Some Steps into Verification of Exact Real Arithmetic

Norbert Müller / Christian Uhrhan

Universität Trier / Universität Siegen
Germany

1 Idea behind the iRRAM

2 Verification approach

3 From C++ to C

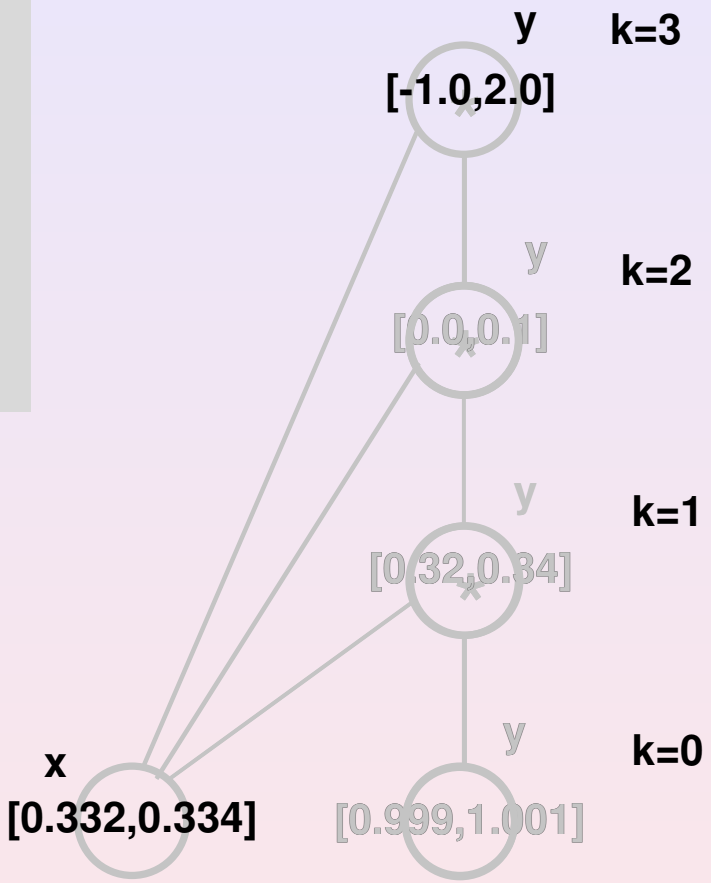
4 Example

5 Work in progress

exact real arithmetic: ‘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;
}
```

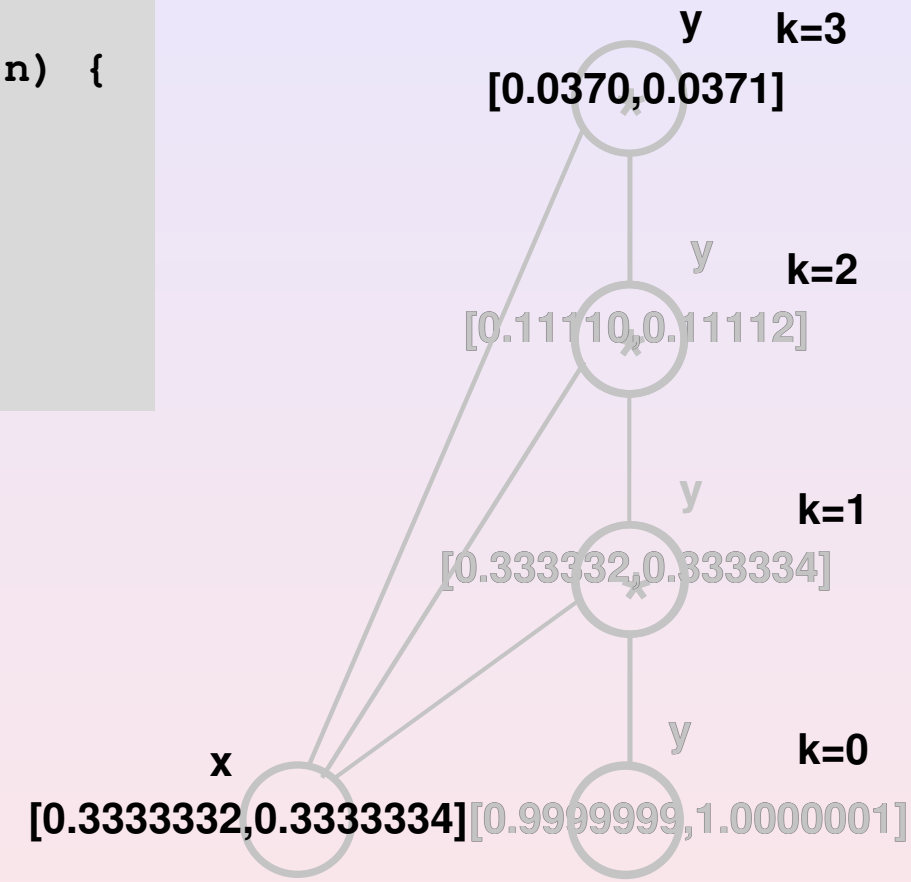


exact real arithmetic: ‘approximating’ approach

```
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

1 Idea behind the iRRAM

2 Verification approach

3 From C++ to C

4 Example

5 Work in progress

- 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 `WHY`) to translate to `coq`
- use `coqide` to write proofs...

Approach in 4 levels:

1. core level

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

2. interval level

   $\leadsto$ verify special interval arithmetic (mainly internal use)

3. basic arithmetic level

   $\leadsto$ verify basic operations on real numbers (mainly internal use)

4. application level

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

1 Idea behind the iRRAM

2 Verification approach

3 From C++ to C

4 Example

5 Work in progress

- constructors / destructors.
- Operator overloading: `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
3 REAL REALREAL_mul(REAL x, REAL y);
4 REAL REALint_mul (REAL x, int  y);
```

- exceptions: `z=x*y` can be modeled

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

1 Idea behind the iRRAM

2 Verification approach

3 From C++ to C

4 Example

5 Work in progress

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

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

```
1 REAL power(const REAL& x, int n) {
2   REAL y=1;
3   for (int k=0; k<n; k=k+1) { y=y*x; }
4   return y;
5 }
```

Translated to C we get:

```
1 REAL REALint_power(const REAL x, int n) {
2   REAL y;
3   { REAL tmp = REAL_from_int32(1);
4     if (exception != 0) return 0; y=tmp;}
5   for (int k=0;k<n;k=k+1)
6     { REAL tmp = REALREAL_mul(y,x);
7       if (exception != 0) return 0; y=tmp;}
8   return y;
9 }
```

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) && 0 <= k <= n &&
3       real_of_iRRAM_REAL (y) == \pow(real_of_iRRAM_REAL (x),k);
4 loop variant n-k;
5 */
6   for (int k=0;k<n;k=k+1)
7     { REAL tmp = REALREAL_mul(y,x);
8       if (exception != 0) return 0; y=tmp;}
```

1 Idea behind the iRRAM

2 Verification approach

3 From C++ to C

4 Example

5 Work in progress

- as far as possible: automate translation of C++ to C
- introduce fast (and verified) datatype for $\mathbb{Z}$
- readjust specifications/proofs of the different levels

Thank you for your attention!

Any questions or remarks?