Efficient Implementation of Exact Real Numbers

CCA’2005, Kyoto

Norbert Th. Müller
Abteilung Informatik — Universität Trier
D-54286 Trier, Germany
1. Introduction
2. Basic Examples
3. Small comparison of packages
4. Basic Implementational Internals
5. Usage and Examples
6. Concluding Remarks
Background: *Computable Real Analysis*

\[\text{TTE} \ (\text{Weihrauch, Brattka, Hertling, Ko, M., ...}) \quad \Leftrightarrow \quad \text{domains} \ (\text{Blanck, Edalat, Escardo, Tucker, ...}) \quad \not\Leftrightarrow \quad \text{BSS-model of Real RAM (Blum, Shub, Smale, ...)}\]

exact real arithmetic in accordance with TTE:

— functional or imperative(!) programming
— *atomic objects* \(x \in \mathbb{R}\)
— fully(!) consistent with real calculus
Computable analysis:

Remember: Computable functions are continuous!
Example 1: iteration with 3 fixed points (J.M. Muller, 1989)

\[ x_0 = \frac{11}{2}, \quad x_1 = \frac{61}{11}, \quad x_{n+1} = 111 - \frac{1130 - \frac{3000}{X_{n-1}}}{X_n} \]
Example 1: iteration with 3 fixed points (J.M. Muller, 1989)

\[ x_0 = \frac{11}{2}, \quad x_1 = \frac{61}{11}, \quad x_{n+1} = 111 - \frac{1130 - \frac{3000}{x_{n-1}}}{x_n} \]

```cpp
void jm_muller(int count) {
    REAL a = REAL(11)/2;
    REAL b = REAL(61)/11;
    REAL c;

    for (long i=0; i<count; i++) {
        cout << a << " " << i << endl;
        c = 111 - (1130 - 3000/a)/b;
        a = b; b = c;
    }
}
```
Example 2: logistic sequence (Kulisch)

\[ x_0 = \frac{1}{2}, \quad x_{n+1} = 3.75 \cdot x_n \cdot (1 - x_n) \]
Example 3: harmonic series

\[ x_n = \sum_{i=1}^{n} \frac{1}{i} \]

Exact Values Using Double
Fixed size arithmetic:

32bit/64bit-int: canonical semantics (mod $2^{32}$, $2^{64}$)
  Hardware

float, double: standardized semantics, IEEE 754/854
  Hardware

Variable size arithmetic, e.g.:

Integer / Rational, e.g.: canonical semantics
  GMP (C, Granlund, et.al)
  numerix (C, ocaml, PASCAL, Quercia)

Multiple Precision Floats, e.g.:
  MP (Fortran 77, R.P.Brent)
  GMP (C, Granlund, et al.)
  MPFR (C, Zimmermann, Lefèvre, et al.)
    (variable mantissa, 32-bit-exponent)
    semantics following IEEE 754/854
until here ↑: equality $x \equiv y$ easily decidable!

**Algebraic numbers:**  
- Leda (Mehlhorn, Schirra, Näher,...)  
- Core library (Yap,...)

until here ↑: equality $x \equiv y$ decidable (in principle) using computation diagrams + separation bounds, but hard
Interval arithmetic with fixed size, eg.: 

filib++ (Wolff von Gudenberg, Hofschuster, Kraemer, ... )
fixed precision interval arithmetic library
C++ software

Interval arithmetic with variable size, eg.: 

MPFI (Revol, Rouillier)
multiple precision interval arithmetic library
C software based on MPFR
(almost) exact arithmetic:

precise computation software (Aberth, 1998)
‘range’ arithmetic
numbers are represented as intervals with ‘small’ diameter
floating point software, base = 10000
C++ software (but no gcc 3.x...)

extended with calculus on elementary functions
( +, −, *, /, \(x^y\), sin, cos, tan, asin, acos, atan, \(e^x\), \(\ln x\), max, min )

prepared for user-driven iterations
computations may fail...
Example: **Logistic function** with ‘range’ arithmetic

```c++
#include "real.h"

int main() {
    cin >> param;

    cin >> precision; set_precision(precision);

    real x = 1 / real(2); real c = 375 / real(100);

    for (i = 1; i <= param; i++) {
        x = c * x * (one - x);
        if (i % 10 == 0)
            cout << i << " " " " << x.str(2, 20) << endl;
    }
}
```
Further packages:
Mathematica, Matlab, MuPad, Magma, Pari, Arithmetic Explorer...

essentially offering mixtures of methods from above..

(older) comparison (Zimmermann, msec/operation on Xeon 400MHz):

<table>
<thead>
<tr>
<th></th>
<th>Maple V.5.1</th>
<th>MuPAD 1.4.2</th>
<th>MPF 3.1</th>
<th>MPFR 2001</th>
<th>Mathematica 4.0.1</th>
<th>Pari 2.0.14 alpha</th>
<th>Magma V2.7-2</th>
<th>Ar. Explorer 1.00</th>
<th>iRRAM 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicat.</td>
<td>1.1</td>
<td>0.48</td>
<td>0.142</td>
<td>0.145</td>
<td>0.37</td>
<td>0.18</td>
<td>1.089</td>
<td>0.51</td>
<td>0.17</td>
</tr>
<tr>
<td>division</td>
<td>11.3</td>
<td>0.56</td>
<td>0.229</td>
<td>0.228</td>
<td>1.12</td>
<td>0.25</td>
<td>1.301</td>
<td>0.40</td>
<td>0.287</td>
</tr>
<tr>
<td>square root</td>
<td>15.0</td>
<td>1.5</td>
<td>0.355</td>
<td>0.177</td>
<td>1.6</td>
<td>0.70</td>
<td>3.41</td>
<td>1.3</td>
<td>0.364</td>
</tr>
<tr>
<td>exponential</td>
<td>53</td>
<td>35</td>
<td>na</td>
<td>6.24</td>
<td>35</td>
<td>14</td>
<td>78.51</td>
<td>na</td>
<td>21.1</td>
</tr>
<tr>
<td>logarithm</td>
<td>69</td>
<td>52</td>
<td>na</td>
<td>7.76</td>
<td>31</td>
<td>23</td>
<td>118.9</td>
<td>na</td>
<td>18.8</td>
</tr>
</tbody>
</table>
Basic concepts of exact real arithmetic:

- Real numbers are atomic objects
- Arithmetic is able to deal with arbitrary real numbers ...
- ... but usual entrance to $\mathbb{R}$ is $\mathbb{Q}$
- Underlying theory: TTE, Type-2-Theory of Effectivity ...
- ... implying: computable functions are continuous!
- ... implying: failing tests $x \leq y$, $x \geq y$, $x = y$ in case of $x = y$!
- ... using multi-valued functions
Basic methods in exact real arithmetic:

- **constructive methods:**
  - data structures behind REAL variables ...
  - ... exactly represent the exact values
  - **explicit computation diagrams**
    - lazy evaluation,
    - usually top-down-evaluation,
    - often using concepts from functional languages

- **approximative methods**
  - data structures behind REAL variables ...
  - ... represent only approximations
  - **implicit computation diagrams**
    - reconstructable in iterations,
    - usually bottom-up-evaluation,
    - ‘decision history’, ‘multi-value-cache’
Packages for Exact Real Arithmetic:

- **CRCalc** (Constructive Reals Calculator, Boehm)
- **XR** (eXact Real arithmetic, Briggs)
- **IC Reals** (Imperial College: Errington, Krznaric, Heckmann et al.)
- **precise computation software** (Aberth)
- **RealLib** (Lambov)
- **iRRAM** (M.)
CRCalc (Constructive Reals Calculator, Boehm)

- **JAVA** implementation of constructive reals
- explicit computation diagrams with OO methods
- top-down evaluation based on scaled `BigInteger`

Excerpt of addition algorithm:

```java
class add_CR extends CR {
    CR op1;
    CR op2;
    add_CR(CR x, CR y) {
        op1 = x;
        op2 = y;
    }

    protected BigInteger approximate(int p) {
        return scale(op1.get_appr(p-2).add(op2.get_appr(p-2)), -2);
    }
}
```
sample program:

```java
CR one = CR.valueOf(1);
CR C = CR.valueOf(375).divide(CR.valueOf(100));
CR X = one.divide(CR.valueOf(2));

for (int i=1;i<param;i++){
    X = C.multiply(X).multiply(one.subtract(X));
    if (i%10==0) {
        System.out.print(i); System.out.print(" ");
        System.out.println(X.toString(20));
    }
}
```
XR (eXact Real arithmetic, Briggs)

- $x \sim \lambda i.a_i$ for $x = \lim_{i \to \infty} a_i \cdot 2^{-i}$, $a_i \in \mathbb{Z}$
- python or C++ (with functional extension FC++):

Excerpt of XR:

```cpp
typedef Fun1<int,Z> lambda;
...

class XR: public XRsig {
  public:
    lambda x;
    Z operator() (const int n) const { return x(n); }
...

class AddHelper: public XRsig {
  XR f; XR g;
  AddHelper(const XR& ff, const XR& gg):
    f(ff), g(gg) {}  
  Z operator() (const int n) const {
    ... return (f(n+2)+g(n+2)+2) » 2; ... 
  }
}```
IC Reals (Imperial College: Errington, Krznaric, Heckmann et al.)

- language: C
- linear fractional transformations using GMP big integer
- generalization of continued fractions
- lazy evaluation, top-down, lazy boolean predicates (multi-valued)

```c
add_R_R(Real x, Real y)
{
    return tensor_Int(x, y, 0, 0, 1, 0, 1, 0, 0, 1);
}

Real tensor_Int(Real x, Real y, int a, int b, int c, int d, int e, int f, int g, int h)
{
    ...
    tenXY->x = x;
    tenXY->y = y;
    ...
}
```
precise computation software (Aberth)

- ‘range’ arithmetic
- numbers are represented as intervals with ‘small’ diameter
- computations may fail, prepared for iterations

RealLib (Lambov)

- computation diagrams
- language: C++
- simplified intervals, single-valued limits, bottom-up evaluation

iRRAM (M.)

- iRRAM = iterative Real RAM
- language: C++
- iterative concept
- simplified intervals, limits, bottom-up evaluation, ...
**Benchmark 1:** Logistic sequence

\[ x_0 = \frac{1}{2}, \quad x_{n+1} = 3.75 \cdot x_n \cdot (1 - x_n) \]

Print 10 decimals of \( x_n \) for moderate \( n \)

⇒

moderately nested operations, needing high precision

**Benchmark 2:** Compute part of the harmonic series

\[ h(n) = \sum_{i=1}^{n} \frac{1}{i} \]

Print 10 decimals of \( h(n) \) for large \( n \)

⇒

deeply nested operations, but: not(!) hard concerning precision
<table>
<thead>
<tr>
<th>package</th>
<th>logistic sequence</th>
<th>harmonic series</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n=1,000</td>
<td>n=10,000</td>
</tr>
<tr>
<td>CRCalc</td>
<td>1359 sec</td>
<td>&gt;1h</td>
</tr>
<tr>
<td>XR2.0</td>
<td>423 sec</td>
<td>&gt;1h</td>
</tr>
<tr>
<td>IC-Reals</td>
<td>1600 sec</td>
<td>&gt;1h</td>
</tr>
<tr>
<td>Aberth</td>
<td>0.5 sec</td>
<td>1468 sec</td>
</tr>
<tr>
<td>RealLib</td>
<td>0.4 sec</td>
<td>85.9 sec</td>
</tr>
<tr>
<td>iRRAM</td>
<td>&lt;0.1 sec</td>
<td>8.6 sec</td>
</tr>
</tbody>
</table>

(computer: P3-1200, Linux, gcc 2.x/3.x)

compare case n=5,000,000 with ordinary arithmetic:

- unverified double: 0.18 sec
- filib++: 1.1 sec
- iRRAM: 10.6 sec

Using just e.g. 7 decimals for case n=5,000,000:

⇒ 64-bit floating point hardware instead of software possible

- RealLib v.3: 0.45 sec
- iRRAM v.2005_02: 0.70 sec
here: **internals** of the iRRAM

but also applicable for the *bottom-up evaluation* of computation diagrams

**basic idea:**
iterate finite approximations (=intervals), with increasing precision

First: example program with basic properties...
#include "iRRAM.h"

// Compute an approximation to e=2.71...
REAL e_approx (int p)
{
    if ( p >= 2 ) return 0;

    REAL y=1,z=2;
    int i=2;
    while ( !bound(y,p-1) ) {
        y=y/i;
        z=z+y;
        i+=1;
    }
    return z;
}

// Compute the exact value of e=2.71...
REAL e(){ return limit(e_approx); };
void compute(){

    REAL euler_number=e();

    int deci_places;

    cout << "Desired Decimals: ";
    cin » deci_places;

    do {
        cout « setRwidth(deci_places+8) « euler_number;

        cout « endl « " Another try? ";
        cin » deci_places;

    } while ( deci_places > 0 );
}
REAL e_approx (int p)
{
    if ( p >= 2 ) return 0; //

    REAL y=1, z=2;
    int i=2; //
    while ( !bound(y, p-1) ) { //
        y=y/i;
        z=z+y;
        i+=1;
    } //
    return z;
};
```c
REAL e_approx (int p)
{
    if ( p >= 2 ) return 0; • ⇐

    REAL y=1, z=2;
    int i=2; • ⇐
    while ( !bound(y, p-1) ) { • ⇐
        y = y/i;
        z = z + y;
        i += 1;
    } • ⇐
    return z;
}
```

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>−3</td>
<td>2</td>
<td>1.00 ± 0.1</td>
<td>2.00 ± 0.1</td>
</tr>
<tr>
<td>−3</td>
<td>2</td>
<td>1.00 ± 0.1</td>
<td>2.00 ± 0.1</td>
</tr>
<tr>
<td>−3</td>
<td>3</td>
<td>0.50 ± 0.2</td>
<td>2.50 ± 0.3</td>
</tr>
<tr>
<td>−3</td>
<td>3</td>
<td>0.50 ± 0.2</td>
<td>2.50 ± 0.3</td>
</tr>
<tr>
<td>−3</td>
<td>4</td>
<td>0.16 ± 0.3</td>
<td>2.66 ± 0.6</td>
</tr>
<tr>
<td>−3</td>
<td>4</td>
<td>0.16 ± 0.3</td>
<td>2.66 ± 0.6</td>
</tr>
</tbody>
</table>

Actual Computation

T

T

?
Simple(?) Solution:

Repeat the computation with improved precision!

Technical Problems: How to...
...repeat?
...improve precision?
...improve computation time?
...implement intervals efficiently?

Theoretical Problems:
Which operations?
Which operators?
Discontinuous functions?!
Internal representation of $\textbf{REAL } x$ mainly as

$$(d-e, d+e)$$

for multiple precision number $d$ and error information $e$

possible choices for $e$:

- $e = 2^p$ for $p \in \mathbb{Z}$
  - (internally: 32-bit-int)
  - precision of $p$ bits
  - usually: one bit lost per operation, bad for nested operations!

- $e = z \cdot 2^p$ for $z \in \mathbb{N}, p \in \mathbb{Z}$
  - (internally: MP number)
  - ordinary interval arithmetic
  - doubled memory, at least doubled time

- $e = z \cdot 2^p$ for $z < 2^{gb}, |p| < 2^{\text{maxexp}}$ (internally: two 32-bit-int)
  - acceptable error propagation, constant overhead
  - simplified intervals, choosen for iRRAM
Iterations happen e.g. in case of:

- divisions,
  if diameter of dividing interval not much smaller than its midpoint
- conversions, printing,
  if the interval is too large for the necessary precision
- comparisons like $x < y$,
  if intervals for $x$ and $y$ have non-empty intersection

In consequence, infinite loops occur in case of:

- division by zero
- comparison of equal numbers

implementation: C++-exceptions
⇒ destructors are called
⇒ pointers can lead to memory leaks
⇒ improper use of static variables is dangerous!
\( c = a \circ b \) on \textit{REAL} is simulated by intervals:

\[
a \sim (d_a \pm e_a), \quad b \sim (d_b \pm e_b) \quad \Rightarrow \quad c \sim (d_c \pm e_c)
\]

- \( d_c \) is computed from \( d_a, d_b \), with (absolute) precision \( 2^p \)
- minimal possible error \( e'_c \) depends on \( e_a, e_b, d_a, d_b \)
- \( e_c \) is computed as \( e'_c + 2^p \)
- \( p \) could be chosen arbitrarily!
- usually \( 2^p \approx e'_c \cdot 2^{-20} \)

\( \Rightarrow \) the precision of each single operation is computed dynamically:
  - reasonably precise for proper error propagation
  - but not more, for fast computation of \( d_c \)
But: where to cut e.g. $1/3 = 0.33333333\ldots$?
$e_c$ should not be too precise!
otherwise: problems similar to rational arithmetic could occur!

So:

- Choose a precision bound $\overline{p}$ depending on the iteration!

- **precision_policy**(ABSOLUTE)
  Choose optimal $p$ with $p > \overline{p}$
  Application: almost always, default policy

- **precision_policy**(RELATIVE)
  Choose optimal $p$ with $p > \overline{p} + |d_c|$
  Application: e.g. AGM iteration
Sequence of precision bounds, for each iteration:

$$\bar{p}_0 \gg \bar{p}_1 \gg \bar{p}_2 \ldots$$

$\bar{p}_{i+1}$ instead of $\bar{p}_i$ usually leads to smaller intervals.

Implemented version:

$$\bar{p}_{i+1} = \bar{p}_i + \alpha \cdot \beta^i$$

with $\bar{p}_0 = \alpha = -50$ and $\beta = 1.25$

<table>
<thead>
<tr>
<th>$\bar{p}_0$</th>
<th>$\bar{p}_1$</th>
<th>$\bar{p}_2$</th>
<th>$\bar{p}_3$</th>
<th>..</th>
<th>$\bar{p}_{10}$</th>
<th>$\bar{p}_{15}$</th>
<th>$\bar{p}_{20}$</th>
<th>$\bar{p}_{25}$</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>-50</td>
<td>-100</td>
<td>-162</td>
<td>-240</td>
<td>..</td>
<td>-1703</td>
<td>-5502</td>
<td>-17096</td>
<td>-52477</td>
<td>..</td>
</tr>
</tbody>
</table>

Heuristic: skip iterations, if ‘right’ precision can be estimated.
Core of the arithmetic:

Fast implementation of multiple precision numbers!

- small number of time critical operations
  essentially: basic arithmetic, shifts on (large) integers
  (from `gmp_n`-part of GMP)
- about 20 routines for MP numbers
  conversions, basic arithmetic, comparison
  (`mpf_t` from GMP, `mpfr_t` from MPFR, etc)

During compilation time: choose one of four MP packages

Advantage: easier debugging
(iRRAM triggered 3 errors in GMP and >10 errors in MPFR!)
common properties:

- 32-bit-exponent
- mantissa up to 500 MB
- direct access using class **DYADIC**
Computable real functions are continuous, but...

for imperative programming, (boolean) functions on \( \text{REAL} \) should be total

execution of loops usually depends on \( \text{REAL} \) parameters

\[ \Rightarrow \text{Imperative programming needs/uses total functions } f : \mathbb{R} \to \mathbb{N} \]

\[ \Rightarrow f \text{ must be multi-valued,} \]

i.e. for same \( x \), \( f(x) \) is a choice from several possible values
Example: \( \text{round}(x) = \text{maybe} \lfloor x \rfloor, \text{maybe} \lceil x \rceil \)

Further examples: \( \text{bound}(x, k), \text{positive}(x, k), \text{size}(x), \ldots \)
Typical application: control of loops like in

```c
REAL e_approx (int p)
{
    ...
    while ( !bound(y,p-1) ) { y=y/i; z=z+y; i+=1; } 
    ...
};
```

Problem with iterations:

```c
if ( round (x) == 0 )
    { cout << "Input A"; cin >> a; }
else
    { cout << "Input B"; cin >> b; }
```

- results of multi-valued functions are stored!
- multi-valued functions are ‘expensive’
- but in general: storing results seems to be cheaper than storing computation diagrams ...
Implemented operators for limits of $a_p : M_1 \rightarrow M_2, \ p \in \mathbb{Z}$:

- $a_p(x)$ converging to **single-valued limit** $f(x)$:

  $$|a_p(x) - f(x)| \leq 2^p$$

- $a_p(x)$ converging to **Lipschitz-continuous single-vld. limit** $f(x)$:

  $$|a_p(x) - f(x)| \leq 2^p$$

  $$|f(x) - f(d)| \leq L \cdot e \text{ for } x \in (d \pm e)$$

- $a_p(x)$ converging to **multi-valued limit** $f(x)$. 
(Simplified) Examples:

```cpp
// Compute e≈2.71...
REAL e_approx (long p) {...};
REAL e() { return limit(e_approx); };

// Approximate sqrt using e.g. Heron's method
REAL sqrt_approx (long p, const REAL& x) {...};
REAL sqrt(const REAL& x)
    { return limit(sqrt_approx, x); }

// Approximate sin using Taylor series
REAL sin_approx (long p, const REAL& x) {...};
REAL sin(const REAL& x)
    { return limit_lip(sqrt_approx, 1, x); }
```
simple idea of implementation of limit \((a, x)\):

- choose \(p\)
- compute \(a(p, x)\) for some \(p\), getting an interval \((d' \pm e')\)
- return the interval \((d' \pm (e' + 2^p))\)

more precise:

\[
\text{in iteration with precision bound } \overline{p}_i, \text{ try }
\]

\[
p = \overline{p}_i, \overline{p}_{i-1}, \overline{p}_{i-2}, \ldots
\]

\[
\text{and use ‘best’ result from these ‘local’ iterations}
\]

technical problems:

- be careful not to run into diagonalization problems!
- stop caching multi-valued results temporarily!
implementation of $\text{limit}_\text{lip}(a, l, x)$:

- suppose $x$ is given as $(d \pm e)$
- compute $a(p, d')$ for $p = \bar{p}_i$, $(d' \pm e')$
- return the interval $(d' \pm (e' + 2^p + 2^l \cdot e))$

more precise:

in the computation of $a(\bar{p}_i, d)$, try precision bounds

$$\bar{p}_{i+1}, \bar{p}_{i+2}, \ldots$$

until the first success

remarks:

- error propagation almost reduced to Lipschitz constant
- usually, only one iteration necessary
- used for almost all elementary functions in the iRRAM
**LAZY BOOLEAN** (Lindsay Errington, IC), extension of boolean values:

\[
\{ T, F, \perp \}
\]

dcpo with non-strict, continuous functions:

\[
(x < y) = \begin{cases} 
T, & x < y \\
F, & x > y \\
\perp, & x = y
\end{cases}
\]

\[
(x == y) = \begin{cases} 
F, & x \neq y \\
\perp, & x = y
\end{cases}
\]

| a || b | T | F | ⊥ |
|-----|----|----|----|
| T  | T  | T  | T  |
| F  | T  | F  | ⊥  |
| ⊥  | T  | ⊥  | ⊥  |

<table>
<thead>
<tr>
<th>a&amp;&amp;b</th>
<th>T</th>
<th>F</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>⊥</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>F</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Continuous conversion to `bool`:

<table>
<thead>
<tr>
<th>b</th>
<th>bool(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>⊥</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Consider e.g. ‘if ( x < 0 ) A else B’

- part A will be executed for negative x
- part B will be executed for positive x
- for x = 0, there will be an infinite loop.

Consider e.g. ‘( x < 1 || x > -1)’

- will be evaluated to T without infinite loops at x = 1 or x = -1.
Basic Data Types of the iRRAM:

Standard C++ and additional built-in data types:

- **INTEGER** \( \mathbb{Z} \)
  - max. \( \approx 500 \) MByte per number \((\text{GMP})\)

- **RATIONAL** \( \mathbb{Q} \)
  - max. \( \approx 500 \) MByte per nom./denom. \((\text{GMP})\)

- **DYADIC** \( m \cdot 2^e \)
  - \( e \): 4 Byte, \( m \): max. \( \approx 500 \) MByte,
  - \((\text{GMP, MPFR, LRGMP})\)

- **REAL** \( \mathbb{R} \)
  - 4-byte-exponent, \( \approx 500 \) MByte mantissa

- **LAZY_BOOLEAN** \( \{ T, F, \bot \} \)
  - dcpo with non-strict functions
Basic Operations

**INTEGER / RATIONAL:**
exact versions of $+,-,\times,/,=,<,\leq,\ldots$

**DYADIC:**
approximating versions of $+,-,\times,/$,
exact versions of $=,<,\leq,\ldots$

**REAL:**
exact versions of $+,-,\times,/,=,$
lazy versions of $<,\leq,\leq,\ldots$

(∃ implicit conversions between all numeric datatypes!)

Higher Operators

`limit, limit_lip, limit_mv, iterate, lipschitz,\ldots`
Derived Data Types

COMPLEX, INTERVAL, REALMATRIX, SPARSEREALMATRIX,

High Level Functions

sqrt, power, maximum, minimum,...
exp, log, sin, cos, asin, acos, sinh, cosh, asinh, acosh,...
mag, mig, interval-{+, −, *, /, exp, log, sin, cos},...

eye, zeroes, solve, matrix-{+, −, *, /, exp},...
Application example 1: (indirect) access to lazy booleans

```cpp
friend int choose(const LAZY_BOOLEAN& b1 = false,
                  const LAZY_BOOLEAN& b2 = false,
                  ...
                  const LAZY_BOOLEAN& b6 = false);
```

- one of the arguments of `choose(b1, b2, b3, ...)` is `T`  
  ⇒ result is the *index* of one of these `T` values
- all arguments of `choose(b1, b2, b3, ...)` are `F`  
  ⇒ result is 0
- no `T` values, but mixture of `F` and `⊥`:  
  ⇒ infinite loop...
Application e.g.

```c
REAL maximum (const REAL& x, const REAL& y){
    switch ( choose ( x>y, x<y, true ) ){
        case 1: return x;
        case 2: return y;
        case 3: return (x+y+abs(x-y))/2;
    }
}
};
```
Application example 2: Input and Output

- iterative structure $\Rightarrow$ redefinition of `cin`, `cout`
- new: `irstream`, `orstream`
- usage: similar to original versions...

```cpp
cout « setRwidth(w):

- output of `REAL` with length `w`
- in form `s.mffffffff...lEseeeee`
- with nonzero leading digit `m`
- last digit `1` may be off by one
```
special case: output of zero: no nonzero leading digit...

```cpp
cout « setRflags(iRRAM_float_relative) « REAL(0):
leads to infinite iterations
```

```cpp
cout « setRflags(iRRAM_float_absolute) « REAL(x):
may give output 0 for small $|x|$`
```
REAL x1 = 3.14159;
REAL x2 = "3.14159";
REAL x3 = pi();
REAL x4 = 1e-100;

cout << setRwidth(60);
cout << x1 << "\n";
cout << x2 << "\n";
cout << x3 << "\n";
cout << setRflags(iRRAM_float_absolute) << x4 << "\n";
cout << setRflags(iRRAM_float_relative) << x4 << "\n";

+.3141589999999999882618340052431449294090270996093750E+0001
+.314159000000000000000000000000000000000000000000E+0001
+.3141592653589793238462643383279502884197169399375105E+0001
0
+.10000000000000019991899802602883619647760788534159E-0099
Application example 3: Improved error propagation

\( z = x \times y \) realized as \( I_z = I_x \times I_y \) with intervals \( I_x, I_y, I_z \)

correctness:

\[ \{ \bar{x} \cdot \bar{y} \mid \bar{x} \in I_x, \bar{y} \in I_y \} \subseteq I_z \]

inclusion is usually sharp, i.e. ‘\( = \)’ instead of ‘\( \subseteq \)’

Now consider \( X \) and \( Y \) dependent, e.g. \( z = x \times (1-x) \)

Then for \( I_x = [\frac{1}{4}, \frac{3}{4}] \) e.g.:

\[ \{ \bar{x} \cdot \bar{y} \mid \bar{x} \in I_x, \bar{y} \in I_y \} = [\frac{3}{16}, \frac{4}{16}] \]

but

\[ I_z = I_x \times (1 - I_x) = [\frac{1}{16}, \frac{9}{16}] \]
REAL iteration(const REAL& x) 
{return REAL(3.75)*x*(1-x);}

REAL iteration_lip(const REAL& x) 
{return REAL(3.75) - REAL(7.5)*x;}

void itsyst_with_lipshitz(int count,int width){
    cout « setRwidth(width)
    REAL x = 0.5;
    for ( int i=0; i<=count; i++ ) {
        if ( (i<100) || (i%10)==0 )
            cout « x « " : " « setw(4) « i « "\n";
        x= lipschitz(iteration,iteration_lip,x);
    }
}

<table>
<thead>
<tr>
<th>n</th>
<th>direct</th>
<th></th>
<th>Lipschitz</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>precision</td>
<td>time</td>
<td>precision</td>
</tr>
<tr>
<td>100</td>
<td>&lt;0.1 sec</td>
<td>-240</td>
<td>&lt;0.1 sec</td>
<td>-162</td>
</tr>
<tr>
<td>1000</td>
<td>&lt;0.1 sec</td>
<td>-2166</td>
<td>&lt;0.1 sec</td>
<td>-610</td>
</tr>
<tr>
<td>10000</td>
<td>8.6 sec</td>
<td>-21407</td>
<td>3.0 sec</td>
<td>-5502</td>
</tr>
<tr>
<td>100000</td>
<td>2172 sec</td>
<td>-200601</td>
<td>732 sec</td>
<td>-52477</td>
</tr>
</tbody>
</table>
Application example 4: **Usage Models of the iRRAM**

(1) full program under control of the iRRAM
   — approximative computations using simplified intervals
   — iteration of the whole (or parts of the) program, if necessary
   — core model for computation of functions $f : \mathbb{R} \rightarrow \mathbb{R}$

(2) integrated into ordinary arithmetic on objects $D \subseteq \mathbb{R}$ as a function

$$
\bar{f} : D \times \mathbb{Z} \rightarrow D \quad \text{with} \quad |\bar{f}(d, p) - f(d)| \leq 2^p
$$

$\Rightarrow$ extension of ordinary arithmetic with exact arithmetic

e.g. defining additional functions for GMP/MPFR

 e.g. reference arithmetic for **double**
iRRAM vs. gsl (GNU Scientific Library):
solving linear equations involving Hilbert matrices

<table>
<thead>
<tr>
<th>dimension</th>
<th>iRRAM</th>
<th>gsl</th>
<th>slowdown</th>
<th>relative error of gsl</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.376 ms</td>
<td>0.00264 ms</td>
<td>142</td>
<td>7.84e-13</td>
</tr>
<tr>
<td>10</td>
<td>2.65 ms</td>
<td>0.0099 ms</td>
<td>268</td>
<td>3.61e-05</td>
</tr>
<tr>
<td>20</td>
<td>29.4 ms</td>
<td>0.051 ms</td>
<td>568</td>
<td>3.63</td>
</tr>
<tr>
<td>50</td>
<td>484 ms</td>
<td>0.629 ms</td>
<td>769</td>
<td>15.6</td>
</tr>
<tr>
<td>100</td>
<td>7640 ms</td>
<td>4.64 ms</td>
<td>1645</td>
<td>1008</td>
</tr>
</tbody>
</table>

iRRAM as backend for MPFR:

<table>
<thead>
<tr>
<th></th>
<th>10 decimals</th>
<th>10000 decimals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>native</td>
<td>iRRAM-based</td>
</tr>
<tr>
<td>e(x)</td>
<td>0.0117 ms</td>
<td>0.0577 ms</td>
</tr>
<tr>
<td>ln(x)</td>
<td>0.0388 ms</td>
<td>0.0696 ms</td>
</tr>
<tr>
<td>sin(x)</td>
<td>0.0427 ms</td>
<td>0.0745 ms</td>
</tr>
<tr>
<td>cos(x)</td>
<td>0.0270 ms</td>
<td>0.0678 ms</td>
</tr>
</tbody>
</table>
Application example 5: Analytic Functions (work in progress)

New data type for analytic functions $F$, e.g.

$$ F = \text{ANALYTIC}_R (f, R, M) $$

with

- an evaluation algorithm $\mathfrak{e}$ for $F$
- $R$ such that $f$ holomorphic on $z \in \mathbb{C}$ with $|z| \leq R$
- $M$ such that $|f(z)| \leq M$ for $|z| \leq R$

```cpp
ANALYTIC_R a_sqr(square, REAL(10), REAL(100));
ANALYTIC_R a_cos(cos, REAL(10), REAL(20000));

ANALYTIC_R test1 = a_sqr.derivative();
ANALYTIC_R test2 = a_cos.integral();

ANALYTIC_R a_prod = a_sqr*a_cos;
ANALYTIC_R test3 = a_prod.integral();
cout << test3(REAL(1)) << "\n";
```
Work in progress / Important questions:

- use hardware floats where possible ⇒ RealLib v3, iRRAM 2005_02pre1
- mix computation diagrams and iterations
- add level for numbers using more than 32-bit-exponents
- integrate exact arithmetic into program verification tools
- can exact arithmetic be faster than 64-bit hardware floats?
  - use (polynomial time) problems where hardware should need exponential time
  - i.e. address problems from integration / differential equations
  - current state: prototype implementation of holomorphic functions