcal{I}$. Here an input $\mathcal{I} \in \mathbb{M}$ is called admissible, iff $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$ has neither failing nor infinite computations.

  So $\mathcal{S}_\mathcal{P}(\mathcal{I})$ is only defined if all possible computations are terminating, and then the value $\mathcal{S}_\mathcal{P}(\mathcal{I})$ consists of all possible results from these computations.

3 Simulative concept of the iRRAM

In order to realize this semantics, the implementation of the iRRAM essentially simulates the real valued parts of $\mathcal{P}$ while preserving (almost) any computation on the discrete parts. Internally, the simulation is done within the access method to the real numbers. In consequence, there is no overhead to the discrete parts of the computations; a purely discrete program within the iRRAM would be as fast as outside of the iRRAM.

- The iRRAM uses a representation of real numbers that is based on a subset of the intervals with rational endpoints. So let $\mathbb{J} = \{(l, r) \mid l < r,\ l, r \in \mathbb{Q}\}$.

  A real number x is uniquely determined by an infinite sequence

  $$\mathcal{J} = ((l_0, r_0), (l_1, r_1), (l_2, r_2), \dots) \in \mathbb{J}^\infty$$

  iff $\lim l_i = \sup l_i = \inf r_i = \lim r_i (= x)$. In this case we write $\varrho(\mathcal{J}) = x$, so ϱ is a (partial) surjection from $\mathbb{J}^\infty$ onto $\mathbb{R}$. Here we only use that all the single intervals contain x and finally become arbitrarily small; nested intervals or a given

convergence speed are not necessary. This *representation* ϱ of the real numbers can be extended in an obvious way to a surjection ϱ from $(\mathbb{J}^\infty)^*$ onto $\mathbb{R}^*$.
Instead of the input space $\mathbb{M} = \mathbb{N}^* \times \mathbb{R}^*$, we use $\overline{\mathbb{M}} = \mathbb{N}^* \times (\mathbb{J}^\infty)^*$, i.e. each real number x is replaced by an (arbitrary) $\mathcal{J} \in \varrho^{-1}(x)$. We will use $\mathcal{I} \overset{I}{\sim} \overline{\mathcal{I}}$ to denote that $\mathcal{I} = (i_d, i_r), \overline{\mathcal{I}} = (i_d, \overline{i_r})$, and $\varrho(\overline{i_r}) = i_r$.
The simulation will obviously not be able to write a real number or an infinite sequence of intervals within finite time, so as an (intermediate) representation of the output we use $\overline{\mathbb{M}}^f = \mathbb{N}^* \times (\mathbb{J}^*)^*$. The relation $\overset{O}{\sim}$ between $\mathbb{M}$ and $\overline{\mathbb{M}}^f$ that corresponds to the simulation of output is more complex than $\overset{I}{\sim}$: For $\mathcal{O} = (o_d, x_1 \circ \ldots \circ x_n) \in \mathbb{M}$ and $\overline{\mathcal{O}} = (\overline{o}_d, (J_{1,1} \circ \ldots \circ J_{1,j_1}) \circ \ldots \circ (J_{\overline{n},1} \circ \ldots \circ J_{\overline{n},j_{\overline{n}}})) \in \overline{\mathbb{M}}^f$ we write $\mathcal{O} \overset{O}{\sim} \overline{\mathcal{O}}$ iff $o_d \in \mathbb{N}^*$ is a prefix of $\overline{o}_d \in \mathbb{N}^*$, $n \leq \overline{n}$, and $x_k \in \bigcap_{j \leq j_k} J_{k,j}$ for any $k \leq n$. So all components from $\mathcal{O}$ must be present in $\overline{\mathcal{O}}$, either in an identical form for the discrete parts, or at least in a ‘consistent’ way for the real parts. It is important to notice that $\overline{\mathcal{O}}$ may have more components than $\mathcal{O}$.

– The configuration set of the simulation will be $\overline{\mathbb{S}} := \mathbb{N}^* \times \mathbb{J}^* \times \overline{\mathbb{M}}^f \times \mathbb{Z} \times \mathbb{N}^*$. Please note that $\overline{\mathbb{S}}$ is denumerable, in contrast to $\mathbb{S}$. So we are able to represent these configurations using ordinary data structures from `C++`.
The two configuration sets are connected via a simulating relation $\overset{S}{\sim}$ with

$$s = (m_d, x_1 \circ \ldots \circ x_n, \mathcal{O}) \quad \overset{S}{\sim} \quad \overline{s} = (\overline{m}_d, J_1 \circ \ldots \circ J_{\overline{n}}, \overline{\mathcal{O}}, \overline{p}, \overline{q})$$

iff $m_d = \overline{m}_d$, $n = \overline{n}$, $x_i \in J_i$ for all $i \leq n$, and if $\mathcal{O} \overset{O}{\sim} \overline{\mathcal{O}}$. So the simulating configuration $\overline{s}$ covers the same information m on the discrete part of the simulated configuration s (i.e. ordinary data structures and process state), but represents each internal real by just one interval. The meaning of the *precision bound* $\overline{p}$ and the *multi-value cache* $\overline{q}$ will be explained later on.
The initial state of the simulation will be $\overline{s}_0 = (\varepsilon, \varepsilon, (\varepsilon, \varepsilon), \overline{p_0}, \varepsilon)$ (again, $\overline{p_0}$ will be explained later).

– On these simulating configurations, the progress of the computation is given by a transition relation $\overrightarrow{\overline{\mathcal{I},\mathcal{P}}} \subseteq \overline{\mathbb{S}} \times \overline{\mathbb{S}}$ (with iterates $\overset{n}{\overrightarrow{\overline{\mathcal{I},\mathcal{P}}}}$ and closure $\overset{*}{\overrightarrow{\overline{\mathcal{I},\mathcal{P}}}}$).
In contrast to the multi-valuedness of $\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}$, this transition $\overrightarrow{\overline{\mathcal{I},\mathcal{P}}}$ will be single-valued, i.e. for any given configuration $\overline{s}$, there is at most one configuration $\overline{s}'$ with $\overline{s} \overrightarrow{\overline{\mathcal{I},\mathcal{P}}} \overline{s}'$.
Again, the output $\overline{\mathcal{O}} = (\overline{o}_d, (J_{1,1} \circ \ldots \circ J_{1,j_1}) \circ \ldots \circ (J_{n,1} \circ \ldots \circ J_{n,j_n}))$ is append-only, i.e. a written natural number will be appended to $\overline{o_d}$, a written interval will either appended to one of the sequences $(J_{k,1} \circ \ldots \circ J_{bek,j_k})$ or will start a new sequence. This is necessary to ensure that the output produced at any point during the simulation will remain valid forever.
In consequence the simulator as implemented in `C++` does not need to store any produced output. Although the configurations contain the output (to simplify the description), it will not lead to any noticeable overhead in the execution of the simulation.

– For inputs $\mathcal{I} \in \mathbb{M}$ and $\overline{\mathcal{I}} \in \overline{\mathbb{M}}$ with $\mathcal{I} \overset{I}{\sim} \overline{\mathcal{I}}$, the transition relations have the following two central properties:

correctness:
For any $\overline{s} \in \overline{\mathbb{S}}$ with $\overline{s_0} \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{*} \overline{s}$ there is (at least one) $s \in \mathbb{S}$ with $s \overset{S}{\sim} \overline{s}$ and $s_0 \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} s$

restricted completeness:
For any $s \overset{S}{\sim} \overline{s}$ and s' with $s_0 \overset{*}{\underset{\mathcal{I},\mathcal{P}}{\Longrightarrow}} s \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s'$ and $\overline{s_0} \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{*} s$ there are $s'' \overset{S}{\sim} \overline{s}''$ such that $s \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s''$ and $\overline{s} \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{*} \overline{s}''$.

It is important to note that we have a one step transition from s to s'', but it may take more time to reach $\overline{s}''$ from $\overline{s}$.

– Instead of the successful configurations $\mathbb{S}^+$, there is a set $\overline{\mathbb{S}}^\circ$ of *reiteration configurations* such that $s \sim \overline{s}$ and $s \in \mathbb{S}^+$ implies $\overline{s} \in \overline{\mathbb{S}}^\circ$.
For such a reiteration configuration, the transition relation $\xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{}$ is defined as

$$(\overline{m_d}, \overline{m_r}, \mathcal{O}, \overline{p}, \overline{q}) \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{} (\varepsilon, \varepsilon, \mathcal{O}, \overline{p}', \overline{q})$$

with $\overline{p}' \ll \overline{p}$, i.e. the simulation is essentially restarted with an improved precision bound p'. Only the multi-value cache $\overline{q}$ is saved during this transition. Of course, the yet produced output can not be deleted, bud it has no influence on the continuation of the simulation.
The multi-value cache q guarantees that a restarted simulation will take the same computational path again: If we have

$$\overline{s_0} \;=\; (\varepsilon, \varepsilon, (\varepsilon, \varepsilon), \overline{p_0}, \varepsilon) \quad \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{*} \quad (\overline{m_d}, \overline{m_r}, (\overline{o_d}, \overline{o_r}), \overline{p}, \overline{q}) = \overline{s}$$

for a reiteration configuration $\overline{s}$, then we will get

$$\overline{s} \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{} (\varepsilon, \varepsilon, (\overline{o_d}, \overline{o_r}), \overline{p}', \overline{q}) \quad \xrightarrow[\mathcal{I},\overline{\mathcal{P}}]{*} \quad (\overline{m_d}, \overline{m_r}', (\overline{o_d}, \overline{o_r}'), \overline{p}'', \overline{q}) = \overline{s}'$$

where $\overline{p}'' \ll \overline{p}'$. Furthermore, although $\overline{m_d}$ and $\overline{m_r}$ are deleted in the first transition after $\overline{s}$, $\overline{m_d}$ will be reconstructed afterwards, and $\overline{m_r}'$ will have the same length as $\overline{m_r}$ but will be contain new, usually smaller, but consistent intervals. The discrete output $\overline{o_d}$ will remain totally unchanged, and the real output $\overline{o_r}'$ will have the same number of interval sequences, but each of these will usually be extended in a consistent way.
In fact, there are many more reiteration configurations than just those simulating $\mathbb{S}^+$: As soon as the intervals in $\overline{m_r}$ are too big to allow a correct continuation of the simulation, a reiteration is done. This will be the case e.g. if we have compare two real variables x_i and x_j, but the simulating intervals J_i and J_j have a nonempty intersection. In these cases, the corresponding $\overline{s}$ will be a reiteration configuration, but the later $\overline{s}'$ (with smaller intervals) may allow for a normal continuation of the simulation.
The same may hold for failing configurations $s \in \mathbb{S}^-$. If the failure is due to an error in the discrete part (like a division of a double by zero), a similar error will appear in the simulation, so a failure in the simulated computation may lead to a failure in the simulation. But if the failure comes from the real part, a reiteration may be executed. For example, a simulated division of a real by zero will always be a reiteration configuration, so it leads to an infinite sequence of reiterations, where none of them will be able get to get past this division operation.

– The reiterations lead to the following third central property of the simulation: **convergence:**

If $\mathcal{I}$ is admissible and $\mathcal{I} \overset{I}{\sim} \overline{\mathcal{I}}$, then there will be one terminating computation

$$s_o \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_1 \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} \dots \underset{\mathcal{I},\mathcal{P}}{\Longrightarrow} s_n = (m_d, m_r, (o_d, o_r) \in \mathbb{S}^+$$

such that there is a (generally infinite) simulating computation

$$\overline{s_0} \underset{\overline{\mathcal{I}},\mathcal{P}}{\overset{*}{\Longrightarrow}} \overline{s_n}^{(1)} \underset{\overline{\mathcal{I}},\mathcal{P}}{\overset{*}{\Longrightarrow}} \overline{s_n}^{(2)} \dots$$

with $s_n \overset{S}{\sim} \overline{s_n}^{(k)}$ for any k, where the output components $\overline{o_d}^{(k)}$ of $\overline{s_n}^{(k)}$ are identical to o_d, and the $\overline{o_r}^{(k)}$ converge to o_r.

So the output from the simulation will converge to one of the possible results from $\mathcal{S}_{\mathcal{P}}(\mathcal{I})$ by following exactly one terminating computation infinitely often. So our simulator shows that $\mathcal{S}_{\mathcal{P}}$ is a computable multi-valued function in the sense of [BrHe94].

Please note that in many applications of the iRRAM, we will only have discrete (i.e. non-real) output o_d and the real output o_r remains empty. Usually, o_d will consist of approximations to real numbers with some finite precision. Here the simulation may stop as soon as a configuration from $\mathbb{S}^+$ is reached, as this discrete output will not be changed again. In this case, we get finite simulations and we also may use the iRRAM as a tool for ordinary approximative computations within arbitrary other programs!

4 Basic Multiple Precision Arithmetic

The backend of the iRRAM consists of an implementation of the intervals that are used to approximate the real numbers. Here multiple precision arithmetic is used, for which many freely available packages exist. Pioneering work in this area had been done by R.P. Brent [Bt75,Bt76,Bt78] with his MP package in `FORTRAN 77`. An overview, although quite old, can be found e.g. in the file `BIGNUMS.TXT` [Rio94] available freely in the internet. At the moment, one of the most elaborate packages is `GMP 3.1` (Gnu Multiple Precision, [Gr00]), using assembler routines at time critical parts.

In its first prototype from August 1996, the iRRAM was based on a small MP package called `LongReal` written by the author. Meanwhile, also `GMP` can be used in two different ways (see below). As the interface to the MP backend is very simple, it would be easy to use other MP packages as well. At the time of writing, an interface definition to MPFR [Zi00] was already started, but not fully functional.

The necessary routines for multiple precision arithmetic can essentially be divided in two parts:

– A few low level but time critical operations that work on arbitrarily long integers (`GMP:gmp_n`) or on the arbitrarily long mantissa of fixed point numbers (`LongReal:Dyadic_Base`), essentially these are the 4 basic arithmetic operations (reduced to these sets) and a shifting operator.

– A higher level of about 20 routines for arbitrary MP numbers (`gmp_f` and `Dyadic`), essentially these are arithmetic operations, comparison, type conversions, etc. Internally, the low level routines are applied whenever possible.

Both `GMP` and `LongReal` allow access to both levels. `LongReal` is also able to use the low level routines of `GMP`. So at compile time, the user can choose between 3 different flavors of the iRRAM: GMP (using `gmp_f` and `gmp_n`), LR for the original `LongReal` (using `Dyadic` and `Dyadic_base`), and LRGMP (using `Dyadic` and `gmp_n`). This approach has the advantage that errors in the implementation can be found much easier. In fact, with the use of LRGMP two errors in the GMP 3.0 routines (in addition to several other internal errors) could be found.

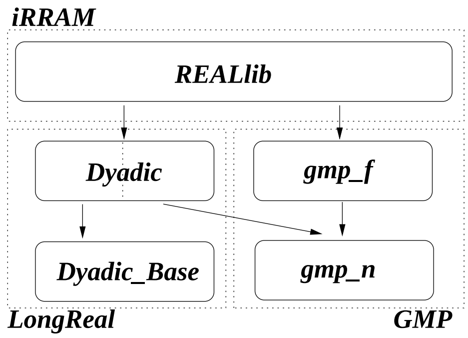

Fig. 1. The iRRAM and its backends

One quite big conceptual difference between `GMP` and `LongReal` is how the still finite precision is handled (i.e. how rounding errors are created): In `GMP`, each variable is allocated with a (changeable) amount s of memory, and after an operation $c = a \circ b$ the variable c contains the best approximation to the exact result $a \circ b$ fitting into the allocated memory, so the single operations of `GMP` work with relative precision of results and each operation introduces an error of order $c \cdot 2^{-s}$.

The concept of `LongReal` is different: After fixing a desired precision p, the operation $c = a \circ b$ yields a result c with $|c - a \circ b| \leq 2^p$, so the single operations work with an absolute precision independent from a, b, and '$\circ$'. In consequence, the user does not need to know anything about memory allocation and can concentrate on the error propagation.

All these three flavors of the iRRAM implement a type of floating point numbers with a variable sized mantissa and a 32 bit exponent called `DYADIC`. On the user level, the differences are hidden: the concept of precision is as in `LongReal`. Overloading of the arithmetic operators further simplifies the usage. In the following we show a few lines of the interface: Setting the precision and overloading the basic arithmetic operators.

```
class DYADIC
{
public:
static void setprec(long p);
...
friend DYADIC  operator +  (const DYADIC& x, const DYADIC& y);
friend DYADIC  operator -  (const DYADIC& x, const DYADIC& y);
friend DYADIC  operator *  (const DYADIC& x, const DYADIC& y);
friend DYADIC  operator /  (const DYADIC& x, const DYADIC& y);
friend DYADIC  operator -  (const DYADIC& x);
...
};
```

The implementation has a restricted form of own memory management for multiple precision variables as it manages a pool of variables that are already initialized. This significantly reduces the overhead for for the frequent creating of new real object, that arises from the overloading operators in `C++`.

5 Simplified Interval Representation

In the previous chapter we presented a simple interface to multiple precision numbers. A similar concept with overloaded operators is used for an interface to real numbers that are represented by the data type `REAL`. As explained in section 3, we only need to use single intervals. So internally, each variable x of type `REAL` is implemented as a multiple precision number d together with information on the absolute error $e \geq |d-x|$. It has already been explained how the infinite character of the reals is represented through these intervals. Essentially, in any stage of the computation, we work with intervals $(d-e, d+e)$ knowing that x must be included in the interval.

The error information e could be stored in many different ways and especially in many different resolutions, and is not necessarily connected to the length of d. We might chose $\{2^{-n} | n \in \mathbb{N}\}$ as set of possible error informations, which could be called a 'precision of n bits'. As an other possibility, we might choose $\{z \cdot 2^p | z \in \mathbb{N}, p \in \mathbb{Z}\}$, so that we are able to describe errors very precise in the manner of a true interval arithmetic.

Both extremes have disadvantages: With the first approach, we would only need one additional integer to represent e (i.e. 4 byte) but we would loose at least one bit of information (concerning the mantissa of d) on any operation, demanding a very high initial precision. The second approach implies storing e as an own multiple precision number (or storing both values $d-e$ and $d+e$), so the storage is doubled and operations are slower.

The iRRAM implements an intermediate solution: The error is stored as a value from $\{z \cdot 2^p | z \in \mathbb{N}, p \in \mathbb{Z}\}$, but restricted to $z < 2^{gb}$ and $|p| < 2^{maxexp}$. The values of gb and $maxexp$ depend on the CPU and the MP package in use: For usual 32 bit CPUs and LR we have $gb = 30$ and $maxexp = 29$.

So internally, each variable x of type `REAL` is implemented as a (simplified) interval $(d \pm e)$, where d is a multiple precision number and e simply consists of two values of type `long` to represent p and z. In consequence, each elementary operation on the intervals consists of one multiple precision operation (where the complexity increases with growing precision) and a few simple operations on integers (with fixed complexity), and we have a significant reduction in the growth of errors (compared to n-bit precision) during single operations by only a small overhead, especially in computing scalar products of vectors where all components are of similar error.

6 Controlling the Precision

Each single operation $c = a \circ b$ on `REAL` variables is transformed to the underlying intervals: If $a \sim (d_a \pm e_a)$ and $b \sim (d_b \pm e_b)$), the multiple precision number d_c of the result $c \sim (d_c \pm e_c)$ is computed from d_a and d_b with an absolute precision 2^p.

Here p essentially depends on the actual errors $e_a = z_a \cdot 2^{p_a}$ and $e_b = z_b \cdot 2^{p_b}$ of a and b: If e_a and/or e_b are quite large, it would be a waste of time to compute d_c very precisely. On the other hand, it will not always be advantageous to choose a very high precision, if e_a and e_b are very small: Already simple divisions with precise values (like $c = REAL(1)/REAL(3)$) would lead to unwanted and very high computational costs. In order to avoid this, p may not be smaller a certain *precision bound* $\overline{p}$ (that will change during the execution of a program and will be explained later in more detail). (Please note that we are working with errors of size 2^p and not 2^{-p}, so p should be larger and not smaller than $\overline{p}$! Usually, p and $\overline{p}$ will be negative.)

As an example, we examine the resulting error propagation for the division of reals: Let $d_a, e_a = z_a \cdot 2^{p_a}$ and $d_b, e_b = z_b \cdot 2^{p_b}$ be given. In order to exclude division by zero, suppose $|d_b| > 2 \cdot e_b > 0$, so $|d_b|/2 \leq |b|$. Suppose we compute d_c with $|d_c - d_a/d_b| \leq 2^p$ for a certain p that will be specified later.

For $i = a$ or $i = b$, let $s_i \in \mathbb{Z}$ be as small as possible with $|d_i| < 2^{s_i}$, i.e. usually $2^{s_i - 1} \leq |d_i| < 2^{s_i}$. In case of $d_i = 0$, s_i will be the above minimal value $2^{-maxexp}$ due to the limitations of the implementation of the multiple precision numbers.

For a, b with $|d_a - a| \leq e_a$ and $|d_b - b| \leq e_b$ we have

$$
\begin{aligned}
&|\frac{a}{b} - d_c| \\
&\leq |\frac{a}{b} - \frac{d_a}{d_b}| + 2^p = |\frac{(a - d_a) \cdot d_b + d_a \cdot (d_b - b)}{b \cdot d_b}| + 2^p \\
&\leq \frac{e_a \cdot |d_b| + e_b \cdot |d_a|}{|d_b|/2 \cdot |d_b|}| + 2^p \\
&\leq \frac{z_a 2^{p_a + s_b} + z_b 2^{p_b + s_a}}{2^{2s_b - 1}} + 2^p = z_a 2^{p_a - s_b + 1} + z_b 2^{p_b + s_a - 2s_b + 1} + 2^p
\end{aligned}
$$

Choosing $p = \max\{p_a - s_b + 1, p_b + s_a - 2s_b + 1, \overline{p}\}$ we have $|a/b - d| \leq 2^p \cdot (z_a \cdot 2^{p_a - s_b + 1 - p} + z_b \cdot 2^{p_b + s_a - 2s_b + 1 - p} + 1)$, where the value in parentheses consists of non-positive exponents of 2 and the integer factors z_a and z_b.

Using appropriate right shifts of z_a and z_b and ordinary integer arithmetic, we can easily compute the *integer* value $z_c = \lceil z_a \cdot 2^{p_a - s_b + 1 - p} + z_b \cdot 2^{p_b + s_a - 2s_b + 1 - p} + 1 \rceil$. So we have $|a/b - d_c| \leq 2^p \cdot z_c =: e_c$.

We have $z_c \le z_a + z_b + 1$, so the value of the error mantissa might grow from operation to operation. If $z_c \ge 2^{gb}$, we reduce z_c by computing $z_c' := z_c/4 + 1$ and $p' = p + 2$, so that still $2^p \cdot z_c \le 2^{p'} \cdot z_c'$, but now within the allowed range for the error mantissa. If the resulting exponent p' gets larger then 2^{maxexp}, we emit an 'overflow warning' and do a reiteration, that will be explained in detail in the next section.

7 Iterating a Computation

Essentially, all computations are done with approximations to the real numbers. Each single operation is executed with a certain precision that is influenced by the precision bound $\overline{p}$.

Due to the accumulation of errors during a longer computation, it may happen that the error of an approximation gets too big (e.g. to compare numbers). This is one of the points where the central concept of the iRRAM (taken from [BrHe95]) is used: *The whole computation* is repeated with a significantly better (i.e. smaller) precision bound $\overline{p}$! So instead of a single and fixed precision bound we have a sequence $\overline{p}_0 \gg \overline{p}_1 \gg \overline{p}_2 \ldots$, where a recomputation with $\overline{p}_{i+1}$ as precision bound instead of $\overline{p}_i$ will usually lead to better approximations.

The choice of these precision bounds has no influence of the correctness of the iRRAM simulation, as long as they get arbitrarily small. On the other hand, the time complexity may be quite different.

It is well known that the asymptotic complexity of sequences of iterated computations is of the same order as the complexity of the last iteration, if we have $\overline{p}_i = \overline{p}_0 - f^i$ for a fixed (e.g. rational) $f > 1$ and if all used operations and functions have a well behaved ('regular', see e.g. [Mu93]) complexity. Applications of such regular or 'smooth' complexity bounds have already been studied e.g. in [Bt76,FiSt74,CoAa89]

So in theory, the choice of $\overline{p_0}$ and f is not essential. Obviously, the starting precision as well as the increment in the iterated computations are critical points in practice. If we start with values for $\overline{p}_0$ and f that are too large, we loose computation time because we are too precise. On the other hand, small values lead to frequent reiterations.

The iRRAM uses $\overline{p}_i = \overline{p}_0 + g \cdot \frac{f^i - 1}{f-1}$, where $\overline{p}_0 = g = -50$ and $f = 1.25$. This leads to the following precisions bounds for the first iterations:

$\overline{p}_0$	$\overline{p}_1$	$\overline{p}_2$	$\overline{p}_3$	$\ldots$	$\overline{p}_{10}$	$\overline{p}_{15}$	$\overline{p}_{20}$	$\overline{p}_{25}$	$\ldots$
$\overline{p}_0 = -50$	$\overline{p}_0 + g = -100$	-162	-240	$\ldots$	-1703	-5502	-17096	-52477	$\ldots$

$\overline{p}_0$, q, and f are not fixed at compile time, but can be changed at the start of any program to optimize its time complexity. The initial value $\overline{p}_0 = -50$ was chosen because the IEEE double precision floating point numbers have a mantissa of slightly more than 50 bits.

An important example for reiterations is the division of reals: One of the assumptions above was $|d_b| > 2 \cdot e_b > 0$ to exclude a division by zero. If this inequality does not hold, we may either have attempted a true division by zero or (usually) the error e_2 in the approximation d_2 is too large to exclude such a division. So this is a point where a reiteration with higher initial precision should be initiated.

This also implies that a division by zero initiates an infinite loop of iterations instead of an exception as in usual arithmetic. As division can not be extended to a total continuous function, this effect is unavoidable.

The same holds with the tests on the real numbers: If we compare two numbers x and y, and the corresponding intervals have a nonempty intersection, we reiterate the computation in the hope that we will get smaller intervals with empty intersection. So comparing numbers leads to an infinite loop if they are identical, which is a well-known property in TTE. On the other hand, if applied to different arguments, comparing will eventually lead to the correct result, maybe on the cost of a high precision.

The current version of the iRRAM implements reiterations by a `longjmp`, i.e. an immediate return to an outer level of control. If the iRRAM is called with `iRRAM_exec(compute)` (which is the default), the jump (usually, but see the chapter on limits) leaves the procedure `compute` and returns to `iRRAM_exec`. Here the precision is incremented as above, then `compute` is restarted from the beginning. Coming versions of the iRRAM might use the exception handling mechanisms of C++ instead of `longjmp`.

8 Input and Output

The iterative character of the computations described above poses problems for interactive usage of the iRRAM: In each iteration each input from `stdin` and each output to `stdout` is repeated. Although this can be used to identify the iterations, any illusion of working with entire real numbers is destroyed.

To avoid this effect, the user should use the special IO-functions `rscanf` for reading one input and `rprintf` for writing (a variable number of) outputs. These functions are able to use the usual format specifiers from `C++`.

In addition, there is a function `rwrite(x,p,w)`, which prints a decimal approximation to a real number with an error of at most 2^{p} and with a width of w characters. If the real number is smaller than 2^{p}, the output consists simply of a single '0' padded with blanks.

Finally there is a function `rshow(x,w)` showing the value x in a field of at most max(9,w) characters. The shown result is correct, however the last decimal might differ by 1. In addition, denormalized output in the form e.g. ".*E-0003" just indicates that the value of x is below 10^{-3}. `rshow` does never lead to reiterations, but in case of a denormalized output, the result might not be very helpful.

The implementation of `rscanf` simply uses buffered input: the input from the user is copied to a finite buffer (currently of fixed size 100000 characters). On each iteration a pointer in the buffer is reset to the beginning of the buffer, then inputs are taken from the buffer moving this pointer. Only when the pointer reaches the end of the buffered input, new characters are appended to the buffer by reading from the standard input.

`rprintf`, `rshow(x,w)` and `rwrite(x,p,w)` simply use a counter for the number of outputs form previous iterations to determine whether a call should actually lead to new output. This counter can be considered as a part of the multi-value cache, as it value must survive the reiterations. Calling `rwrite(x,p,w)` may imply a reiteration if the necessary error bound is not given in the actual iteration.

Input and output of reals (i.e. of infinite objects, not just approximations like in `rwrite(x,p,w)`) are not implemented yet. As already said at the end of section 3, this implies that our programs may stop as soon as they exit from being called via `iRRAM_exec`. Output of real numbers could of course be easily implemented by opening an own output stream for each written real value. In reiterations we would simply append a new interval to the corresponding stream. But as these streams are necessarily of infinite length, the simulation must be continued forever with improved precision bounds after reaching terminating configurations. This could be useful on machines with more than one processor or in computer clusters, where we might pipe the output form one iRRAM into another one in order to achieve parallelization. Until know, it has simply not been necessary to implement these input or output functions.

9 Multi-valued Functions

In traditional arithmetic for single or double precision numbers, there are many functions that cannot be adopted directly to the iRRAM: Essentially, the semantics of the iRRAM must be continuous, whereas these functions are not continuous.

Sometimes, it is possible to implement similar functions by restricting the domains, like division with the exception of zero or the tests, where identical arguments lead to infinite loops.

Another possibility has been used in [BrHe94,BrHe95,Br96,Br99]: operations with uncertainty that usually give the correct result but also are allowed to give a 'slightly' wrong answer near points where the operation is not continuous in its arguments. So instead of computing with ordinary functions we may also compute with relations $R \subseteq \mathbb{R}^2$ or set-valued functions $F : \mathbb{R} \to 2^{\mathbb{R}}$. (Both notions are equivalent, simply use $F(x) = \{y|(x,y) \in R\}$.) The underlying concept of a 'computable relation' had already been briefly defined in [We87].

Computing such a set-valued function F for an argument x means that the result of the computation of F on x must be one of the values in the set $F(x)$. An alternative approach (where a computation must compute *all* values of $F(x)$) has been intensively discussed in [Br99]. Although that approach has advantages from the theoretical side, any implementation would surely be quite ineffective. A comparison between the two approaches can be found in [Mu00].

This concept has been incorporated into the iRRAM since its very first prototype [Mu96]: In the iRRAM, the set-valued F appears like an ordinary (but *multi-valued*) function $f : \mathbb{R} \rightsquigarrow \mathbb{R}$, where the value $f(x)$ must be one of the elements of $F(x)$. The actual choice of the value is hidden from the user and appears to be nondeterministic: The same real number x may lead to different results $f(x)$ at different points of a computation.

In the following, we list most of the intrinsic multi-valued functions that are implemented in the iRRAM. Of course, the user can easily construct new functions of this kind, either through the usual control structures of `C++` or using a special limit operator that will be explained in the next chapter.

– `long size (const REAL& x);`

`size(x)` is one of the integer values $k = \lfloor \log_2 |x| \rfloor + 1$ or $k = \lceil \log_2 |x| \rceil + 1$. So the value of `size(x)` is not exactly fixed, but it is sure that $|\mathrm{x}|$ lies between 2^{k-2} and 2^k. The iRRAM is allowed to return any of the possible values. In consequent

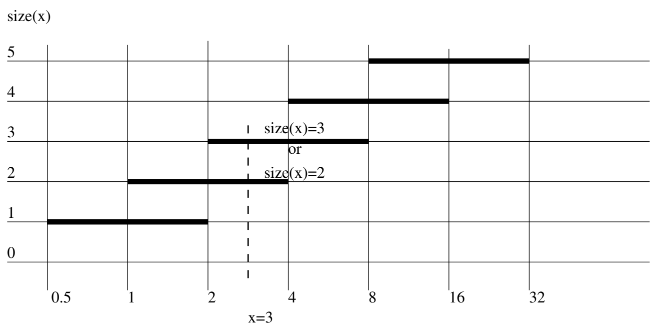

Fig. 2. A graph of the multi-valued function `size(x)`

calls with the same real value x, we might even get different values for `size(x)`. From the view of the programmer, this is a kind of nondeterminism in the program. On the simulating level, where x is represented as an interval $(d \pm e)$, the choice of k is deterministic: We simply compute $k = \lceil log_2(|d| + e) \rceil]$, which immediately implies $|x| < |d| + e \le 2^k$. If $e \le |d|/4$, then $|x| > |d| - e > (|d| + e)/2 > 2^{k-2}$, so we return k as the value of `size(x)`.
If $e > |d|/4$, the precision of the interval is not sufficient, so we invoke a reiteration. This implies for example, that usage with argument $x = 0$ leads to an infinite loop. Because of the limited exponent of the MP numbers, it is sufficient to use `long` as result type for `size`.

- `bool bound (const REAL& x,const long k);`
`bound(x,k)` is a test whether $|\mathrm{x}|$ is smaller than 2^k. The result will be correct if $|\mathrm{x}| > 2^k$ or $|\mathrm{x}| \le 2^{k-2}$. In any other case, the result also is defined, but may be incorrect. Calling the function may imply a reiteration, but the test will finish as soon as the error is small enough:

$$\mathtt{bound}(x, k) = \begin{cases} \text{true}, & |x| \le 2^{k-2} \\ \text{true or false}, & 2^k \ge |x| > 2^{k-2} \\ \text{false} & |x| > 2^k \end{cases}$$

So $\mathtt{bound}(x, k) = \text{true}$ implies $|\mathrm{x}| \le 2^k$, while $\mathtt{bound}(x, k) = \text{false}$ implies $|\mathrm{x}| \ge 2^{k-2}$.
If it is necessary to check that a number is small enough (usually given as the error of an approximating algorithm), `bound(x,k)` should be preferred to `size(x)<k` as there is no singularity for $x = 0$.

– `bool positive    (const REAL& x, const long k);`
  `positive(x,k)` tests whether `x` is positive. In the interval $(-2^k, 2^k)$, the result may be wrong:

$$\texttt{positive}(x,k) = \begin{cases} \text{true}, & x > 2^k \\ \text{true or false}, & -2^k \leq x \leq 2^k \\ \text{false} & x < -2^k \end{cases}$$

  In contrast to the comparison operator '$x < 0$', the function `positive` has the advantage to be total, so it can be used even if it is unknown whether $x = 0$. Calling `positive(x,k)` may invoke reiteration, but again the test will finish as soon as the error is small enough.

– `DYADIC  approx     (const REAL& x, const long p);`
  `approx(x,k)` results in a `DYADIC` result d such that $|\mathtt{x} - d| \leq 2^{-k}$. Calling the function may invoke reiteration. The function can be used to enhance the set of `DYADIC` functions: If `REAL f(const REAL& x)` computes $f : \mathbb{R} \to \mathbb{R}$, then for any `DYADIC d,` we can get a `DYADIC` approximation to $f(d)$ with an error of at most 2^p simply with `approx(f(d),p)` (using an implicit type conversion from `d` to type `REAL`).

These functions can be used e.g. in the construction of loops: For any p and $x \geq 0$, the following (multi-valued!) function returns a real y such that $|y - \sqrt{x}| \leq 2^p$ using Heron's iteration algorithm for the square root.

```
REAL sqrt1(long p,REAL x) {
  REAL a=1,b=x;
  do {
     a=(a+b)/2;
     b=x/a;
  } while ( !bound(a-b,prec) );
  return a;
}
```

In our simulation of the iRRAM, we must be careful when an intrinsic multi-valued function is called: In different iterations, we get different intervals, so that the (deterministic) computation of the function might lead to different results. For example, when computing $\mathtt{size}(x)$, an early iteration might deliver $\lceil \log_2 |x| \rceil + 1$ as result, based on some interval $(d \pm e)$. A later iteration might get another interval $(d' \pm e')$ with smaller $d' + e' < d + e$, so at the same stage of the the computation we now get $\lfloor \log_2 |x| \rfloor + 1$. Although both results are consistent with the definition of `size`, we might have trouble with the synchronization of the flow control in the iterations, as can easily be seen in the following few lines of source code:

```
if ( size(x) <p  ) {
  rprintf("Input A:");   rscanf(a);
} else {
  rprintf("Input B:");   rscanf(b);
}
```

This could lead to a totally different behavior in different iterations. The iRRAM solves this problem using its multi-value cache: It simply recalls any known values from the cache before computing new values (that are immediately appended to the cached values). In consequence, any flow of information from the continuous world of reals to the discrete world stays unchanged through the reiterations. This again implies that the flow of control in a computation will be repeated in reiterations up to the point where the previous iteration ended. As a side effect, we only need to store a sequence of the results and a pointer to the last recalled valued, and do not need any information on the context of the operations to reproduce their results.

The current implementation simply defines an array of 100000 `long`s to store the results of `size` or of tests and an additional array of the same size to store the results of approximations using `approx`. In consequence, these operations should be applied as rarely as possible!

In the following chapters, we discuss environments where caching the results is not necessary (or even causes errors). In such circumstances, the operations can be used freely without permanently occupying memory.

10 Computation of Limits

A unique feature of the iRRAM are several operators for the computation of limits of certain families of real numbers or real-valued functions. On one hand, this ability implies that the iRRAM e.g. is not able to check whether two given real numbers are equal (e.g. if one of them is such a limit). On the other hand, the limit operator is very powerful: Almost any important object in analysis is defined as some kind of a limit, e.g. computing the square root can be done by computing the limit of the function `sqrt1` from the previous chapter.

This capability of limit computing extends the computational power of the iRRAM in a way that the scope of algebraic of symbolic computations is left, also the computational model of Blum, Shub, and Smale[BSS89] is no longer applicable. The correct corresponding theoretical model is Type 2 Theory of Effectivity (TTE) [Ko91,We95,We97], although this model works essentially on a Turing machine level. Example definitions of computations on topological structures like $\mathbb{R}$ on an abstract level (i.e. without using representations) can be found in [TZ99] (but without multi-valued functions and without an internal limit operator) or [Br99] (but with a slightly different approach to multi-valued functions).

The iRRAM implements three types operators for limits:

- for families $\{a_p \mid p \in \mathbb{Z}\}$ of functions $a_p : M_1 \dashrightarrow M_2$ converging uniformly and with known speed to a *single-valued limit* f with $f(x) = \lim\limits_{p \to -\infty} a_p(x)$.
- for families $\{a_p \mid p \in \mathbb{Z}\}$ of functions similar to above, but where we have additional information on the limit f (essentially f must fulfill a *Lipschitz condition*).
- for families $\{a_p \mid p \in \mathbb{Z}\}$ of functions converging to a *multi-valued limit* $f(x)$.

Here M_1 and M_2 are the metric spaces of real numbers, real matrices, or complex numbers. The iRRAM is already prepared to deal with other metric spaces, although until now it was not necessary to implement such a case.

The limit operators need $\{a_p \mid p \in \mathbb{Z}\}$ as one of their parameters, usually as a function a of $p \in \mathbb{Z}$ and $x \in M_1$. This implies that there must be a program computing a, so the operators will only be applied to computable parameters, i.e. to families of (multi-valued) continuous functions.

We will discuss all three types in detail but, for simplicity, only for the case $M_1 = M_2 = \mathbb{R}$ (or $\mathbb{C}$).

10.1 Simple Limits

For limits of the first type we need a family $\{a_p\}$ e.g. of functions $a_p : \mathbb{R} \rightsquigarrow \mathbb{R}$ converging to a function $f : \mathbb{R} \dashrightarrow \mathbb{R}$ in the sense that $|a_p(x) - f(x)| \leq 2^p$ for any $x \in dom(f)$ (and any possible value of the multi-valued $a_p(x)$). The corresponding limit operator is

```
REAL  limit   (REAL a(long, const REAL&),  const REAL& x);
```

It should be emphasized that the functions a_p will usually be multi-valued, but the limit f must be single-valued. In consequence, f must be a (maybe partial) continuous function.

An example for $a_p(x)$ is `sqrt1(p,x)` already mentioned in the previous chapter; here the limit f is the square root defined on $\mathbb{R}_0^+$. In the iRRAM, we may now define the square root function by

```
REAL sqrt(const REAL& x) {return limit(sqrt1,x);}
```

This function `sqrt` can be used like any of the built-in elementary functions!

The idea of the implementation is quite simple: If we want to compute such a limit $\lim_{p \to -\infty} a_p(x)$ in an iteration, we simply compute and return $a_p(x)$ for some p, such that p tends to $-\infty$ with improving precision bounds $\overline{p_i}$ during the iterations.

Three problems must be faced: (1) in each iteration, the second argument `x` to `limit` is only known with errors, (2) the error of the computed result $a_p(x)$ is 2^p plus the error in the computation, and (3) the flow of control will be different in different iterations!

The argument `x` in `limit(a,x)` will not be exact during any stage of the computation. So we are only able to compute the values $a_p(x)$ with an error at least corresponding to the modulus of continuity of the function a_p using the given approximation to x. In practice, the best achievable result will usually be much worse, as the computation of $a_p(x)$ need not be optimal with respect to error propagation. To solve this problem, we try to compute $a_p(x)$ for several values p that are chosen the same way as the precision bounds: In the i-th iteration of the iRRAM using precision bound $\overline{p}_i$, we consider the values $a_p(x)$ for $p = \overline{p}_0, \overline{p}_1, \ldots, \overline{p}_i$. Then we take that result giving the smallest error (as the sum of the error from the computation of $a_p(x)$ and the bound 2^p for the difference between $a_p(x)$ and the limit $f(x)$).

So here we face the situation that a failure of a computation due to insufficient precision should *not* lead to a recomputation of the whole program. Instead of this, we have *local iterations* of the computations, i.e. the `longjmp` from within the computation of the limit must no longer lead to `iRRAM_exec`, but to the procedure implementing `limit`. In addition, `limit` has a `longjmp` to an outer level, if $a_p(x)$ can not be computed for any of the tested values of p due to the error in `x`. After that outer `longjmp`, the program will return to the computation of the limit with a better precision bound and hence a (usually) better approximation for the argument x. This ensures that eventually there will be successful tries of $a_p(x)$ even for arbitrarily small p. This again ensures that the error of the limit will get arbitrarily small in later iterations. This behavior of the limit operator (and of all the other intrinsic functions) is the origin for the conversion property defined in section 3.

For different iterations of the iRRAM, we now can not avoid different flows of control: We have to compute a_p for several different values p depending on the precision bounds $\overline{p}_i$. But due to the construction, the different control flows are restricted to the computation of the limit, which is continuous in its argument x. So it is sufficient for the iRRAM to stop using its multi-value cache as long as it is in the process of computing a limit.

In consequence, the local changes of the flow of control will have no influence on the outer flow of control unless there are side effects. So neither input, output nor any other side effects (e.g. assignments to global variables) are allowed in the algorithm computing $a_p(x)$. Unfortunately, this can not be expressed within the syntax of `C++`.

Computing the limit in the way described above has an important disadvantage: The precision bounds $\overline{p}_j$ were chosen sparse in order to get an optimal behavior of the time complexity of the whole computation. But now this implies that the error of the result $f(x)$ might be very big compared to the error of the argument x: Sometimes it will not be possible to compute $a_{\overline{p}_i}(x)$ successfully using precision bound $\overline{p}_i$, regardless of the actual value of $\overline{p}_i$, simply because in that iteration x already has an error of this size. So it might be that in any iteration, the best achievable result of the limit operator might be $a_{\overline{p_{i-1}}}(x)$ implying a very big loss of precision!

For example, numerical iterations $x_{i+1} = \Phi(x_i)$, where Φ is implemented using this limit operator, might only be usable for small values of i, as we will need at least iteration p_i of the iRRAM to get x_i with an error of p_0.

Some heuristic improvements should be mentioned: If the limit operator is called in the iteration with precision bound $\overline{p_i}$, the values $a_p(x)$ are computed in the modified order $p = \overline{p_i}, \overline{p_0}, \overline{p_1}, \ldots, \overline{p_{i-1}}$ and with a precision bound that is changed to $\overline{p_{i+1}}$ during the computation of the limit. This change in the precision bound increases the probability that $a_{\overline{p_i}}$ can be computed successfully. If this is the case and if the resulting interval for $a_{\overline{p_i}}$ is not larger than $2^{\overline{p_{i-1}}}$, then it is taken as the result of the limit in this iteration (of course increased by $2^{\overline{p_i}}$). This implies that very often it is not necessary to test more than one value.

10.2 Lipschitz Limits

If we have more information on the limit, we can do much better than above. Very often, we know that a (maybe lengthy) computation leads to a quite smooth result. One example is the trigonometric function sine, where we know that $|\sin(x) - \sin(y)| \leq |x - y|$. In this case, we know that it would be possible to approximate $\sin(x)$ by $\sin(d)$ with an error of only e, if x is known to be in the interval $(d \pm e)$. But as a longer computation is necessary to compute sin, any ordinary interval arithmetic on $(d \pm e)$

would yield an interval $(d' \pm e')$ with a much larger error $e' \gg e$. So the central idea is to use interval arithmetic not for $\sin(d \pm e)$, but for $\sin(d \pm \tilde{e})$ for a very small $\tilde{e}$ yielding an interval $(d'' \pm e'')$ and then to use $(d'' \pm (e + e''))$ as a valid interval approximation for $\sin(x)$.

We may generalize this idea as follows: If x is known as an interval $(d \pm e)$ when computing the limit and if we know an upper bound 2^ℓ of a Lipschitz constant for the limit f in the interval $(d \pm e)$, we may simply choose p such that 2^p is slightly bigger than $2^\ell \cdot e$ and compute $a_p(d)$. (A similar concept had already been introduced in [Mu88] as 'locally Lipschitz continuous functions'.) Then $|a_p(d) - f(x)| \leq |a_p(d) - f(d)| + |f(d) - f(x)| \leq 2^p + 2^\ell \cdot e \leq 2^{1+p}$, i.e. $a_p(d)$ differs from the limit $f(x)$ by at most 2^{1+p}.

d can be treated as an exact real value, so we are able to compute $a_m(d)$ with an arbitrary small error. To do this, we might start a new version of the iRRAM just to compute this value. In fact, the iRRAM simply uses local iterations for the computation of $a_p(d)$ with precision bounds $\overline{p_{i+1}}, \overline{p_{i+2}} \ldots$ until one of these iterations also gives an error of less than 2^{1+p}.

These iterations work similar to above: The `longjmp` within the computation of the $a_p(d)$ is redirected temporarily to the procedure implementing `limit_lip`. However, there are two big differences to the implementation of the simple limit operator: For the simple limit we had a fixed bounding precision within the different iterations, but the index p of $a_p(d)$ was changed from iteration to iteration; the Lipschitz limit is computed with a fixed index p of $a_p(d)$, but the bounding precision changes. This also implies that the application of the Lipschitz limit will never lead to a reiteration of the whole computation.

Again, the local iterations do not need a multi-value cache as their result leads to a continuous function. As the local iterations start with precision bound $\overline{p_{i+1}}$ and only need to get result with error larger than $2^{\overline{p_i}}$, it will happen very often that already the first iteration is successful.

The corresponding limit operator takes the form

```
REAL  limit_lip  (REAL a(long, const REAL&),
                  long lip,  const REAL& x);
```

where the variable `lip` corresponds to the logarithm ℓ of the Lipschitz constant 2^ℓ from above.

This operator has two important advantages compared to the simple limit operator introduced before: (1) The growth of the error bounds is reduced to the minimal possible amount and in consequence (2) the computation time is also reduced very often, because smaller error bounds usually lead to fewer iterations.

Most of the functions that are defined in the iRRAM use this concept, e.g. $\exp$, $\sin$ and $\cos$. Here we will show how to implement the maximum function using the Lipschitz limit operator.

To compute the maximum of two numbers using floating point arithmetic, usually a program similar to the following is used:

```
REAL maximum(const REAL& x, const REAL& y)
{ if ( x > y ) return x; else return y;}
```

Obviously, this is not the right solution for the iRRAM, as we are not able to get a result of the comparison if x and y are equal. So the right way is to interpret the maximum as a limit:

```
REAL max_approx (long k, const REAL& x, const REAL& y)
{ if ( positive(x-y,k) ) return x; else return y; };

REAL maximum  (const REAL& x, const REAL& y)
{ return limit_lip(max_approx,0,x,y); };
```

10.3 Multi-valued Limits

In the previous examples, the results of multi-valued functions were from discrete sets (boolean, integer, finite strings etc.) and it was possible to compute these results in finite time. Especially it was always possible to store the actual outcome of a multi-valued function in order to reuse it in a subsequent iteration of the iRRAM. This is no longer possible, if we are considering multi-valued functions with results e.g. from the set of real numbers.

Of course, simple functions of this kind can be constructed using the discrete multi-valued tests, e.g. $\texttt{modulo}(x,y) := x-ky$ where $k \in \mathbb{Z}$ with $(k-1)y \leq x \leq (k+1)y$. This function computes one of the possible values z, $|z| \leq y \neq 0$, such that $\frac{x-z}{y}$ is an integer. It can be used to scale the arguments of periodic trigonometric functions to get efficient implementations:

```
REAL modulo (const REAL& x, const REAL& y){
   return x-round(x/y)*y;
};
```

Here $\texttt{round}(x)$ is a multi-valued integer function with $\mid x - \texttt{round}(x) \mid < 1$. When applying the `modulo` function, the iRRAM would automatically store the result of `round`, which is sufficient to reconstruct the value of `modulo` in possible later reiterations.

A more elaborate example concerns the square root of complex numbers: The square root of a number z is one of the two numbers x or $-x$, where $x^2 = z$, which immediatley leads to multi-valuedness. If $z \in \mathbb{R}_0^+$, it is easy to choose one of the values: usually the positive root is taken.

For arbitrary $z \in \mathbb{C}$, a continuous choice of one of the roots is not possible: Any continuous function $r : \{e^{i\phi} | 0 \leq \phi \leq 2\pi\} \to \mathbb{C}$ on the unit circle satisfying $r(z)^2 = z$ would lead to

$$r(1) = \lim_{\phi \to 0+} r(e^{i\phi}) = - \lim_{\phi \to 2\pi-} r(e^{i\phi}) = -r(1)$$

which is a contradiction.

If we restrict ourself to $z \neq 0$, a multi-valued implementation of the complex square root can be implemented using the simple multi-valued tests: If $z = a + ib \neq 0$, at least one of the real numbers a and b must also be different from 0. Then one of the complex roots can be computed using the ordinary real square root, but the computation may depend on the non-zero value that is determined in a multi-valued way. The only small difficulty is to find this non-zero value without having a test for equality of real numbers:

```
COMPLEX csqrt(const COMPLEX& z)
{
  REAL a,b,c,d,r,s;
  COMPLEX y;
  long choice;

  r= abs (z);  a= real(z);   b= imag(z);

  if ( ! bound(  b, size(r)-3) ) {
    if ( b > 0 )  choice=1;  else  choice=2;
  } else {
    if ( a > 0 )  choice=3;  else  choice=4;
  }

  c= sqrt( (r+a) / 2 );     d= sqrt( (r-a) / 2 );

  switch ( choice ) {
  case 1:    y = COMPLEX ( c      ,   d    ); break;
  case 2:    y = COMPLEX ( c      ,  -d    ); break;
  case 3:    y = COMPLEX ( -c     , -b/2/c ); break;
  case 4:    y = COMPLEX ( b/2/d ,   d    ); break;
  }
  return y;
}
```

Each value of the variable `choice` corresponds to one of the following four open half planes in $\mathbb{C}$: $\{z \mid imag(z) > 0\}$, $\{z \mid imag(z) < 0\}$, $\{z \mid real(z) > 0\}$, and $\{z \mid real(z) < 0\}$. So for any $z \neq 0$, the loop in the program will finally be left after setting an appropriate value for `choice`.

It can easily be verified that the returned value is indeed one of the two roots of z. Figure 3 shows results of this function for the square roots on circles with radius 1, 2, and 3. The results for $b < 0$ and $a < 0$ are adjacent, so only three of the four values for `choice` can be distinguished. Changing the result for $a > 0$ to the more 'natural' `COMPLEX (c,b/2/c)` would have the effect that all results are adjacent (leading to a less interesting figure).

This solution for the square root is not fully satisfying: We use `size` to find a lower bound for the nonzero value from a or b. So for $z = 0$, an infinite loop occurs, and for arguments near to 0, the complexity grows to infinity.

To overcome both problems, the complex square root can be seen as a limit: For arguments z with $|z| > \varepsilon$, let $f(\varepsilon, z)$ be one of the roots of z, otherwise let $f(\varepsilon, z) = 0$ as an approximation. Then simply let $\varepsilon \to 0$.

When trying to implement this within the iRRAM, we must face the problem that for different ε or for different approximations of z, the values $f(\varepsilon, z)$ might switch between the two roots of z. But when computing the limit, we may not switch between the roots, otherwise in different iterations changed flows of control could occur. To prevent this, the arguments for the operator computing multi-valued limits must fulfill additional properties:

```
friend COMPLEX limit_mv (COMPLEX f(long p,
```

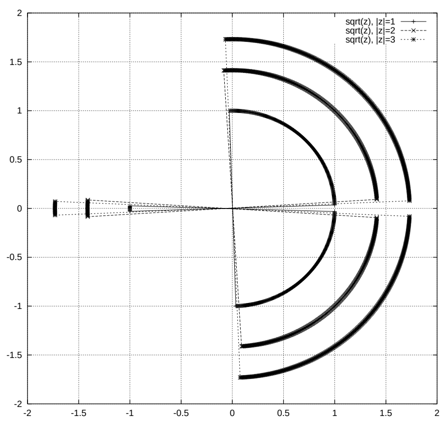

Fig. 3. Values of the complex square root in the iRRAM, using `csqrt`

```
                    long* choice,
                    const COMPLEX& z),
          const REAL& z);
```

The parameter `choice` can be interpreted as follows: It selects a subset of the possible values of the limit that may be computed by `f`. The (initial) value of 0 has the interpretation that all the possible values are still allowed. If `f` changes the value of `choice`, then this must correspond to a reduction of the set of allowed values. At the end of the computation of `f`, the result of `f` must be an approximation (with an error of at most 2^p) to all of the values that still correspond to `choice`.

To compute the limit, the iRRAM proceeds almost as in the case of the usual limit operator, i.e. different values of `p` are checked whether the computation of `f(p,choice,z)` leads to an approximation of the limit. The very first try starts with `choice=0`, i.e. `f` may approximate any of the possible values. For the other tests (in the same or later iterations), the value of `choice` is passed from one computation of `f` to the next using the multi-value cache. So a result of a computation of `f(p,choice,z)` at some stage in the iRRAM must be a refinement of the previous computations at this stage.

In a full theoretical definition of the limit operator, its (multi-valued) argument `f` would be of signature (e.g.) $f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{C} \to \mathbb{C} \times \mathbb{Z}$ where f is supposed to be computable and the second (integer) components of both the argument and the result of f correspond to the values of `choice` before and after the call to `f`. Theoretically, the whole history of the (finite!) computations of `f` could be encoded in this value. However, in practice only a few essential steps of the computation need to be encoded, so choosing type `long` for `choice` should be far reaching, although this might look like a very restricted concept at the first view.

In the case of the complex square root, the value of `choice` will need at most one change: If z is so big compared to the intended precision that 0 is no longer a valid approximation to the roots of z, `choice` is changed to indicate in which of the half planes z is located. This is sufficient to choose one of the roots of z. This choice will not be changed again. A corresponding implementation being able to compute a square root for any complex z including 0 looks as follows:

```
COMPLEX csqrt1(long p, long* choice, const COMPLEX& z)
{ REAL a,b,c,d,r;
  COMPLEX y;

  r= abs (z);  a= real(z);  b= imag(z);

  if ( *choice == 0 ) {
    if ( !bound(r, 2*p) )
      if ( ! bound(b,p-2) ) {
        if ( b > 0 )  *choice=1;  else  *choice=2;
      } else {
        if ( a > 0 )  *choice=3;  else  *choice=4;
      }
    }
  c= sqrt( (r+a) / 2 );  d= sqrt( (r-a) / 2 );

  switch ( *choice ) {
  case 0:     y = COMPLEX  ( 0   ,    0  );  break;
  case 1:     y = COMPLEX  ( c   ,    d  );    break;
  case 2:     y = COMPLEX  ( c   ,   -d  );    break;
  case 3:     y = COMPLEX  ( c , b/2/c );    break;
  case 4:     y = COMPLEX  ( b/2/d, d  );    break;
  }
  return y;
}

COMPLEX  sqrt(COMPLEX z) { return limit_mv(csqrt1,z); }
```

11 Optimizing: Hints and Assertions

In the following we briefly discuss three methods to optimize the efficiency of programs for the iRRAM:

– If a part of a numerical algorithm is known to have modulus of continuity significantly larger than in the rest of the program, e.g. if we try to compute a function with a large first derivative, it may improve the overall complexity significantly if we temporarily use a higher precision which may lead to less reiterations. In this case, the sensitive part of the program can be marked with a pair of functions `stiff_begin()` and `stiff_end()`, where the precision bound is improved by the first function and reset by the second function.
  This technique has been used e.g. in the AGM methods for π and $\log(x)$. Similar effects could be achieved by temporarily switching the influence of the precision bound on the actual computations from absolute to relative precision (which is not implemented yet).

The use of these functions is simply a *hint* to the simulator how to proceed most efficiently. Using these functions might change the flow of control because of the possibility of changing results from multi-valued functions. However, their use can not influence the correctness of a program, only its time complexity.

The following two methods behave different, their use is not only a hint for the simulator but an *assertion* that a part of a program has certain special properties (that can not be verified by the simulator). The simulator will optimize its behavior correspondingly

– The multi-value cache is an expensive simulation tool, as each cached results needs to be stored until the end of the program. This is why the use of multi-valued intrinsics should be reduced to the minimum.

  On the other hand, there are many computations leading to a single-valued continuous results although they use such multi-valued decisions. As an example, consider $\sin(x)$ again. It is well known that using a Taylor series in the computation of $\sin(x)$ for large x is a big waste of time. It is better and quite simple to use the periodicity of this trigonometric function for a range reduction and to compute a y of size $\pi/4$ or less such that either $\sin(x) = \pm\sin(y)$ or $\sin(x) = \pm\cos(y)$. To do this, we have to apply multi-valued tests, as we have to distinguish between 4 distinct cases that later fit together to a result that is single-valued and continuous in x. In consequence, it would not be necessary to cache these tests.

  If such a part of a program is marked with the functions `continuous_begin()` and `continuous_end()`, the caching is temporarily stopped. The programmer must take care himself that it is impossible to enter or leave the section without calling the corresponding function (except maybe for reiterations), as this could easily destroy the consistency of the multi-value cache.

  Another important application is found in linear algebra: If we invert a nonsingular matrix M, the inverse M^{-1} is determined uniquely and the mapping from M to M^-1 is continuous. Here the classical algorithms based on Gaussian elimination internally rearrange the matrices due to the size of pivot elements which requires multi-valued tests that need not to be cached.

– If we know that a sequence of operations has a significantly better modulus of continuity than we achieve through the normal interval arithmetic, we may be able to reduce the growth of the error bounds using a technique similar to the Lipschitz limit operator: Instead of calling a function f with known Lipschitz constant 2^l directly, we may use the form `lipschitz`(f, l, x). Here the precision bound is increased during the computation, and instead of $f(x)$ we compute $f(d)$ for the center d of the interval $(d \pm e)$ representing x. The resulting interval is corrected according to the Lipschitz constant and the size e of the interval. In consequence, the error propagation between x and $f(x)$ is almost reduced to l.

12 Overview: Classes and Functions

In the following we give a brief overview of the new classes, operators, and functions, that are implemented in the iRRAM. Unless explicitly stated, these functions are intrinsic, i.e. their implementations may use private properties of the class `REAL` and so has access to the underlying intervals that are hidden to the ordinary user.

We will not repeat the limit operators here, as they have been explained previously in detail.

Classes

`DYADIC` is simply a wrapper for the underlying multiple precision arithmetic.
The most important class `REAL` is derived from a basic (mostly hidden) class `METRIC_OBJECT` that provides a rudimentary garbage collection necessary for the reiterations and that is the basic class for the limit operators. Further derived classes are `COMPLEX`, `REALMATRIX`, `SPARSEREALMATRIX`, where the latter aims at very large but mostly empty matrices.
All those classes have a destructor ~, a default constructor, and a copy constructor = for assignments.

Real arithmetic

`REAL`s can be constructed from `int`, `long`, `char*`, `double`, `DYADIC`, and from `REAL` itself. In consequence, there are corresponding implicit type conversions, e.g. `x=REAL("3.1415")` can be written as `x="3.1415"`.
The overloaded arithmetic operators ‘`x+y`’, ‘`x-y`’, ‘`-y`’, ‘`x*y`’, and ‘`x/y`’ can be used in a naive way, but a division by `REAL(0)` leads to an infinite loop.
The usual binary tests ‘`x<y`’, ‘`x<=y`’,‘`x>y`’, and ‘`x>=y`’ exist, but there is no test on equality or inequality of reals! These tests will lead to infinite loops if `x` and `y` are equal. So ‘`x<y`’ and ‘`x<=y`’ really are the same function!
The multi-valued functions ‘`size(x)`’, ‘`positive(x,p)`’, ‘`bound(x,p)`’, ‘`approx(x,p)`’, and ‘`round(x)`’ have been explained previously. As `round(x)` delivers a result of type `long` (which is very restricted in size), there is a corresponding function `round2(x)` with a result of type `REAL`.
Quite often, multiplication with a (positive or negative) power of 2 is necessary, which is implemented as an intrinsic function `scale(x,k)`.
Finally there is a library of mathematical functions that are not intrinsic (i.e. they are defined essentially ‘high level’ but usually with use of the limit operators). These function are still very efficient, as the necessary overhead is small.
There is a small set of algebraic functions: `abs(x)`, `power(x,n)` `sqrt(x)`, `root(x,n)`, `modulo(x,y)`, `maximum(x,y)`,and `minimum(x,y)`, furthermore there are transcendental functions like `exp(x)`, `log(x)`, and a full set of trigonometric functions and their inverses, from `sin(x)` to `acosech(x)`. In addition, the special numbers `pi()`$= \pi = 3.1415...$, `ln2()`$= \log(2) = 0.6931...$ and `euler()`$= e = 2.1718...$ are defined (necessarily as functions).

Matrix arithmetic

The classes `REALMATRX` and `SPAREREALMATRIX` deal with matrices of real components. As they are derived form the class `METRIC_OBJECT`, limit operators can be applied. The corresponding algorithms are not intrinsic, as they all are derived from real arithmetic! The most important constructors are `REALMATRIX(r,c)` and `SPARSEREALMATRIX(r,c)` for a (sparse) matrix of `r` rows and `c` columns, as well as `REALMATRIX(m)` and `SPARSEREALMATRIX(m)` to get a copy of a matrix `m`. To access matrix elements, the parentheses `()` have been overloaded, allowing assignments like `m(1,2)="3.1414"`.

Basic matrix arithmetic 'm1+m2', 'm1-m2', 'm1*m2', 'm1/m2' is implemented, where the latter computes a solution of the linear system $M_2 \cdot X = M_1$ by Gaussian elimination (if M_2 is not singular). Multiplication m*x and division m/x by scalar values x are defined. In addition, eye(n), zeroes(r,c) or ones(r,c) deliver the unit matrix, or a matrix full of zeroes or ones.
The matrix exponential $\mathtt{exp}(m) := \sum m^k/k!$ is an example for the implementation of a matrix transcendental.

Complex arithmetic
COMPLEX numbers are a further example for the possibility of user-defined data types. Only a few functions have been implemented: basic arithmetic, abs(z), real(z), imag(z), as well as the complex square root that has been the central example for a multi-valued limit in section 10.

Input and Output
The special role of I/O and the functions rscanf, rprintf, rshow(x,w) and rwrite(x,p,w) has already been mentioned in section 8.

Special Functions and Operators
The special functions continuous_begin(), continuous_end(), that modify the caching strategy, as well as stiff_begin(), stiff_end(), and the operator lipschitz(f,l,x), that modify the error propagation, have been discussed in the section 11.

13 Importing special MP functions

One of the aims of the iRRAM is to get an abstract level of computation independent from the underlying MP package. This would imply that only the intersection of all the functions of the different MP packages could be used. To get access to special functions of the MP packages, the interface to the MP can be extended.

This possibility has been used for the multiple precision square root in order to get an implementation of sqrt that is much faster than the version shown in section 10: The interface to the backend of the iRRAM for Longreal with low-level GMP routines (which can be found is LRGMP_interface.h) contains the lines

```
#define MP_has_sqrt 1
#define MP_sqrt(z1,z,n) Dyadic_LRSQRT(&(z1),&(z),n)
```

In REALLIB.cc, where most of the higher level mathematical functions are implemented, there is a corresponding #ifdef MP_has_sqrt that switches at compile time between a high level version from section 10 and a faster intrinsic version using MP_sqrt.

14 The iRRAM and IEEE 754

Although the iRRAM implements exact arithmetic, so that program should run forever, it can be used as an extensible library of functions with variable precision as long as not input or output of real numbers is used: If the programmer defines an own version of main, he is able to call iRRAM_exec(compute) several times within his program, even with several different functions instead compute(). As the argument of iRRAM_exec is of type void (), compute may neither have arguments not return a result. So the only possibility to pass data between main and compute is to use global variables. If these variables are of type DYADIC, we essentially get a new function extending the multiple precision backend. For example, the cosine function (that is e.g. still missing in GMP) could be implemented similar to the following way:

```
DYADIC glob_arg,glob_result;
long glob_precision;

void compute_sin() {
  glob_result=approx(cos(REAL(glob_arg)),glob_precision);
}

DYADIC cos(DYADIC x, long p) {
  glob_arg=x;
  glob_precision=p;
  iRRAM_exec(compute_cos);
  return glob_result;
}
```

Here we use three global variables: for the argument, the result and the precision of the computation. Please remember: we are implementing a finite precision version of cosine using our exact real version, so we have to specify the precision of the result!

Current multiple precision packages like [Zi00] try to smoothly extend the current hardware floating point arithmetic and try to stick to the principles of IEEE 754 floating point arithmetic. This implies that certain shortcomings of earlier hardware like non-monotonic behavior for monotonic functions should be avoided. One similar aspect are exact rounding modes, i.e. it should be possible to specify that the result of MP operations is either the next, the greatest smaller or the smallest larger MP number compared to the exact result.

From the view of exact real arithmetic, exact rounding modes are similar to testing whether two reals are equal: Nontrivial single-valued real functions with a discrete set of possible values are necessarily discontinuous and hence not computable! On the other hand, any correct implementation of a theoretically monotonic function using exact arithmetic will be monotonic, and any 'irregular' non-monotonic behavior of the conversion to discrete sets can be completely explained using multi-valued functions, so exact rounding is not an issue in exact arithmetic.

It would even be possible to create MP routines with exact rounding for functions f using the iRRAM as backend, as long as the exact values $f(d)$ for MP numbers d are not exactly representable as MP numbers again. E.g. for $\cos(d)$ with $d \neq 0$, we could simply iterate the procedure above with different values for glob_precision until we get enough *different* bits to allow the exact rounding. It should be sufficient almost always to choose an initial bounding precision just smaller than necessary for the expected value to decide the rounding without any further iterations. Rare exceptions with higher several iterations could surely be tolerated.

15 Exploring the Borders

In the following we present a few examples for the time complexity of the iRRAM. All computations have been performed using an AMD Athlon 800 MHz (512 KD cache, slot A) on an ASUS mainboard K7M equipped with 256 MB 100 MHz SDRAM. The backend was LRGMP using GMP 3.1, operating system was Linux with kernel 2.4.0-test4. Programs had been compiled with `gcc`, version 2.95.2.

– **Selected values and functions**
  Compute n decimals of $\sqrt{1/3}$, π, $\log(1/3)$, and $\sin(1/3)$:

Number n of Decimals	10	100	1000	10000	100000
Time for conversion	8 μs	37 μs	39 μs	14 ms	1.5 s
Time for π	-	-	-	140 ms	9.1 s
Time for inversion	3.4 μs	6.0 μs	80 μs	2.9 ms	140 ms
Time for $\sqrt{1/3}$	17 μs	30 μs	132 μs	4.7 ms	360 ms
Time for $\log(1/3)$	290 μs	740 μs	6 ms	240 ms	22.9 s
Time for $\sin(1/3)$	71 μs	210 μs	3.5 ms	450 ms	157 s

  As the internal format of the multiple precision backend is in binary form, we have to split the computation time is two parts: he computations, a part of the time was taken for the conversion from the internal binary format to the the decimal form, which is not optimized yet and has complexity of order $\mathcal{O}(n^2)$. The other given times are do not include this conversion time, so it has to be added to the other given values in order to get the full time for computation with the desired precision including the output.
  π is computed using a quartically convergent algorithm due to the Borwein brothers [Ba88,Bo87]. Due to an internal optimization, we were not able to measure the time necessary for the computation of π for small n.
  The square root $\sqrt{\ }$ is implemented with a variant of Heron's quadratic convergent method, but computed with variable precision as in [Bt76].
  The implementation of the natural logarithm $\log$ uses an AGM method based on ideas from [Sch90]. It needs good precomputed approximations for π and $\log(2)$. The measured times are without these precomputations, so they are valid for all but the first evaluation of $\log$.
  $\sin$ uses a Taylor series approach after an appropriate range reduction that also uses a precomputed π. The given times are without this precomputation.
– **Iterated function systems**
  Compute the iterates x_i of the so called logistic function $x_i = f(x_{i-1}) = 3.75 \cdot x_{i-1} \cdot (1-x_{-1})$, where we start with $x_0 = 0.5$ and print the first 6 decimal places of each x_i:

Index i	1000	5000	10000	50000	100000
Value x_i	+.791747	+.814694	+.824205	+.283081	+.666947
Max. Bounding Precision	-2166	-10888	-21407	-102635	-200601
Execution Time	60 ms	1.56 s	8 s	385 s	2054 s

Using `double` instead of `REAL`, beginning from x_{80} the 6th digit is incorrect, x_{90} has only one correct digit left, and finally from x_{100}, all digits are wrong:

	`REAL`	`double`
x_{60}	+.7990863343	0.79908633$\boxed{70}$
x_{70}	+.4521952998	0.4521952$\boxed{586}$
x_{80}	+.8561779966	0.85617$\boxed{59906}$
x_{90}	+.7399137486	0.7$\boxed{400517104}$
x_{100}	+.8882939922	0.$\boxed{9017659679}$
x_{110}	+.7156795292	0.$\boxed{2201217854}$

This example has been taken from [Ku96], where a similar table was shown up to x_{450} using the software package C-XSC [Kl93]. Unfortunately, this package is quite restricted in the mantissa length as well as the exponent range, such that it is impossible to compute x_{500} using the package.

– **Matrix arithmetic**
Compute an approximation (with maximal error 2^{-50}) of the inverse of the (bad conditioned) Hilbert matrix H_n of size $n \times n$ using the built-in matrix arithmetic (i.e. with Gaussian elimination):

Dimension n	50	100	150	200	250
Max. bounding precision	-1037	-2745	-4372	-5502	-6915
Used Memory	1.46MB	9.2MB	40 MB	81MB	152 MB
Execution Time	3.2 s	79 s	457 s	1200 s	3052 s

Compute an approximation (with maximal error 2^{-50}) to the inverse of the well conditioned matrix $H_n + \mathbb{1}_n$ of size $n \times n$:

Dimension n	50	100	150	200	250	500
Max. bounding precision	-100	-100	-100	-100	-100	-162
Used Memory	0.5 MB	2.1 MB	4.7 MB	8.4 MB	13 MB	52MB
Execution Time	0.7 s	5.4 s	19 s	45s	91 s	1237 s

Obviously, the condition of the matrix has big influence on the necessary bounding precision, which explains the big differences in execution time and memory consumption between the two examples.
The results should be compared to the results of similar computations using e.g. `octave` (a freely available high-level interactive language for numerical computations), where optimized routines working with the limited precision of the hardware are used (without error control):
Octave is already unable to invert the Hilbert matrix of size 12 (giving a warning `matrix singular to machine precision`). ON the other hand, the inversion of the well-conditioned matrix $H_n + \mathbb{1}$ with $n = 500$ takes only 12 s, so the iRRAM is about a factor of 65 slower. This factor is different for other types of CPUs, e.g. for an AMD K6-200 with its weaker floating point performance we get a factor of 35. Please have in mind that here we essentially have to compare a software solution with hardware (on the same CPU)!

16 Future Work

The current state of the iRRAM allows reliable and efficient exact arithmetic. But of course, many enhancements and optimizations can be imagined. In the following we list just a few of them:

- Until know, the iRRAM has only been tested using the GNU `g++` compiler and under Linux. A port to other popular systems should be easy.
- There should be a better garbage collection that clears all temporary objects whenever a reiteration happens. At the moment, there is only a restricted garbage collector that is able to recover all memory occupied by objects derived from `METRIC_OBJECT`. However, any memory allocated on the stack using `alloca` is freed automatically, as the reiterations are implemented using `longjmp`. On the other hand, big applications will use almost all their memory for real variables.
- The set of `DYADIC` functions should be extended using the techniques described in this paper, also `COMPLEX` versions of the transcendental and trigonometric functions would be important.
- The algorithms for `exp` and for the trigonometric functions should be further improved using AGM methods, see e.g. [Bt76].
- New data types for (full range) integers or rationals should be added.

References

[Ba88] D.H. Bailey, The computation of π to 29,360,000 Decimal Digits Using Borweins Quartically Convergent Algorithm *Mathematics of Computation* Vol. **50** Number 181 (1988) 238-296

[Bo87] J.M. Borwein & P.B. Borwein, Pi and the AGM, A study in analytic number theory, Wiley, New York, 1987

[BoCa90] H. Boehm & R. Cartwright, Exact Real Arithmetic: Formulating real numbers as functions. In T. D., editor, *Research Topics in Functional Programming,* 43-64 (Addison-Wesley, 1990)

[BSS89] L. Blum & M. Shub & S. Smale, On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines, *Bulletin of the AMS* **21**, 1, July 1989

[Bt75] R.P. Brent, The complexity of multiple precision arithmetic, Proc. Seminar on Complexity of Computational Problem Solving, Queensland U. Press, Brisbane, Australia (1975) 126-165

[Bt76] R.P. Brent, Fast multiple precision evaluation of elementary functions, *J. ACM* **23** (1976) 242-251

[Bt78] R.P. Brent, A Fortran multiple precision package, *ACM Trans. Math. Software* **4** (1978), pp 57-70

[CoAa89] Cook, S.A. und Aanderaa, S.O., On the minimum computation time of functions, *Trans. Amer. Math. Soc.* **142** (1969) 291-314

[FiSt74] Fischer, M.J. und Stockmeyer, L.J. Fast on-line integer multiplication, *J. Comput. System Scis.* **9** (1974) 317-331

[Br96] V. Brattka, Recursive characterisation of computable real-valued functions and relations, Theoret. Comput. Sci. **162** (1996),47-77

[BrHe95] V. Brattka & P. Hertling, Feasible Real Random Access Machines, *Informatik Berichte 193 - 12/1995, FernUniversität Hagen*,

[BrHe94] V. Brattka & P. Hertling, Continuity and Computability of Relations, *Informatik Berichte 164 - 9/1994, FernUniversität Hagen*,

[Br99] V. Brattka, Recursive and Computable Operations over Topological Structures, Thesis, *Informatik Berichte 255 - 7/1999, FernUniversität Hagen*

[EdPo97] A. Edalat & P. Potts, A new representation for exact real numbers, *Proc. of Mathematical Foundations of Programming Semantics 13*, Electronic notes in Theoretical Computer Science 6, Elsevier Science B.V., 1997, URL: `www.elsevier.nl/locate/entcs/volume6.html`

[Gr00] T. Granlund, GMP 3.1, `http://www.swox.com/gmp/`

[GL00] P. Gowland, D. Lester, The Correctness of an Implementation of Exact Arithmetic, *4th Conference on Real Numbers and Computers, 2000, Dagstuhl, 125-140*

[HM00] S. Heinrich & E. Novak et al., The Inverse of the Star-Discrepancy depends linearly on the Dimension, *Acta Arithmetica, to appear*

[Kl93] R. Klatte & U. Kulisch et al., C-XSC , a C++ Class Library for Extended Scientific Computing (Springer, Berlin 1993)

[Ko91] K. Ko, Complexity Theory of Real Functions, (Birkhäuser, Boston 1991)

[Ku96] U. Kulisch, Memorandum über Computer, Arithmetik und Numerik (Universität Karlsruhe, Institut für angewandte Mathematik)

[Mm96] V. Ménissier-Morain, Arbitrary precision real arithmetic: design and algorithms, *J. Symbolic Computation, 1996,* **11**

[Mu88] Müller, N.Th., Untersuchungen zur Komplexität reeller Funktionen, *Dissertation* (FernUniversität Hagen, 1988)

[Mu93] N.Th. Müller, Polynomial Time Computation of Taylor Series, *Proc. 22 JAIIO - PANEL '93, Part 2, Buenos Aires, 1993,* 259-281
(also available at `http://www.informatik.uni-trier.de/~mueller`)

[Mu96] Müller, N.Th., Towards a real Real RAM: a Prototype using `C++`, (preliminary version), *Second Workshop on Constructivity and Complexity in Analysis, Forschungsbericht Mathematik-Informatik, Universität Trier 96-44*, Seiten 59-66 (1996) (also available at `http://www.informatik.uni-trier.de/~mueller`)

[Mu97] N.Th. Müller, Towards a real RealRAM: a Prototype using `C++`, *Proc. 6th International Conference on Numerical Analysis*, *Plovdiv, 1997*

[Mu98] N.Th. Müller, Implementing limits in an interactive RealRAM, *3rd Conference on Real Numbers and Computers, 1998, Paris,* 13-26

[Mu00] N.Th. Müller, A Note on how to compute Multi-Valued Functions *draft, in preparation*principles of IEEE 754 floating point arithmetic.

[Rio94] M. Riordan, `ftp://ripem.msu.edu/pub/bignum/BIGNUMS.TXT`

[Sch90] A. Schönhage, Numerik analytischer Funktionen und Komplexität, *Jber. d. Dt. Math.-Verein.* 92 (1990) 1-20

[TZ99] J.V. Tucker, J.I. Zucker, Computation by 'While' programs on topological partial algebras, *Theoretical Computer Science* 219 (1999) 379-420

[We87] K. Weihrauch, Computability (volume 9 of: *EATCS Monographs on Theoretical Computer Science*), (Springer, Berlin, 1987)

[We95] K. Weihrauch, A Simple Introduction to Computable Analysis, *Informatik Berichte 171 - 2/1995, FernUniversität Hagen*

[We97] Weihrauch, K., A Foundation for Computable Analysis, *Proc. DMTCS´96*, (Springer, Singapore, 1997) 66-89

[Zi00] P. Zimmermann, MPFR: A Library for Multiprecision Floating-Point Arithmetic with Exact Rounding, *4th Conference on Real Numbers and Computers, 2000, Dagstuhl, 89-90*, see also `http://www.loria.fr/projets/mpfr/`