Some Steps into Verification of Exact Real Arithmetic

Norbert Müller*†

Universität Trier
Germany

*j.w.w. Christian Uhrhan (Siegen), Michael Schausten (Trier), Michael Simon (Trier)

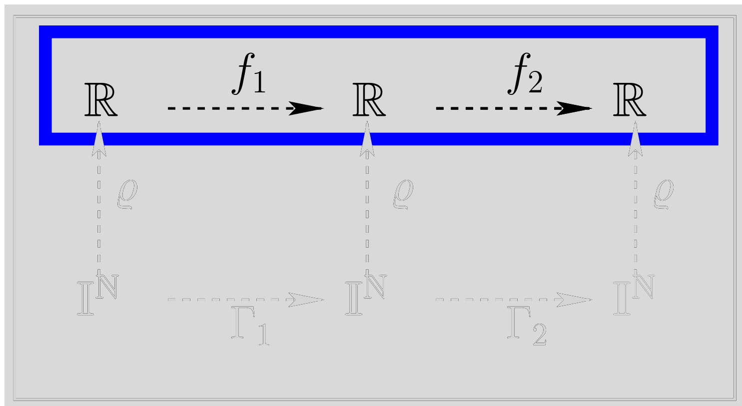
†EU project: CID Marie Skłodowska-Curie 731143

1 Introduction

2 Verification approach

3 Work in progress

Computable analysis (via 'representations'):



Remember: Computable functions are **continuous!**

Verified packages for exact real arithmetic:

- **`Haskell/PVS`**, David Lester, 2008
- **`Haskell`**/Coq, Russell O'Connor, 2009
- **`OCaml`**/Coq, Andrej Bauer, 2009
- **`Haskell`**/Coq, Krebbers/Spitters, 2011

... but we need verified *and* fast algorithms!

expression	decimals	K/S	`iRRAM`	factor
$cos(10^{50})$	20000	1.45s	76ms	20
$sin(sin(sin(1)))$	5000	2.3s	22ms	100
$log(1+log(1+log(1+log(1+\pi))))$	500	2.5s	0.3ms	1000

- work in progress: verification of fast real arithmetic in `C++`...
- ... but verifying `C++` is hard...

Two objectives:

- internal use: verify correctness of the iRRAM package
- external use: develop verification tools for the user

Current approach based on verification of `C`:

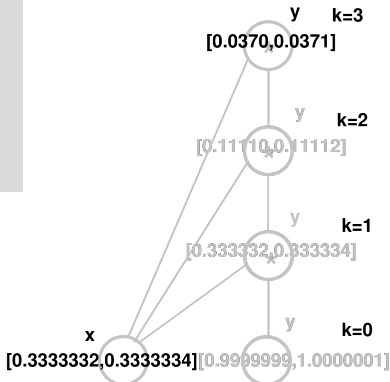
- use `ACSL` to specify semantics
- use `FRAMA-C` (with `Jessie` and `WHY3`) to translate to COQ
- use `coqide` to write proofs...

Precise view on computations:

$$
\begin{array}{ccccccccc}
\overline{x}_0 & \xrightarrow{f_0} & \overline{x}_1 & \xrightarrow{f_1} & \overline{x}_2 & \xrightarrow{f_2} & \ldots & \xrightarrow{f_{n-1}} & \overline{x}_n \\
\uparrow & & \uparrow & & \uparrow & & & & \uparrow \\
\begin{pmatrix} \vdots \\ O_{0,p} \\ \vdots \\ O_{0,1} \\ O_{0,0} \end{pmatrix} & \xrightarrow{\Gamma_1} & \begin{pmatrix} \vdots \\ O_{1,p} \\ \vdots \\ O_{1,1} \\ O_{1,0} \end{pmatrix} & \xrightarrow{\Gamma_2} & \begin{pmatrix} \vdots \\ O_{2,p} \\ \vdots \\ O_{2,1} \\ O_{2,0} \end{pmatrix} & \xrightarrow{\Gamma_3} & \ldots & \xrightarrow{\Gamma_n} & \begin{pmatrix} \vdots \\ O_{n,p} \\ \vdots \\ O_{n,1} \\ O_{n,0} \end{pmatrix}
\end{array}
$$

iRRAM: exact real arithmetic with 'approximating' approach

```
REAL z = power( "0.33333333" , 3);

REAL  power(const REAL& x, int n) {
  REAL y=1;
  for (int k=0; k<n; k=k+1)
       { y=x*y; }
  return y;
}
```



- iteration of computations!
- 'Exceptions' are the rule...

data structures behind REAL variables ...

... represent only approximations

iRRAM: Use specialized realizers!

$$
\begin{array}{ccccccccc}
\overline{x}_0 & \xmapsto{f_1} & \overline{x}_1 & \xmapsto{f_2} & \overline{x}_2 & \xmapsto{f_3} \;\dots\; \xmapsto{f_n} & \overline{x}_n \\
\uparrow & & \uparrow & & \uparrow & & \uparrow \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
O_{0,p} & \xmapsto{\Gamma_{1,p}} & O_{1,p} & \xmapsto{\Gamma_{2,p}} & O_{2,p} & \xmapsto{\Gamma_{3,p}} \;\dots\; \xmapsto{\Gamma_{n,p}} & O_{n,p} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
O_{0,1} & \xmapsto{\Gamma_{1,1}} & O_{1,1} & \xmapsto{\Gamma_{2,1}} & O_{2,1} & \xmapsto{\Gamma_{3,p}} \;\dots\; \xmapsto{\Gamma_{n,p}} & O_{n,1} \\
O_{0,0} & \xmapsto{\Gamma_{1,0}} & O_{1,0} & \xmapsto{\Gamma_{2,0}} & \perp & \dots & \perp
\end{array}
$$

1 Introduction

2 Verification approach

3 Work in progress

Approach in 4 levels: ‘separation of concerns’ [Cyril Cohen]

- **Level 4:** applications

  $\rightsquigarrow$ verify non-basic operations and user tools (mainly external use)

- **Level 3:** basic arithmetic

  $\rightsquigarrow$ verify convergence to basic operations on real numbers (mainly internal use)

- **Level 2:** intervals

  $\rightsquigarrow$ verify basic operations on approximations (mainly internal use)

- **Level 1:** core routines

  $\rightsquigarrow$ verify arithmetic for (arbitrarily precise) floating point numbers (mainly internal use)

Verification level 1: core routines for floating point arithmetic
(M., Michael Schausten, 2010)

- `MPFR`: out of reach...
- `double`:
  - Gappa (Génération Automatique de Preuves de Propriétés Arithmétiques)
  - Coquelicot (Coq library for real analysis)
- `sizetype`:
  - standard `Coq` reals suffice...

sizetype:

```
typedef unsigned int SIZETYPEMANTISSA;
typedef int SIZETYPEEXPONENT;

typedef struct _sizetype {
   SIZETYPEMANTISSA mantissa;
   SIZETYPEEXPONENT exponent;
} sizetype;
```

Semantics in **ACSL**:

```
/*@
  logic real sizetype_value{L}(sizetype* a)
    = (\at((*a).mantissa, L) * \pow(2, \at((*a).exponent, L)));

  predicate mantissa_valid(sizetype* a) =
        ((*a).mantissa <= max_mantissa);

  predicate exponent_valid(sizetype* a) =
      ((*a).exponent >= MP_min && (*a).exponent <= MP_max);

  predicate sizetype_valid(sizetype* a) =
      (mantissa_valid(a) && exponent_valid(a));
*/
```

```
 1 // sizetype_add(x,y,z) ****************************************
 2 // Add y and z yielding x
 3 // Argument x must be different from y and z!
 4 // The resulting value may be a bit larger than the exact value,
 5 // The resulting value may never be smaller than the exact value
 6 // ***********************************************************
 7
 8 inline void sizetype_add(
 9           sizetype& x,
10     const sizetype& y,
11     const sizetype& z)
12 {
13   sizetype_add_wo_norm(x,y,z);
14   sizetype_normalize(x);
15 }
```

```
1  // sizetype_add_wo_norm(x,y,z) ***********************************
2  // Add y and z yielding x
3  // Argument x must be different from y and z!
4  // The resulting value may be a bit larger than the exact value,
5  // The resulting value may never be smaller than the exact value
6  // *****************************************************************
7
8  inline void sizetype_add_wo_norm(
9          sizetype& x,
10   const sizetype& y,
11   const sizetype& z)
12 {
13   if ( y.exponent >  z.exponent) {
14     x.exponent= y.exponent;
15     x.mantissa= y.mantissa
16                + scale(z.mantissa,(x.exponent-z.exponent)) + 1;
17   } else {
18     x.exponent= z.exponent;
19     x.mantissa= z.mantissa
20                + scale(y.mantissa,(x.exponent-y.exponent)) + 1;
21   }
22 }
```

```
1  inline void sizetype_normalize( sizetype& e) {
2    if (iRRAM_unlikely(e.mantissa < min_mantissa)) {
3          e.mantissa <<= BIT_RANGE;
4          e.exponent -= BIT_RANGE;
5    }
6    if (iRRAM_unlikely( e.mantissa >=  max_mantissa ) ) {
7          e.mantissa = ( e.mantissa>> DIFF_BITS ) + 1;
8          e.exponent += DIFF_BITS;
9    }
10   if (iRRAM_unlikely( e.exponent < MP_min ) ){
11     e.exponent = min_exponent;
12   }
13   if (iRRAM_unlikely( e.exponent >= MP_max ) )
14   {
15     iRRAM_DEBUG1(1,"exponent too big in sizetype_normalize ");
16     REITERATE(0);
17   }
18 }
```

Original version:

```
1 #define iRRAM_DEBUG1(level,p)    iRRAM_DEBUG0((level),cerr << p)
2 #define iRRAM_unlikely(x)         __builtin_expect(!!(x), 0)
3 #define REITERATE(x)   { inReiterate = true; throw Iteration(x);};
```

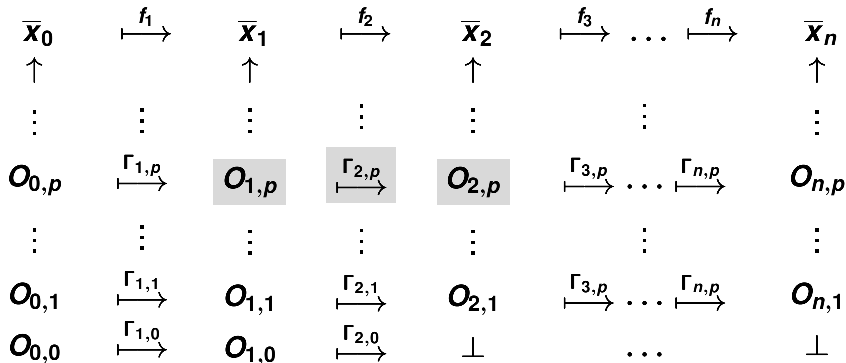
Verified version:

```
1 #define iRRAM_unlikely(b) (b)
2 #define iRRAM_DEBUG1(x,y) {}
3 #define REITERATE(x) { exception_occured = 1; }
```

verified formal specification of **sizetype_add**

```
1  /*@
2    requires \valid(x) && \valid(y) && \valid(z);
3    requires (exception_occured != 0) ||
4          ((−2147483640 <= (*y).exponent
5                           && (*y).exponent <= 2147483644)
6        && (−2147483640 <= (*z).exponent
7                           && (*z).exponent <= 2147483644)
8        && sizetype_valid(y) && sizetype_valid(z));
9
10   ensures (exception_occured != 0) ||
11         (sizetype_value{Here}(x) >=
12               sizetype_value{Old}(y)
13             + sizetype_value{Old}(z))
14       && sizetype_valid(x));
15 */
16 inline void sizetype_add(
17           sizetype& x,
18     const sizetype& y,
19     const sizetype& z)
```

Verification level 2: interval arithmetic (M., Michael Simon, 2011)



enhanced specification of operations on **sizetype**:

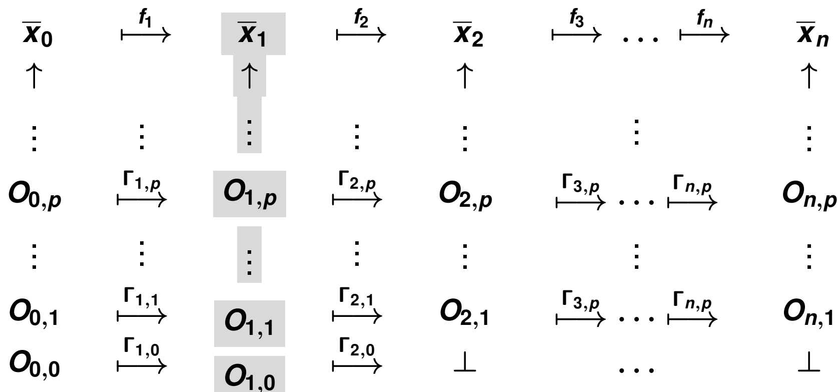
```
/*@
  requires \valid(x);

  assigns x->exponent, x->mantissa;

  ensures \valid(x)
          && (real_of_sizetype_pointer{Old}(x) + \pow(2,zexp)
                       <= real_of_sizetype_pointer{Here}(x))

          && 2* (real_of_sizetype_pointer{Old}(x) + \pow(2,zexp)
                       >= real_of_sizetype_pointer{Here}(x));
*/
void sizetype_inc_one(const sizetype_pointer x, const int zexp);
```

slightly simplified datatype **REAL**:

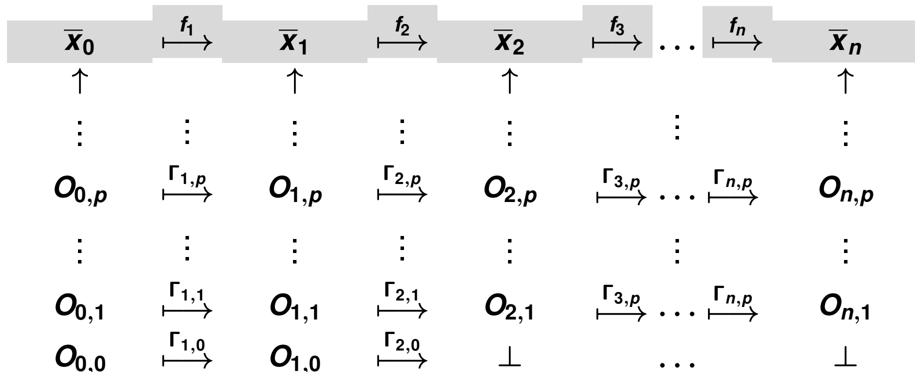
```
typedef struct _MP { int dummy; } *MP_pointer;

/*@ axiomatic conversion {
      logic real real_of_MP(MP_pointer x);
    }
*/

typedef struct REAL
{
      MP_pointer value;
      sizetype   error;
      //sizetype    vsize;
      //double lower_pos,upper_neg;
} *REAL_pointer;
```

```
1  int ACTUAL_STACK_actual_prec = 0;
2  int exception_occured = 0;
3
4  /*@
5      requires \valid(x) && \valid((x->value)) ...
6  ...
7      ensures   (real_of_MP(x->value)+real_of_MP(y->value)
8           + real_of_sizetype{Old}(&(x->error))
9           + real_of_sizetype{Old}(&(y->error))
10        < real_of_MP(\result->value)
11          + real_of_sizetype{Here}(&(\result->error)));
12 */
13 REAL_pointer add (const REAL_pointer x,
14                    const REAL_pointer y)
15 {
16   MP_pointer zvalue;
17   sizetype_pointer zerror;
18 ...
19   MP_add(x->value,y->value,zvalue,local_prec);
20
21   sizetype_add_wo_norm(&zerror,&x->error,&y->error);
22   sizetype_inc_one(&zerror, ACTUAL_STACK_actual_prec);
23   return REAL_constructor(zvalue,zerror);
24 }
```

Verification level 3: basic arithmetic (open...)



Verification level 4: applications (M., Christian Uhrhan, 2011)



- `jessie` basically uses classical reals in Coq
- necessary extensions e.g.:

```
1  Axiom jessie_coq_tan:
2    forall x : R, jessie_why.tan x = Rtrigo.tan x .
3
4  Axiom jessie_coq_sin:
5    forall x : R, jessie_why.sin x = Rtrigo_def.sin x .
6
7  Axiom jessie_coq_cos:
8    forall x : R, jessie_why.cos x = Rtrigo_def.cos x .
9
10 Axiom jessie_coq_pi:
11   jessie_why.pi = Reals.AltSeries.PI .
```

- better: **`Coq-CoRN`** and **`MathCLasses`** (Spitters et.al.)

constructors / destructors:

- only weak support for memory (de-)allocation
- currently(???), **`jessie`** does not implement **`malloc`** and **`free`**
- although **`ACSL`** has **`fresh`**, **`free`**
- so we tried to use ‘preallocated’ memory:

```
1  /* heap for MP */
2  typedef struct _HEAP_MP {
3    int* used[HEAPSIZE];
4    MP_data data[HEAPSIZE];
5  } HEAP_MP;
6
7  /* heap for REAL */
8  typedef struct _HEAP_REAL {
9     int* used[HEAPSIZE];
10    REAL data[HEAPSIZE];
11 } HEAP_REAL;
```

`ACSL`-logic quite weak:

- memory changes via **`assigns`** clauses
- semantics of **`assigns a,b,c`** is:

  no memory changes except on **{*`a,b,c`*}**

  i.e., **`assign`** works on sets of locations
- logic for sets not fully functional... (no usable set variables)

Operator overloading

C++ allows `x*y*z` instead of `mul(mul(x,y),z)`

```
1 // C++ version
2 friend REAL operator   * (const REAL& x, const REAL& y);
3 friend REAL operator   * (const REAL& x, const int&  y);
```

```
1 // translation to C
2 REAL_pointer REALREAL_mul(REAL_pointer x, REAL_pointer y);
3 REAL_pointer REALint_mul (REAL_pointer x, int  y);
```

Exceptions

z=x*y can be modeled as

```
{REAL_pointer tmp = REALREAL_mul (x,y);
  if(exception != 0) return 0; z=tmp;}
```

abbrevated via macro definition:

```
#define SIGNALS(z,y,x) {
    z value=x;
    if(exception !=0) return 0; y=value;
}
```

- **C** has methods **longjmp** / setjmp
- but not yet implemented in **ACSL**
- maybe **Krakatoa** for **Java** can be used as basis

Example: power x^n with $x \in \mathbb{R}$ and $n \in \mathbb{N}, n \geq 0$.

A working implementation in the **iRRAM** is:

```
REAL power(const REAL& x, int n) {
  REAL y=1;
  for (int k=0; k<n; k=k+1) { y=y*x; }
  return y;
}
```

Translated to **C** we get:

```
REAL_pointer REALint_power(const REAL x, int n) {
  REAL_pointer y;
  SIGNALS(REAL_pointer, y, REAL_from_int32(1));

  for (int k=0;k<n;k=k+1)}
     SIGNALS(REAL_pointer, y, REALREAL_mul(y,x));
  }
  return y;
}
```

Example: power x^n with $x \in \mathbb{R}$ and $n \in \mathbb{N}, n \geq 0$.

```
1 /* "function contract" for power of real numbers (level 4) */
2 /*@
3 requires   valid_REAL(x) && n >= 0;
4 assigns    exception ;
5 ensures    exception==0   ==> ( valid_REAL(\result)   &&
6            real_of_iRRAM_REAL (\result) ==
7                \pow(real_of_iRRAM_REAL (x),n) );
8 */
9 REAL REALint_power(const REAL x, int n);
```

'trivial' loop invariant in **ACSL**:

```
1  /*@
2    loop invariant valid_REAL(y);
3    loop invariant real_of_iRRAM_REAL (y)
4                     == \pow(real_of_iRRAM_REAL (x),k);
5    loop invariant 0 <= k <= n;
6    loop variant n-k;
7  */
8    for (int k=0;k<n;k=k+1){
9        SIGNALS(REAL_pointer, y, REALREAL_mul(y,x));
10   }
```

1 Introduction

2 Verification approach

3 Work in progress

What have we done so far?

- 30 central functions verified
- about 4% of 800 functions in total with about 12000 lines of code
- still a lot of work to do...

Current work:

- define verifiable language for level 4...