Enhancing *imperative* exact real arithmetic with functions and logic

Norbert Müller

Universität Trier

---

1 Email: mueller@uni-trier.de
Introduction

Basic structure of the iRRAM

Function objects

Application Examples

Final remarks
The ‘right’ programming language for real arithmetic?

- functional? (O’Connor-05), (Bauer+Kavkler-08), (Escardo-??), (Müller-95), ...

  \[ x \in \mathbb{R} \approx \Phi : \mathbb{N} \rightarrow \mathbb{Q} \]

  \[ f : \mathbb{R} \rightarrow \mathbb{R} \approx \Psi : (\mathbb{N} \rightarrow \mathbb{Q}) \rightarrow (\mathbb{N} \rightarrow \mathbb{Q}) \]

  - usual lazy evaluation
    \[ \sim \] (often) abysmal performance
    \[ \sim \] use ‘stateful’ functional programming (Monads, OCAML)

- imperative (Err.+Heck.-02), (Briggs-06), (Lambov-07), (Müller-96), ....

  - CPUs work in imperative way
    \[ \sim \] easier to exploit all the possibilities of current CPUs

- optimal(?): use best parts of both programming schemes

- where to switch: elegancy (functional) \[ \leftrightarrow \] efficiency (imperative)?

- many flavors of exact real arithmetic might survive...
Main focus: low-precision computations!

- Optimized for up \(\approx 1000\) bits per number
- Allowing big data sets (\(\sim\) linear algebra)
- Higher precision seamlessly usable
- Suitable for billions of dependant operations, e.g.

\[
\sum_{1 \leq n \leq 10^9} \frac{1}{n}
\]

(double precision ‘approximation’: simple loop, <10 seconds)

Today:

- few new internals of iRRAM package since CCA-2005 (Nijmegen)
- try expressiveness of functional languages in C++
- approach started 2005, since 2008 part of published iRRAM
Motivating example: Differential equations

- try to solve IVPs
- efficient solutions need holomorphic flow
- computing with Taylor series is much faster near the center...

→ solve IVPs in steps...

example: $\mathbf{y} = \dot{\mathbf{x}}$, $\mathbf{x} = -\dot{\mathbf{y}}$ with solution $(\sin(t), \cos(t))$

→ Iterate: $f_i := IVP(\mathbf{x}_i)$, $\mathbf{x}_{i+1} := f_i(t_i)$
1 Introduction

2 Basic structure of the iRRAM

3 Function objects

4 Application Examples

5 Final remarks
Aim: be competitive with ordinary ‘double’ arithmetic!

- consider bit complexity (at low level!)
- Turing machines are ‘realistic’ model (Type-2 or Oracle)
- differences between constructive / classical logic not yet important

TTE is reflected in many implementational (often hidden!) details:

- use some admissible representation for $\mathbb{R}$
- I/O operations can model Type-2-machines
- other modes look like oracle machines
- metric spaces with dense subsets can be used
- use multi-valued functions
- ...

Norbert Müller (Universität Trier) Nijmegen 2009
Other Approaches

- using functional extension \( F_{\text{C++}} \) of \( \text{C++} \), (Briggs-06)

\[
x \in \mathbb{R} \approx \lambda : \mathbb{Z} \rightarrow \mathbb{Z}
\]

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; ...
    }
}

Directed Acyclic Graphs (DAGs)

- DAGs: simple programming exercise...
- quite memory intensive: $\approx 50$ bytes per node
- Total number of DAG nodes at most $\approx 100$ million.
- DAG-construction roughly comparable to ‘lazy evaluation’
Removing the algebraic structure of DAGs

- implemented sequence $F$ of functions $F : \mathbb{N} \times \mathbb{R} \rightarrow \mathbb{R}$
- known speed of convergences to limit $f$, e.g.

$$|f(z) - F(n, z)| \leq 2^{-n}$$

- then $Y = f(X)$ is implemented as $Y = \text{limit}(F, X)$
Evaluation of DAGs

- conversion from DAGs to other datatypes...
- usually as a multivalued function
- top-down (Briggs-06):
  - gross overestimations of precision
  - bad performance...
- bottom-up (Lambov-07)
  - based on interval arithmetic
  - in iterations
  - reasonable performance
  - efficient evaluation not trivial!

\[ X_0 = 0.5; \]
\[ X_1 = (1 - X_0) \times (X_0 \times 3.75); \]
\[ X_2 = (1 - X_1) \times (X_1 \times 3.75); \]
\[ X_3 = (1 - X_2) \times (X_2 \times 3.75); \]

\[ X_0 = 0.5; \]
\[ X_1 = (1 - X_0) \times (X_0 \times 3.75); \]
\[ X_2 = (1 - X_1) \times (X_1 \times 3.75); \]
\[ X_3 = (1 - X_2) \times (X_2 \times 3.75); \]

iRRAM:

- forget about the DAG..., 
- directly iterate the program!

\[ \sim \] admissible representation using intervals
\[ \sim \] sequence spread over time ...
\[ \sim \] based on: \( \mathbb{R} \) is metric space with ‘simple’ dense subset \( \mathbb{Q} \)
Underlying Intervals:

- usually: centered intervals $\mathcal{I}$
  \[ \mathcal{I} = \{ x : d - r \leq x \leq d + r \} \]
  center $d$: multiple precision (e.g. MPFR)
  radius $r$: fixed precision (but large exponent range)

- in initial iteration:
  \[ \mathcal{I} = \{ x : l \leq x \leq u \} \]
  with $l, r$: double precision (hardware, fast!!!!!!)

Optimization of this initial iteration:

- inline functions, no function calls for basic arithmetic!
- `malloc` will not be called.
- $[l, u]$ is stored as pair $(l, -u)$ (no change of rounding modes!)
- use SSE2-parts of CPUs for parallelization
- iRRAM: dynamic switching between interval types
  (RealLib: fixed at compile time)
### Short benchmark...

$$\sum_{i=1}^{n} \frac{1}{i}$$ as

```c
x=0; one=1;
for (i=1; i<=n; i++)
  x = x + one/i
```

<table>
<thead>
<tr>
<th>package</th>
<th>$n$</th>
<th>precision</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>unverified double</td>
<td>$n = 10^9$</td>
<td>(10 decimals)</td>
<td>7.5s</td>
<td>$\approx 0$ B</td>
</tr>
<tr>
<td>Reallib, DAGs</td>
<td>$n = 10^7$</td>
<td>-</td>
<td>-</td>
<td>1.7 GB</td>
</tr>
<tr>
<td>Reallib, DAGs</td>
<td>$n = 10^9$</td>
<td>-</td>
<td>-</td>
<td>unfeasable</td>
</tr>
<tr>
<td>iRRAM</td>
<td>$n = 10^9$</td>
<td>5 decimals</td>
<td>15s</td>
<td>$\approx 0$ B</td>
</tr>
<tr>
<td>Reallib, DAG-free</td>
<td>$n = 10^9$</td>
<td>5 decimals</td>
<td>12.5s</td>
<td>$\approx 0$ B</td>
</tr>
<tr>
<td>iRRAM</td>
<td>$n = 10^9$</td>
<td>20 decimals</td>
<td>300s</td>
<td>$\approx 0$ B</td>
</tr>
<tr>
<td>Reallib, DAG-free</td>
<td>$n = 10^9$</td>
<td>20 decimals</td>
<td>550s</td>
<td>$\approx 0$ B</td>
</tr>
</tbody>
</table>
Lazy Booleans:

- Extend `bool = \{ T, F \}` to
  
  `LAZY_BOOLEAN = \{ T, F, \bot \}` (Err.+Heck.-02)

| a || b | T | F | \bot |
|----|----|----|----|------|
| T  | T  | T  | T  | T    |
| F  | T  | F  | T  | F    |
| \bot | T | \bot | \bot | \bot |

<table>
<thead>
<tr>
<th>a&amp;&amp;b</th>
<th>T</th>
<th>F</th>
<th>\bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>\bot</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>\bot</td>
<td>\bot</td>
<td>F</td>
<td>\bot</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>!a</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>\bot</td>
<td>\bot</td>
<td>F</td>
</tr>
</tbody>
</table>

`(x<y) = \begin{cases} 
T, & x < y \\
F, & x > y \\
\bot, & x = y 
\end{cases}

<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>\bot</td>
<td>undefined</td>
</tr>
</tbody>
</table>

- Implicit conversion to `bool` e.g. in `if ( a<b && c>d ) { ... }`
- Conversion `\bot` to `bool` leads to iteration
- Expressions like `x<1 || x>0` are total now!
1 Introduction

2 Basic structure of the iRRAM

3 Function objects

4 Application Examples

5 Final remarks
Functions as objects?

- older functional approaches in C++ already too slow for
  \[ x \in \mathbb{R} \approx \Phi : \mathbb{N} \rightarrow \mathbb{Q} \]

- so no real functions as objects... (only as predefined algorithms...)
  \[ f : \mathbb{R} \rightarrow \mathbb{R} \approx \Psi : (\mathbb{N} \rightarrow \mathbb{Q}) \rightarrow (\mathbb{N} \rightarrow \mathbb{Q}) \]

- standard TTE approach:
  - use a metric on functions spaces
  - implement some dense subset
  - implement functions as limits
  \[ \leadsto \text{exponential space complexity...} \]
Solution

- Use DAGs...
- Add enough structure to get an admissible representation...
- ... but don’t try to implement the conversion procedures...
- What then is the main ‘evaluation’ for these DAGs?
  - functional values at points?
  - functional values at sets or intervals?

Current state (neither minimal nor complete...)

- use ‘algebraic approach’ (functional values at points)
- influenced by (Brattka-98)

```
template<class PARAM, class RESULT> class FUNCTION;
```
### Function objects

#### For arbitrary parameter and result types:

<table>
<thead>
<tr>
<th>General constructors</th>
<th>( h = \text{from_algorithm}) (my_alg) \hfill ( h(x) = g(f(x)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h = \text{from_value}) (my_constant)</td>
<td>\hfill ( h(x) = (f(x), g(x)) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structural operators</th>
<th>( h = \text{compose}(g,f), h=g(f) ) \hfill ( h(x) = g(f(x)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h = \text{product}(f,g) ) \hfill ( h(x, y) = (f(x), g(y)) )</td>
<td></td>
</tr>
<tr>
<td>( h = \text{juxtaposition}(f,g) ) \hfill ( h(x) = (f(x), g(x)) )</td>
<td></td>
</tr>
</tbody>
</table>

| Evaluation | \( y = f(x) \) |

<table>
<thead>
<tr>
<th>Projections</th>
<th>( h_1 = \text{first}(f), h_2 = \text{second}(f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( hi = \text{projection}(f,i) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partial Application</th>
<th>( h = \text{bind_first}(f,a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h = \text{bind_second}(f,b) )</td>
<td></td>
</tr>
</tbody>
</table>

#### For result types with arithmetic operations:

<table>
<thead>
<tr>
<th>Special arithmetic</th>
<th>( h = f+g )</th>
</tr>
</thead>
</table>

| Special constructors | \( h = \text{polynomial}(\text{coeff\_vector}) \) |

| Miscellaneous | \( h = f+c, h = c+f, \ldots \) |

#### For result type lazy\_boolean:

<table>
<thead>
<tr>
<th>Comparison operators</th>
<th>( h = f&lt;g ) \hfill ( h(x) = f(x) &lt; g(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h = f&gt;g )</td>
<td></td>
</tr>
</tbody>
</table>

| Logical operators | \( h = f||g \) |
|-------------------|-------------------------------------------------|
| \( h = f&&g \) | |
| \( h = !_f \) | |
1 Introduction

2 Basic structure of the iRRAM

3 Function objects

4 Application Examples

5 Final remarks
A simple function...

Exercise: implement logistic map

\[ x_{i+1} = f(x_i) \ , \ f(x) = 3.75 \cdot (1 - x) \cdot x \]

```cpp
#include "iRRAM.h"
using namespace iRRAM;

REAL id_REAL( const REAL& x ){ return x; };

void compute(){
    int count; cin >> count; REAL x = 0.5;

    FUNCTION<REAL,REAL> REAL X = from_algorithm(id_REAL);

    FUNCTION<REAL,REAL> LM = REAL(3.75) * (REAL(1) - X) * X;

    for ( int i=1; i<=count; i++ ) x = LM(x);

    cout << x << "\n";
}
```

(eficiency here almost identical to direct implementation)
Dedekind cuts:

- Example: $\sqrt{2}$ can be defined by
  - bounds $l = 1$, $u = 2$
  - lower set $L = \{ x \mid x < 0 \vee x^2 < 2 \}$
  - upper set $U = \{ x \mid 0 < x \land x^2 > 2 \}$

- Represent e.g. $L$ as a predicate:
  ```cpp
  REAL id_REAL( const REAL& x ) { return x; };
  FUNCTION<REAL,REAL> X = from_algorithm(id_REAL);
  FUNCTION<REAL,LAZY_BOOLEAN> L = X<REAL(0) || X*X<REAL(2);
  ```
Basic idea usable as constructor for a `REAL`, e.g.

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

Now implement $\sqrt{2}$ by

```cpp
void compute(){
  REAL id_REAL( const REAL& x ){ return x; };
  FUNCTION<REAL,REAL> X=from_algorithm(id_REAL);

  REAL x = cut(1,2,
      X<REAL(0) || X*X<REAL(2),
      REAL(0)<X && REAL(2)<X*X);

  cout « setRwidth(100) « x « "\n";
}
```

current implementation of `cut`:
- unoptimized trisection with slow limit operator..
- just useful as proof of concept...
Predicates on real numbers:

- predicates on reals $\approx$ functions to \texttt{LAZY\_BOOLEAN}
- ‘real’ logic in \texttt{C++}
- small example:

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

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

void compute(){
    REAL_EXPRESSION X = projection(variables,0);
    REAL_EXPRESSION Y = projection(variables,1);
    REAL_EXPRESSION Z = X*Y+X/Y;
    REAL_PREDICATE P = Z > X || X < REAL(1) ;

    UNIVERSE assign(2);
    assign[0] = 1; assign[1] = 2;
    cout « Z(assign) « " " « P(assign) « "\n";
}
```
1 Introduction

2 Basic structure of the iRRAM

3 Function objects

4 Application Examples

5 Final remarks
A lot of the elegance of functional languages can be ported to C++

Current state allows evaluation with good efficiency and reasonable error propagation

Functions objects can easily be extended:
  ▶ Evaluation could be optimized with regards to error propagation (with losses in speed)
  ▶ This is closely connected to efficient evaluation on sets (not only intervals...)
  ▶ Things like automated differentiation can easily be added...
  ▶ The predicates should be enhanced with quantification... (efficiently?)

Parts of verification on real numbers (O’Connor-05),(Bauer-08) can be expressed in object-oriented languages!

A hypothetical approach mixing object-oriented and genuinely functional languages might switch between the paradigms on a quite high level!
Thank you for your attention!
Any questions or remarks?


References

und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
<http://drops.dagstuhl.de/opus/volltexte/2006/714> [date of citation: 2006-01-01].


[Escardo-??] Martín Escardo. ??????