Formal verification of numerical programs: from C annotated programs to Coq proofs

Sylvie Boldo

Formal verification of numerical programs: from C annotated programs to Coq proofs*

Sylvie Boldo

INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
Sylvie.Boldo@inria.fr

Abstract. Numerical programs may require a high level of guarantee. This can be achieved by applying formal methods, such as machine-checked proofs. But these tools handle mathematical theorems while we are interested in C code. To achieve this high level of confidence on C programs, we use a chain of tools: Frama-C, its Jessie plugin, Why and Coq. This requires the C program to be annotated: this means that each function must be precisely specified, and we will prove the correctness of the program by proving both that it meets its specifications and that it does not fail. Examples will be given to illustrate the features of this approach.

1 Introduction

Given a program using floating-point arithmetic, it is pretty hard to know the final rounding error of the result. We are interested in proving numerical programs with a very high level of guarantee by using formal methods.

Each floating-point result is a correct rounding of the exact real value for all basic operations (addition, subtraction, multiplication, division and square root). This property is defined in the IEEE-754 standard [1] and all modern processors comply with it. Nevertheless, even if each computation is correct, *i.e.* the best possible, there is no guarantee that the final result after many such computations is still accurate.

Static analysis is an approach for checking a program without running it. Deductive verification techniques which perform static analysis of code, rely on the ability of theorem provers to check validity of formulas in first-order logic or even more expressive logics. They usually come with expressive specification languages such as JML [2, 3] for Java, ACSL [4] for C, Spec# [5] for C#, etc. to specify the requirements.

For automatic analysis of floating-point codes, there exist several methods for bounding the final error of a program, including interval arithmetic, forward and backward analysis [6, 7]. Another approach is abstract interpretation based static analysis, that includes Astrée [8, 9] and Fluctuat [10, 11].

* This work was funded by the Foſt (ANR-08-BLAN-0246-01) project of the French national research organization (ANR).

Floating-point arithmetic has been formalized using deductive formal methods since 1989 in order to formally prove hardware components or algorithms [12–14]. We machine-check all proofs using the Coq proof checker [15]. We use a high-level formalization of floating-point numbers [16, 17].

There exist few works on specifying and proving behavioral properties of floating-point programs in deductive verification systems. A work on floating-point in JML for Java is presented in 2006 by Leavens [18]. Another proposal has been made in 2007 by Boldo and Filliâtre [19]. Ayad and Marché extended this to increase genericity and handle exceptional behaviors [20]. Boldo and Nguyen extended this to handle multiple architectures and compilers [21].

2 Tools Chain

We start from annotated C programs using ACSL [4]. Each function is annotated with pre-conditions (what the function `requires` from the inputs) and post-conditions (what the function `ensures` at its end). The annotations and requirements (pointer dereferencing for example) are then transformed into proof obligations that have to be solved by proof assistants or decision procedures.

The Frama-C framework has floating-point annotations based on [19] that allow to specify numerical programs. More precisely, each floating-point number has a ghost value called exact which does not suffer from rounding. This real value is then computed with the same operations as the float value except that the ghost operation is exact. The macro `round_error`(f) is then used for

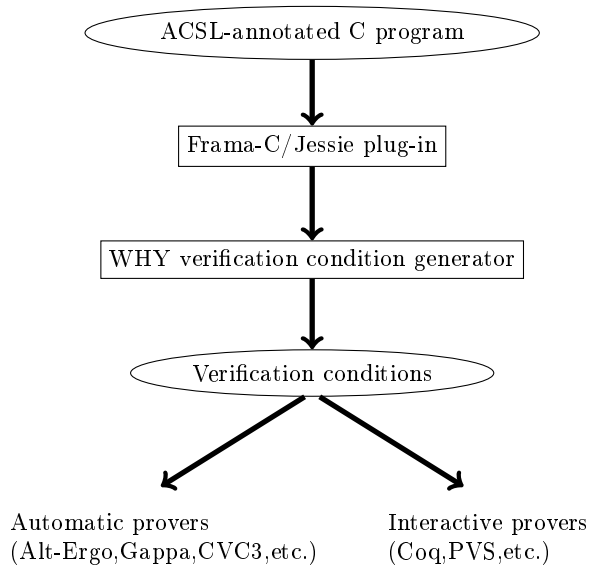

Fig. 1. Chain of tools: from the C program to the proof obligations.

$|f - \text{exact}(f)|$. For example, to compute a naive exponential by the polynomial evaluation of `1+x+x*x/2`, the corresponding exact value is $1 + x + \frac{x^2}{2}$ (with mathematical operations). Inside the annotations, all computations are exact.

We then use the Frama-C platform[1] associated with the Jessie plugin that uses Why [22]. This chain is described in Figure 2. The C code is given to the Jessie plugin of Frama-C, it creates a Why file of proof obligations (theorems to prove) that can be translated in order to be given to either automatic or interactive provers.

3 Examples

The full code of all these examples (and more) and their proofs are available on `http://www.lri.fr/~sboldo/research.html`.

3.1 Sterbenz subtraction

This function computes the exact subtraction if the inputs are near enough one to another [23]. Note that the division and multiplication inside the annotations are exact.

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

3.2 Veltkamp/Dekker algorithm

This function computes the exact error of the multiplication [24,25] with only floating-point operations (and no FMA). There are also underflow restrictions and overflow restrictions so that no infinity will be created [26].

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @           \abs(x) <= 0x1.p995 &&
  @           \abs(y) <= 0x1.p995 &&
  @           \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                 ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  int i;
  C=1;
  /*@ loop invariant C== \pow(2., i) && 0 <= i <= 27;
    @ loop variant (27-i); */
  for (i=0; i<27; i++)
    C*=2;
  C++;
  /*@ assert C == \pow(2.,i) + 1. && i==27; */
```

[1] `http://frama-c.cea.fr/`

```
  px=x*C;
  qx=x-px;
  hx=px+qx;
  tx=x-hx;

  py=y*C;
  qy=y-py;
  hy=py+qy;
  ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

3.3 Kahan algorithm for an accurate discriminant

This function computes an accurate discriminant using Kahan's algorithm [27]. The result is proved correct within 2 ulps. Overflow and underflow restrictions are given [28]. We needed an axiomatic to ensure the definition of ulp is the proper one that is omitted here.

```
/*@ requires
  @      (b==0.    || 0x1.p-916 <= \abs(b*b)) &&
  @      (a*c==0.  || 0x1.p-916 <= \abs(a*c)) &&
  @      \abs(b) <= 0x1.p510 && \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @      \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.   || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */


double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

3.4 Wave Equation Resolution Scheme

This function is a finite difference numerical scheme for the resolution of the one-dimensional acoustic wave equation [29, 30]. Its rounding error bound requires a high-level predicate defined in Coq as containing a property "there exists a function from $\mathbb{Z}$ to $\mathbb{R}$ such that...". For simplicity, initialization functions are omitted. Note that these proofs are not fully done as they require the scheme not to diverge.

```
/*@ axiomatic dirichlet {
  @  predicate analytic_error{L}
  @           (double **p, integer ni, integer i, integer k, double a)
  @           reads p[..][..];  } */

/*@ requires ni >= 2 && nk >= 2
  @          && l != 0
  @          && dx > 0. && dt > 0.  && v > 0.
  @          && \exact(dx) > 0. && \exact(dt) > 0.
  @          && \exact(v)==v
  @          && \abs(\exact(dx) - dx) / dx <= 0x1.p-53
  @          && \abs(\exact(dt) - dt) / dt <= 0x1.p-51
  @          && 3./5. <= \exact(dt)/\exact(dx) * \exact(v) <= 1-0x1.p-50
  @          && 0x1.p-1000 <= v <= 0x1.p1000
  @          && ni <= 0x1.p64 && nk <= 4194304
  @          && \exact(dx) <= 1;
  @
  @ ensures \forall int i; \forall int k;
  @     0 <= i <= ni ==> 0 <= k <= nk ==>
  @     \round_error(\result[i][k]) <= 85./2*0x1.p-52*(k+1)*(k+2); */

double **forward_prop(int ni, int nk, double dx, double dt, double v,
                      double xs, double l) {
  double **p;
  int i, k;
  double a1, a, dp;

  a1 =  dt/dx*v;
  a  = a1*a1;
  /*@ assert 1./4 <= a <= 1 && 0 < \exact(a) <= 1 &&
    @         \round_error(a) <= 0x1.p-49; */

  p = array2d_alloc(ni+1, nk+1);

  p[0][0]=0.;
  /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,0,a);
    @ loop variant ni-i; */
  for (i=1; i<ni; i++) {
    p[i][0] = p_zero(xs, l, i*dx);
  }
  p[ni][0] =0.;

  p[0][1] = 0.;
  /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,1,a);
    @ loop variant ni-i; */
  for (i=1; i<ni; i++) {
    dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
    p[i][1] = p[i][0] + 0.5*a*dp;
  }
  p[ni][1] = 0.;

  /* propagation = time loop */
  /*@ loop invariant 1 <= k <= nk && analytic_error(p,ni,ni,k,a);
    @ loop variant nk-k; */
  for (k=1; k<nk; k++) {
    p[0][k+1] = 0.;

    /* time iteration = space loop */
    /*@ loop invariant 1 <= i <= ni && analytic_error(p,ni,i-1,k+1,a);
      @ loop variant ni-i; */
    for (i=1; i<ni; i++) {
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

4 Conclusion

We have proved that a very high guarantee on numerical programs is achievable. A given specified program can be proved in its deeper details using Coq and therefore ascertaining its correctness. We have a usable chain from the annotated program to the mathematical theorems that is able to secure a program both from its rounding errors and from its other possible failures (pointer dereferencing, out-of-bound array accesses...). For example, there are 84 safety proof obligations for the example of Section 3.4 that have to do with memory access, variant decrease, overflow, and precondition for function call.

This approach suffers from several drawbacks. The first one is that the specifications must be given. We prove the specifications but we do not infer them at all. This could be solved by using abstract interpretation or by using any other tools to infer specifications (for example turned into another Frama-C plugin). We then have to prove them to ensure their correctness. Therefore, we do not rely on the correctness of the external tool.

The second drawback is that all those examples need interactive proofs in Coq. This can be done on small programs, but automatizations is essential to spread those techniques. A first way is to use Gappa [31, 32] as an automatic prover output of Why. This is quite convenient but it may mean a drop of guarantee. Hopefully, Gappa is able to produce a Coq proof and we may use a Gappa tactic inside Coq to benefit from Gappa's automations inside interactive proofs [33].

A never-ending perspective is to find cunning techniques to better bound the rounding errors, especially when they compensate. A technique that states the analytical error has been developed in [28] for the example of Section 3.4 where usual methods gave an error proportional to 2^k that was cut down to k^2. This technique of the analytical error and precise floating-point error cancellation coming with its formal proof is new. The reason is that it requires very generic specifications as the loop invariant needs to be logically defined: it states there exists a function that has such and such property. And ACSL allows us to express such a high-level property on a C program. We then use Coq as a back-end to formally check the specifications. This genericity is an advantage compared to automatic methods that cannot express our loop invariant.

References

1. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (2008)
2. : JML-Java Modeling Language `www.jmlspecs.org`.
3. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (June 2005) 212–232
4. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (2008) `http://frama-c.cea.fr/acsl.html`.
5. Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# Programming System: An Overview, Springer (2004) 49–69

6. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1963)
7. Higham, N.J.: Accuracy and stability of numerical algorithms. SIAM (2002)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP. Number 3444 in Lecture Notes in Computer Science (2005) 21–30
9. Monniaux, D.: Analyse statique : de la théorie à la pratique. Habilitation to direct research, Université Joseph Fourier, Grenoble, France (June 2009)
10. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In Yi, K., ed.: SAS. Volume 4134 of LNCS., Springer (2006) 18–34
11. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., VÃľdrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: FMICS. Volume 5825 of LNCS., Springer (2009) 53–69
12. Carreño, V.A., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications, Aspen Grove, UT (September 1995)
13. Russinoff, D.M.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS Journal of Computation and Mathematics **1** (1998) 148–200
14. Harrison, J.: Formal verification of floating point trigonometric functions. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, Austin, Texas (2000) 217–233
15. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
16. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland (2001) 169–184
17. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
18. Leavens, G.T.: Not a number of floating point problems. Journal of Object Technology **5**(2) (2006) 75–83
19. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194
20. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In Giesl, J., Hähnle, R., eds.: Fifth International Joint Conference on Automated Reasoning. LNAI, Edinburgh, Scotland, Springer (July 2010)
21. Boldo, S., Nguyen, T.M.T.: Hardware-independent proofs of numerical programs. In Muñoz, C., ed.: Proceedings of the Second NASA Formal Methods Symposium. Number NASA/CP-2010-216215 in NASA Conference Publication, Washington D.C., USA (April 2010) 14–23
22. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification (CAV). Volume 4590 of LNCS., Springer (July 2007)
23. Sterbenz, P.H.: Floating point computation. Prentice Hall (1974)
24. Veltkamp, G.W.: Algolprocedures voor het berekenen van een inwendig product in dubbele precisie. RC-Informatie 22, Technische Hogeschool Eindhoven (1968)
25. Dekker, T.J.: A floating point technique for extending the available precision. Numerische Mathematik **18**(3) (1971) 224–242

26. Boldo, S.: Pitfalls of a full floating-point proof: Example on the formal proof of the veltkamp/dekker algorithms. In: Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR), Seattle, USA (August 2006) 52–66
27. Kahan, W.: On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic. World-Wide Web document (November 2004)
28. Boldo, S.: Kahan's algorithm for a correct discriminant computation at last formally proven. IEEE Transactions on Computers **58**(2) (February 2009) 220–225
29. Boldo, S.: Floats & Ropes: a case study for formal numerical program verification. In: 36th International Colloquium on Automata, Languages and Programming. Volume 5556 of Lecture Notes in Computer Science - ARCoSS., Rhodos, Greece, Springer (July 2009) 91–102
30. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In: Proceedings of the first Interactive Theorem Proving Conference. LNCS, Edinburgh, Scotland, Springer (July 2010)
31. de Dinechin, F., Lauter, C., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France (2006) 1318–1322
32. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software **37**(1) (2009)
33. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. Volume 5625 of Lecture Notes in Artificial Intelligence., Grand Bend, Canada, Springer (July 2009) 59–74