Some Steps into Verification of Exact Real Arithmetic[⋆]

Norbert Th. Müller[1] and Christian Uhrhan[2]

[1] Abteilung Informatik, FB IV, Universität Trier, Germany
[2] Universität Siegen, Faculty IV, Germany

Abstract. The mathematical concept of real numbers is much richer than the double precision numbers widely used as their implementation on a computer. The field of 'exact real arithmetic' tries to combine the elegance and correctness of the mathematical theories with the speed of double precision hardware, as far as possible. In this paper, we describe an ongoing approach using the specification language `ACSL`, the tool suite `Frama-C` (with `why` and `jessie`) and the proof assistant `Coq` to verify central aspects of the `iRRAM` software package, which is known to be a fast `C++` implementation of 'exact' reals numbers.

1 Introduction

The verification of programs using double precision numbers often is very complicated: the semantics of this number format does not coincide with the semantics of real numbers, i.e. with the definitions and results found in textbooks on calculus. On the other hand, it is possible to implement 'exact' real numbers in software ([BK08, OS10, Mue01, Les08, Lam07], to name a few), so here verification should be a lot easier and could concentrate on mathematical aspects of the problem and not on the peculiarity of the double precision numbers.

Some of these 'exact' implementations have already been verified themselves: [Les08] used `Haskell/PVS`, [OS10] used `Haskell/Coq`, and [BK08, Bau08] used `OCaml/Coq`. Unfortunately, these implementations are much slower in general than simple computations with double precision, and also much slower than other implementations for real numbers like [Mue01, Lam07]. So what we would like to have is *one* implementation of real numbers that is *exact*, *fast* and *proven to be correct* at the same time.

This motivates an ongoing project started in 2010 where we try to verify at least central aspects of the `iRRAM` software package [Mue01], which is known to be a fast `C++` implementation of exact reals numbers. Unfortunately, we do not know of any tools for direct verification of `C++` programs, so we took the following approach: Using the specification language `ACSL`, we specify the semantics of core routines of the package, then we use the tool suite `Frama-C` (with `why` and `jessie`) as well as the proof assistant `Coq` in order to verify versions of these routines that have been manually transformed from `C++` to `C`. Currently, we

[⋆] This work was partially supported by the DFG project 446 CHV 113/240/0-1.

use `frama-c-Nitrogen-20111001`, `why-2.30` and `Coq 8.3pl2`; some proofs have already partly been rewritten to use `why3-0.71` instead of `why-2.30`. Automatic provers like `Alt-Ergo` or `CVC3` could be used to verify some conditions of our test examples, but none of them could do a complete verification.

There are two objectives behind the project: the *internal* goal is just to verify correctness of the `iRRAM` package, while the *external* goal is to develop verification tools for other users of exact real arithmetic.

Our approach works in 4 levels, that are treated in parallel:

1. *core level*: arbitrarily precise floating-point numbers (mainly internal use)
2. *interval level*: interval arithmetic (mainly internal use)
3. *basic arithmetic level*: basic operations on real numbers (mainly internal use)
4. *application level*: non-basic operations and user tools (mainly external use)

As an example consider the multiplication $x * y$ of real numbers: although level 3 (basic arithmetic) is not yet fully proven, we can already use the multiplication as an exact operation on level 4 (applications). In this paper we will describe parts of this level 4, so basic real operations are assumed to be working correctly.

In section 2, we will briefly describe the background of exact real arithmetic, which will motivate why we emphasize the correct behavior of the verified routines concerning exception handling. Chapters 3 and 4 describe the tools we use, section 5 contains a detailed example and section 6 gives a short summary.

We estimate that until now about 5% of the complete package have been proven: The package consists of about 800 functions and 12000 lines of code, 30 central functions have been considered so far. As the specifications on the different levels are mutually dependent, both specifications and proofs might have to be readjusted later.

2 Exact Real Arithmetic

As the set $\mathbb{R}$ of real numbers is not countable, implementing real numbers must be significantly different from an implementation of countable sets like the natural or even rational numbers. The theoretical background here is usually called 'computable analysis' or 'type-2-theory of effectivity', see [BHW07, We00]: a real number x is represented as a sequence $(r_n)_{n \in \mathbb{N}}$ of rational numbers r_n with a known rate of convergence. Usually this convergence is expressed as a constraint '$\forall n : |x - r_n| \leq 2^{-n}$', i.e. x represented as a converging sequence $(I_n)_{n \in \mathbb{N}}$ of intervals, where $I_n := \{y \in \mathbb{R} : |y - r_n| \leq 2^{-n}\}$. 'Exact real arithmetic' now tries to use similar concepts to implement real numbers on real-world computers.

Functional languages are ideal candidates here, and there exist quite many prototypical implementations based on this programming paradigm: [OS10], [BK08], [Les08], just to name a few. Unfortunately, the performance of these approaches is usually very bad and they can only be used for academic examples. Newer functional based implementations try to improve the performance using 'stateful' functional programming (e.g., [BK08] using monads in `OCAML`).

Imperative or object-oriented programming languages, as a different paradigm, first have to be enhanced with mechanisms to work with infinite objects like sequences. This is often done by explicit construction of computation diagrams, see e.g. [Lam07]. The performance already increased dramatically, compared to the functional approaches. Unfortunately, the diagrams need a lot of memory.

Already in 1996, the iRRAM package was presented, where computation diagrams were avoided. Instead, iterations of the underlying numerical algorithm are used. This can easily be achieved using the concept of *exceptions* in C++: The algorithm under consideration is executed with interval arithmetic where each real number is represented by a single (initially quite imprecise) interval. If during the computation these intervals grow too large to get satisfactory results, an exception is thrown and the algorithm is executed with smaller intervals. This is repeated until the results are precise enough, i.e. until the algorithm finishes without throwing any exceptions. Although this idea seems to waste computation time, it turned out to be amazingly fast and the memory impact is neglectable compared to approaches previously mentioned: the iRRAM can sometimes perform a billion of dependant operations in a few seconds, where the size of the computation diagrams alone would easily amount to more than 100GB.

In this paper, the main focus is on how we deal with this aspect of exceptions.

3 Verification of C Programs

To verify the iRRAM we use a combination of the proof assistant Coq and the frameworks Frama-C and Why. Coq is a theorem prover which can be used to formulate and proof theories. Frama-C is a static analysis framework for the C programming language which for example provides tools for dead code elimination (Spare Code) but also for formal verification of C programs through a plugin called jessie. The framework Why can be seen as a general verification condition generator. It takes an annotated program as input and is able to generate verification conditions for that input for several proof assistants including Coq.

In order to verify a C program we first have to provide a formal specification of the program. For that we give a formal (predicative) description of the semantics of the C program using the so called *ANSI C Specification Language* (ACSL). As a C program usually consists of a large collection of functions, each of them has to be annotated with a so called 'function contract'.

Next, the annotated C program serves as input to the jessie plugin and then to Why, which is generating the verification conditions in Coq. Having done that we then have to prove that our program in fact is correct with respect to its specification.

4 From C++ to C

First we have to translate the C++ code to C code for the purpose of verification. In fact our specifications and even the resulting correctness proofs should easily be adaptable as soon as there is a verification tool for C++ programs (e.g. as an extension to the jessie plugin) since the semantical description should differ only slightly (e.g. to describe that a function may throw an exception).

We had to consider the following C++ concepts in order to get a reliable translation from C++ to C: (a) classes, (b) constructors and destructors, (c) operator overloading, and (d) exceptions.

(a) Classes can be translated to structs in C. Of course, classes are equipped with a collection of methods for manipulating instances of objects. Additionally, the visibility modifiers like private do not have a counterpart in C. Currently, we simply treat everything to be public.

Taking into account that methods implicitly have access to the 'this'-pointer, every method of n parameters is actually a function with $n+1$ parameters, where the first parameter is a pointer to the object itself:

```
// C++ class for real numbers
class REAL {
public:
double as_double (const int p) const;
...
```

```
// translation to C
typedef struct REAL { ... } *REAL;
...
double as_double(const REAL this,
                 const int p);
```

(b) The C language does not have constructors and destructors. Fortunately it is easy to detect where constructors are called, so we can replace them by corresponding C functions. Destructors are much harder to handle, as they are almost always called implicitly at the end of a lifetime of objects. Currently we simply ignore the destructors (and rely on a hypothetical garbage collection).

(c) Operator overloading is very useful to keep syntactical structures simple. This especially holds for mathematical software, where writing $x \cdot y \cdot z$ simply as x*y*z instead of mul(mul(x,y),z) can significantly improve readability and reduce errors at the same time. The translation, however, is tedious but quite trivial, as soon as we know the involved classes. Since C does not support overloading we have to define a function with a name of its own:

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

```
// translation to C

REAL REALREAL_mul(REAL x, REAL y);

REAL REALint_mul (REAL x, int  y);
```

(d) The exception mechanism is the most complicated aspect. For JAVA, e.g. the Krakatoa tool[MPMU04] contains a signals construct for the functions contracts to represent exceptions. C itself does not have exceptions, so ACSL does not have any support for exception handling. So currently we have no option for an easy specification of the (vital) exceptions.

We choose to model exceptions by extending the source code: A global pointer exception is introduced in the C version carrying the information about any thrown exceptions. As long as this pointer remains 0, no exception occurred. So using a multiplication z=x*y can be modeled in fact as

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

On the application level, this is sufficient for verification, as these exceptions are not caught by the application but by the runtime environment of the iRRAM (which is still unverified). To verify this runtime environment, however, we will really need to translate all aspects of the exceptions, maybe using the C functions setjmp and longjmp (that are not yet supported in Frama-C).

5 Example

As an example for a verified function we consider the power function computing 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 REALint_power(const REAL x, int n) {
  REAL y;
  { REAL tmp = REAL_from_int32(1); if (exception != 0) return 0; y=tmp;}
  for (int k=0;k<n;k=k+1)
    { REAL tmp = REALREAL_mul(y,x); if (exception != 0) return 0; y=tmp;}
  return y; }
```

The corresponding function contract in ACSL is build as follows: With requires, we express that the caller has to ensure that x points to a valid (i.e. correctly constructed) data structure for real numbers and that n is non-negative. The assigns part describes the side effects which may happen by calling the function: in our case both the result of the function as well as the exception pointer might be modified. Finally the ensures part expresses that in the program state immediately after returning from the function the result points to a valid real object and represents the n-th power of x, unless an exception was thrown.

```
/*@
requires   valid_REAL(x) && n >= 0;
assigns    \result, exception ;
ensures    exception==0  ==> ( valid_REAL(\result)
  &&  real_of_iRRAM_REAL (\result) ==  \pow(real_of_iRRAM_REAL (x),n) );
*/
REAL REALint_power(const REAL x, int n);
```

The last part of the ensures clause is very important here: This is the mathematical statement we want to prove, i.e. that the result is the n-th power of x. The function real_of_iRRAM_REAL is actually a logical defined function mapping iRRAM REALs to the ideal reals Coq knows about, and pow is mapped to Coqs power function. As every interpretation of the value of REAL data will happen via similar mappings, REAL *is* an implementation of real numbers.

What remains to be done for our example is the formal proof in Coq, for which we enhance the source code with the following loop invariant in ACSL:

```
/*@
loop invariant valid_REAL(y) && 0 <= k <= n &&
                     real_of_iRRAM_REAL (y) == \pow(real_of_iRRAM_REAL (x),k);
loop variant n-k;
*/
  for (int k=0;k<n;k=k+1)
    { REAL tmp = REALREAL_mul(y,x);  if (exception != 0) return 0; y=tmp;}
```

The `variant` expresses that the (non-negative) value $n-k$ is decreasing, so that we are able to prove that the loop eventually terminates. Having done this we were able to finish the proof of correctness for our example.

6 Summary

As the example showed, the mathematical part of the verification of `iRRAM` algorithms on the application level turns out to be quite easy, as we can rely on the exactness of the operations (unless exceptions occur) and we can use the knowledge already present in `Coq`s libraries on real numbers. Meanwhile, we are also quite certain that the conversion from `C++` to `C` could be done automatically, e.g. by some suitable pre-compilation, at least as far as we need it.

Currently, we would like to concentrate first on other parts of the whole verification: one important task here will be to replace 32- or 64-bit integers almost everywhere by a fast (and verified) datatype for $\mathbb{Z}$ using a similar concept as for real numbers: either operations are correct in the mathematical sense (so without any overflow), or an exception has to be thrown. Then our power operator would not just be correct for 32-bit numbers but for arbitrary $n \in \mathbb{Z}$.

A far goal is to address total correctness, i.e. to identify those cases where no exceptions will be thrown. This will be much harder to do, as equality of real numbers is not decidable. Additionally, out-of-memory errors will be very hard to predict, as they depend on the necessary precision in a computation.

References

[Bau08] Bauer, A.: Efficient computation with Dedekind reals. In: 5th International Conference on Computability and Complexity in Analysis, CCA 2008, Hagen, Germany, August 21-24 (2008)

[BK08] Bauer, A., Kavkler, I.: Implementing real numbers with rz. Electron. Notes Theor. Comput. Sci. 202, 365–384 (2008)

[BHW07] Brattka, V., Hertling, P., Weihrauch, K.: A Tutorial on Computable Analysis. In: Barry Cooper, S., Löwe, B., Sorbi, A. (eds.) New Computational Paradigms: Changing Conceptions of What is Computable, pp. 425–491. Springer, New York (2008)

[Lam07] Lambov, B.: Reallib: An efficient implementation of exact real arithmetic. Mathematical Structures in Computer Science 17(1), 81–98 (2007)

[Les08] Lester, D.R.: The world's shortest correct exact real arithmetic program? In: Proc. 8th Conference on Real Numbers and Computers, pp. 103–112 (2008)

[MPMU04] Marché, C., Paulin-Mohring, C., Urbain, X.: The krakatoa tool for certification of java/javacard programs annotated in jml. J. Log. Algebr. Program. 58(1-2), 89–106 (2004)

[Mue01] Müller, N.T.: The iRRAM: Exact Arithmetic in C++. In: Blank, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, pp. 222–252. Springer, Heidelberg (2001)

[OS10] O'Connor, R., Spitters, B.: A computer-verified monadic functional implementation of the integral. Theor. Comput. Sci. 411(37), 3386–3402 (2010)

[We00] Weihrauch, K.: Computable analysis: An introduction. Springer-Verlag New York, Inc. (2000)