Platform-independent model of fix-point arithmetic for verification of the standard mathematical functions

Nikolay V. Shilov[1], Dmitry A. Kondratyev[2], and Boris L. Faifel[3]

[1] Innopolis University, Innopolis, Russia
shiloviis@mail.ru
[2] A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia
apple-66@mail.ru
[3] Yuri Gagarin State Technical University of Saratov, Russia
catstail@yandex.ru

Abstract

In the talk we present axiomatic of fix-point computer arithmetics that we use in our platform-independent incremental combined approach to specification and verification of the standard functions `sqrt`, `cos` and `sin` that implement mathematical functions $\sqrt{\ }$, cos and sin. The talk will be an updated version of the talk presented (without formal publication) at *Logical Perspectives 2021: Summer School and Workshop* (June 14–19, 2021, Steklov Mathematical Institute, Moscow, Russia, https://lp2021.mi-ras.ru/workshop.html).

1 Introduction

One who has a look at verification research and practice may observe that there exist *verification in large (scale)* and *verification in small (scale)*: verification in large deals (usually) behavioral properties of large-scale complex critical systems like the *Curiosity* Mars mission [4], while verification in small addresses (usually) functional properties of small programs like computing the standard trigonometry functions [3, 2].

Our research "Platform-independent approach to formal specification and verification of standard mathematical functions" deals with *verification in small*. It may look like that it is about the same topic as [3, 2] i.e. formal verification of the standard computer functions that implement mathematical functions. But there are serious differences between [3, 2] and our research project.

Our research project is aimed onto a development of an incremental combined approach to the specification and verification of the standard mathematical functions. Platform-independence means that we attempt to design a relatively simple axiomatization of the computer arithmetic in terms of real, rational, and integer arithmetic (i.e. the fields $\mathbb{R}$ and $\mathbb{Q}$ of real and rational numbers, the ring $\mathbb{Z}$ of integers) but don't specify neither base of the computer arithmetic, nor a format of numbers' representation. Incrementality means that we start with the most straightforward specification of the simplest easy to verify algorithm in real numbers and finish with a realistic specification and a verification of an algorithm in computer arithmetic. We call our approach combined because we start with a manual (pen-and-paper) verification of some selected algorithm in real numbers, then use these algorithm and verification as a draft and proof-outlines for the algorithm in computer arithmetic and its manual verification, and finish with a computer-aided validation of our manual proofs with some proof-assistant system (to avoid appeals to "obviousness" that are very common in human-carried proofs).

2 A Brief of the Approach Results

In our approach we start with easy-to-verify Hoare total correctness assertions [1] for logical specification of imperative algorithms that implements the computer functions in "ideal" real arithmetic, and finish with computer-aided verification of the computer functions in computer fix-point arithmetic. Full details of our approach can be found in [6, 5].

In a journal (Russian) paper [6] an *adaptive* imperative algorithm implementing the Newton-Raphson method for a square root function $\sqrt{}$ has been specified by total correctness assertions and verified manually using Floyd-Hoare approach in both fix-point and floating-point arithmetics; the post-condition of the total correctness assertion states that the final overall truncation error is not greater that $2ulp$ where ulp is *Unit in the Last Place* — the unit of the last meaningful digit.

The paper [6] has reported also two steps towards computer-aided validation and verification of the used adaptive algorithm. In particular, an implementation of a fix-point data type according to the axiomatization can be found at `https://bitbucket.org/ainoneko/lib_verify/src/`; ACL2 computer-carried proofs of (i) the consistency of the computer fix-point arithmetic axiomatization, and (ii) the existence of a look-up table with initial approximations for $\sqrt{}$ are available at `https://github.com/apple2-66/c-light/tree/master/experiments/square-root`.

In a work-in-progress electronic preprint [5] platform-independent and incremental approach is applied for manual (pen-and-paper) verification (using Floyd-Hoare approach) of the computer functions `cos` and `sin` (that implement mathematical trigonometric functions cos and sin) for fix-point argument values in the rage $[-1, 1]$ (in radian measure); the post-condition of the total correctness assertion states that the final overall truncation error is not greater that $\frac{3n \times ulp}{2(1-ulp)}$ where $n = O(|\ln \varepsilon|)$ and $\varepsilon > 0$ is user-defined computational error (in ideal real arithmetic).

3 Fix-point Arithmetic

Below we present version axiomatization (modulo "ideal" arithmetic of real, rational and integer numbers) of a computer (platform-independent) fix-point arithmetic data type as in [6]. (Please remark that we explicitly admit that there may be several different fix-point data types simultaneously.)

A fix-point data-type (with Gaussian rounding) $\mathbb{D}$ satisfies the following axioms.

- The set of values $Val_{\mathbb{D}}$ is a finite set of rational numbers $\mathbb{Q}$ (and reals $\mathbb{R}$) such that
  - it contains the least $\inf_{\mathbb{D}} < 0$ and the largest $\sup_{\mathbb{D}} > 0$ elements,
  - altogether with
    - all rational numbers in $[\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$ with a step $\delta_{\mathbb{D}} > 0$,
    - all integers $Int_{\mathbb{D}}$ in the range $[-\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$.
- Admissible operations include machine addition $\oplus$, subtraction $\ominus$, multiplication $\otimes$, division $\oslash$, integer rounding up $\lceil\ \rceil$ and down $\lfloor\ \rfloor$.

  Machine addition and subtraction. If the exact result of the standard mathematical addition (subtraction) of two fix-point values falls within the interval $[\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$, then machine addition (subtraction respectively) of these arguments equals to the result of the mathematical operation (and notation $+$ and $-$ is used in this case).

Machine multiplication and division. These operations return values that are nearest in $Val_{\mathbb{D}}$ to the exact result of the corresponding standard mathematical operation: for any $x, y \in Val_{\mathbb{D}}$

- if $x \times y \in Val_{\mathbb{D}}$ then $x \otimes y = x \times y$;
- if $x/y \in Val_{\mathbb{D}}$ then $x \oslash y = x/y$;
- if $x \times y \in [\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$ then $|x \otimes y - x \times y| \leq \delta_{\mathbb{D}}/2$;
- if $x/y \in [\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$ then $|x \oslash y - x/y| \leq \delta_{\mathbb{D}}/2$;

Integer rounding up and down are defined for all values in $Val_{\mathbb{D}}$.

- Admissible binary relations include all standard equalities and inequalities (within $[\inf_{\mathbb{D}}, \sup_{\mathbb{D}}]$) denoted in the standard way $=$, $\neq$, $\leq$, $\geq$, $<$, $>$.

Finally let us mention that we implement (prototype) our (platform-independent) fix-point (an floating-point) arithmetic data type. Of course, currently, tools to increase the precision of fix/floating-point computations are available in many industrial platforms (C / C ++, Java, Python), but in the above languages, data of a non-standard numeric type are represented by objects, and it takes effort to *link* them with standard numeric types. Instead, we design and implement a simple programming language with the following built-in numeric types — fixed-point numbers (parameterized by user-specified rational $\delta_{\mathbb{D}} > 0$ — Unit in the Last Place *ulp* — and the least $\inf_{\mathbb{D}} < 0$ and the largest $\sup_{\mathbb{D}} > 0$ elements), rational numbers, floating-point numbers with a definable mantissa size [7].

References

[1] K.R. Apt, F.S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 2009.

[2] G. Grohoski. Verifying oracle's sparc processors with acl2. slides of the invited talk for 14th international workshop on the acl2 theorem prover and its applications. http://www.cs.utexas.edu/users/moore/acl2/workshop-2017/slides-accepted/grohoski-ACL2_talk.pdf, 2017.

[3] J. Harrison. Formal verification of floating point trigonometric functions. *Lecture Notes in Computer Science*, 1954:217–233, 2000.

[4] G.J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, 2014.

[5] N. V. Shilov, B. L. Faifel, S. O. Shilova, and A. V. Promsky. Towards platform-independent specification and verification of the standard trigonometry functions. arXiv:1901.03414, https://arxiv.org/abs/1901.03414, 2019.

[6] N. V. Shilov, D. A. Kondratyev, I. S. Anureev, E. V. Bodin, and A. V. Promsky. Platform-independent specification and verification of the standard mathematical square root function. *Modeling and Analysis of Information Systems*, 25(6):637–666, 2018. (In Russian. English translation: Automatic Control and Computer Sciences, 2019, Vol. 53, No. 7, pp. 595–616.).

[7] B. L. Faifel, N. V. Shilov. A programming language and environment for floating point computations with an adjustable mantissa size and rational numbers. Submitted to *Problems Of Control, Information Processing And Transmission* (CIPT-2021) September 9 – 10, 2021, Saratov, Russia. (10 p. Manuscript. In Russian.)