Combining Coq and Gappa for Certifying Floating-Point Programs

Sylvie Boldo Jean-Christophe Filliâtre Guillaume Melquiond

Proval, Laboratoire de Recherche en Informatique
INRIA Saclay–ÎdF, CNRS, Université Paris Sud

ANR-05-BLAN-0281-04 "CerPAN"
ANR-08-BLAN-0246-01 "F∮ST"

July 6, 2009

Motivation

Floating-point arithmetic is efficient, but FP numbers have

- a limited range ($\longrightarrow$ exceptional behaviors),
- a limited precision ($\longrightarrow$ inaccurate results).

Motivation

Floating-point arithmetic is efficient, but FP numbers have

- a limited range ($\rightarrow$ exceptional behaviors),
- a limited precision ($\rightarrow$ inaccurate results).

Accumulating 0.1 during 864000 iterations:

```
float f = 0;
for (int i = 0; i < 10 * 60 * 60 * 24; ++i)
  f = f + 0.1f;
printf("f = %g\n", f);
```

Result: $f = 87145.8$. Expected value: 86400. Error: $+0.86\%$

Motivation

Floating-point arithmetic is efficient, but FP numbers have

- a limited range (→ exceptional behaviors),
- a limited precision (→ inaccurate results).

Accumulating 0.1 during 864000 iterations:

```
float f = 0;
for (int i = 0; i < 10 * 60 * 60 * 24; ++i)
  f = f + 0.1f;
printf("f = %g\n", f);
```

Result: $f = 87145.8$. Expected value: 86400. Error: $+0.86\%$

Consequence:

Gulf War 1, a Patriot system had not been rebooted for 48 hours, it fails to intercept a Scud missile: 28 casualties.

Some Other Numerical Failures

- 1983, truncation in computing an index of Vancouver Stock Exchange makes it to drop to half its value.

Some Other Numerical Failures

- 1983, truncation in computing an index of Vancouver Stock Exchange makes it to drop to half its value.
- 1987, the inflation in UK is computed with a rounding error: pensions are off by £100M for 21 months.

Some Other Numerical Failures

- 1983, truncation in computing an index of Vancouver Stock Exchange makes it to drop to half its value.
- 1987, the inflation in UK is computed with a rounding error: pensions are off by £100M for 21 months.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.

Some Other Numerical Failures

- 1983, truncation in computing an index of Vancouver Stock Exchange makes it to drop to half its value.
- 1987, the inflation in UK is computed with a rounding error: pensions are off by £100M for 21 months.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.

Some Other Numerical Failures

- 1983, truncation in computing an index of Vancouver Stock Exchange makes it to drop to half its value.
- 1987, the inflation in UK is computed with a rounding error: pensions are off by £100M for 21 months.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
- 2007, Excel displays 77.1×850 as 100000.

  Note: only 12 floating-point inputs fail.
  Probability to uncover them by random testing: 10^{-18}.

Verification of Numerical Programs

Floating-point programs require a lot of care when designing them.
But pen-and-paper certification is long, tedious, and error-prone.

Verification of Numerical Programs

Floating-point programs require a lot of care when designing them.
But pen-and-paper certification is long, tedious, and error-prone.

Some existing tools:

1. Caduceus/Why: a proof-obligation generator for annotated C,
2. Coq: a formal system with a library on FP arithmetic,
3. Gappa: an automated prover for FP properties.

Verification of Numerical Programs

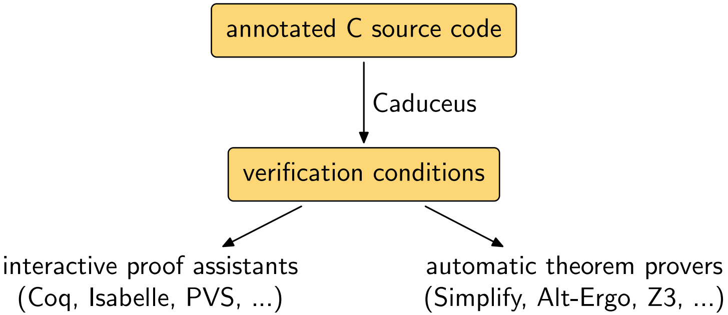
Floating-point programs require a lot of care when designing them.
But pen-and-paper certification is long, tedious, and error-prone.

Some existing tools:

1. Caduceus/Why: a proof-obligation generator for annotated C,
2. Coq: a formal system with a library on FP arithmetic,
3. Gappa: an automated prover for FP properties.

Objective: combine Caduceus, Coq, and Gappa,
to ease the certification of numerical C code.

Handling C Programs: Caduceus/Why



- Takes pre/post conditions and (in)variants from comments.
- Computes weakest preconditions from the C code.
- Sends the resulting proof obligations to various provers.

Formally Verifying Proof Obligations: Coq

- Generic proof system:
  provides high-level construct, e.g., induction.
- A comprehensive library on floating-point arithmetic.
- No automation, especially for FP properties:
  even trivialities have to be proved by hand.

Handling Floating-Point Properties: Gappa

Gappa: tool for checking properties on real-valued expressions and generating formal proofs of these properties.

Handling Floating-Point Properties: Gappa

Gappa: tool for checking properties on real-valued expressions and generating formal proofs of these properties.

Property (Correctness of integer division on Itanium)

Given a and b positive integers less than 65536,

assuming y_0 is a 11-bit FP number equal to $b^{-1} \pm 0.2\%$,
if $q_0 = a \times y_0$ and $e_0 = (1 + 2^{-17}) - b \times y_0$ and $q_1 = q_0 + e_0 \times q_0$,

then the reals q_0, e_0, and q_1 fit in binary80 FP numbers,
and the relative error between q_1 and a/b is in $[0, a^{-1})$.

Corollary: *q_1 is computable and $\lfloor q_1 \rfloor = \lfloor a/b \rfloor$.*

Handling Floating-Point Properties: Gappa

Gappa: tool for checking properties on real-valued expressions and generating formal proofs of these properties.

Property (Correctness of integer division on Itanium)

Given a and b positive integers less than 65536,

assuming y_0 is a 11-bit FP number equal to $b^{-1} \pm 0.2\%$,
if $q_0 = a \times y_0$ and $e_0 = (1 + 2^{-17}) - b \times y_0$ and $q_1 = q_0 + e_0 \times q_0$,

then the reals q_0, e_0, and q_1 fit in binary80 FP numbers,
and the relative error between q_1 and a/b is in $[0, a^{-1})$.

Corollary: *q_1 is computable and $\lfloor q_1 \rfloor = \lfloor a/b \rfloor$.*

Pen-and-paper proof: several pages. Gappa proof: 3 lines.

Outline

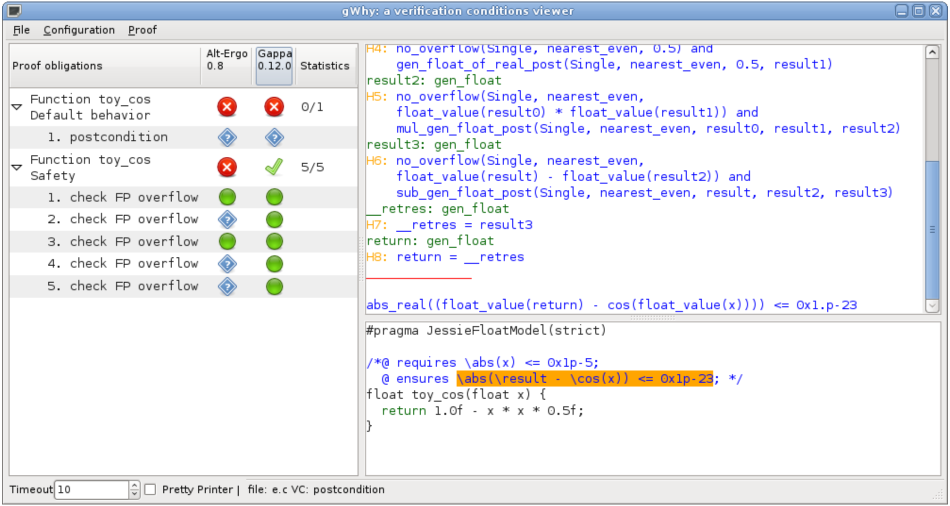
1. Introduction
2. Certifying Numerical Code
3. The Gappa Tactic
4. Conclusion

Example: a Toy Implementation of Cosine

Example: a floating-point implementation of cosine around zero.

```
/*@ requires \abs(x) <= 0x1p-5;
  @ ensures  \abs(\result - \cos(x)) <= 0x1p-23;
 */
float toy_cos(float x) {
  return 1.f - x * x * .5f;
}
```

Note: Implementation is trivial, postcondition is not.



Note: automated provers discharge all the POs related to safety, so exceptional behaviors are proved not to occur.

What about the accuracy of the result?

Using Coq for the Remaining Proof Obligation

Property (Accuracy of the computed value)

Given x *a finite floating-point number such that* $|x| \leq \frac{1}{32}$,

$$|(\circ(1) \ominus ((x \otimes x) \otimes \circ(5/10))) - \cos x| \leq 2^{-23}.$$

($\ominus$*,* $\otimes$*,* $\circ$ *are binary32 floating-point operators.)*

Using Coq for the Remaining Proof Obligation

Property (Accuracy of the computed value)

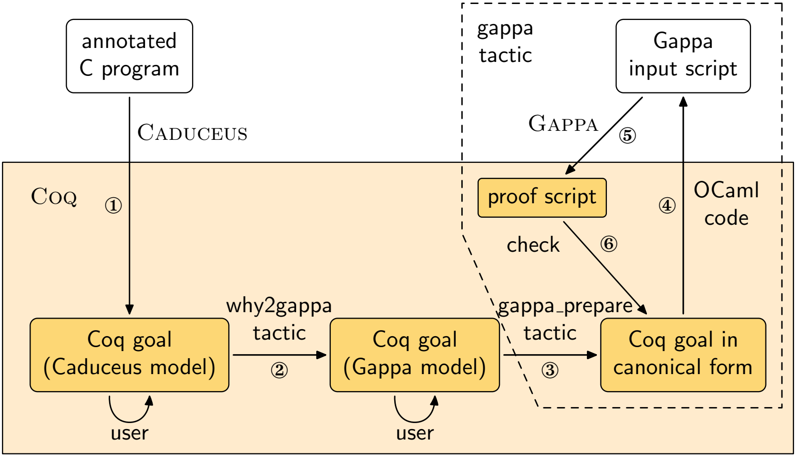
Given x a finite floating-point number such that $|x| \leq \frac{1}{32}$,

$$|(\circ(1) \ominus ((x \otimes x) \otimes \circ(5/10))) - \cos x| \leq 2^{-23}.$$

($\ominus$, $\otimes$, $\circ$ are binary32 floating-point operators.)

```
Proof.
  intros. why2gappa.
  assert ( Rabs ((1 - (x*x) * (5/10)) - cos x)
                 <= 7/134217728 )%R
    by interval with (i_bisect_diff x).
  gappa.
Qed.
```

Calling Gappa from Coq



The gappa Tactic

Property (Example)

Given x and y real numbers such that $\lfloor x \rfloor = \lfloor y \rfloor$,

$$2 \leq \sqrt{4 + (x - y)^2} \leq \frac{9}{4}$$

The gappa Tactic

Property (Example)

Given x and y real numbers such that $\lfloor x \rfloor = \lfloor y \rfloor$,

$$2 \leq \sqrt{4 + (x - y)^2} \leq \frac{9}{4}$$

Coq description and proof:

```
Goal forall x y,
  floor x = floor y ->
  2 <= sqrt (4 + (x - y) * (x - y)) <= 9/4.
Proof.
gappa.
Qed.
```

Reification of the Goal into Inductive Objects

Goal is changed into a β-convertible term:

```
convert_goal  (* evaluable conversion function *)

  (y :: x :: nil) (* list of unknown expressions  *)
  (floor :: nil)  (* list of recognized functions *)

  (raEq     (* 1st hypothesis is an equality *)
     (reApply 1 (reUnknown 2))   (* floor x *)
     (reApply 1 (reUnknown 1))   (* floor y *)
    :: nil, (* no other hypotheses *)

  raBound   (* goal is an enclosure *)
    (Some (reInteger 2))  (* left bound *)
    (reUnary uoSqrt       (* enclosed expr *)
       (reBinary boAdd (reInteger 4)
          (reBinary boMul ... ...)))
    (Some (reBinary boDiv (* right bound *)
             (reInteger 9) (reInteger 4))))
```

Transformation into a Gappa-Compatible Goal

Property (Before)

$$\lfloor x \rfloor = \lfloor y \rfloor \Longrightarrow 2 \leq \sqrt{4 + (x-y)^2} \leq \frac{9}{4}$$

Property (After)

$$\lfloor x \rfloor - \lfloor y \rfloor \in [0, 0] \Longrightarrow \sqrt{4 + (x-y)^2} \in [1 \cdot 2^1, 9 \cdot 2^{-2}]$$

Note: expressions are enclosed in intervals with FP bounds.

Reification + β-reduction = reflection
$\Rightarrow$ one single theorem suffices to produce a goal suitable for Gappa.

Calling Gappa

Coq generates an input file from the reified goal:

```
{ fixed<0,dn>(x) - fixed<0,dn>(y) in [0b0, 0b0] ->
sqrt(4 + (x - y) * (x - y)) in [1b1, 9b-2] }
```

and sends it to Gappa,

Calling Gappa

Coq generates an input file from the reified goal:

```
{ fixed<0,dn>(x) - fixed<0,dn>(y) in [0b0, 0b0] ->
sqrt(4 + (x - y) * (x - y)) in [1b1, 9b-2] }
```

and sends it to Gappa, which verifies it and generates a Coq proof:

```
(fun (_x : R) (_y : R) =>
let f1 := Float2 (0) (0) in
let i1 := makepairF f1 f1 in
let r2 := float2R ((rounding_fixed roundDN (0)) _x) in
(* ... 96 other lines ... *)
```

Calling Gappa

Coq generates an input file from the reified goal:

```
{ fixed<0,dn>(x) - fixed<0,dn>(y) in [0b0, 0b0] ->
sqrt(4 + (x - y) * (x - y)) in [1b1, 9b-2] }
```

and sends it to Gappa, which verifies it and generates a Coq proof:

```
(fun (_x : R) (_y : R) =>
let f1 := Float2 (0) (0) in
let i1 := makepairF f1 f1 in
let r2 := float2R ((rounding_fixed roundDN (0)) _x) in
(* ... 96 other lines ... *)
```

This λ-term is then loaded and checked by Coq.

The type of this term is exactly the transformed goal;
this is a complete formal proof.

Application: Solving 1D Wave Equation

```
/*@ invariant 1 <= k <= nk
        && analytic_error(p,ni,ni,k,a) */
for (k=1; k<nk; k++) {
  p[0][k+1] = 0.;
  /*@ invariant 1 <= i <= ni
          && analytic_error(p,ni,i-1,k+1,a) */
  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.;
}
```

- Coq proof reduced from 735 to 10 lines by using Gappa.
- Better specification found for the program!

Application: Accurate Discriminant

```
/*@ requires xy == round(x*y) &&
  @   (x*y == 0 || 2^^(-969) <= |x*y|) &&
  @   |x| <= 2^^995 && |y| <= 2^^995 && |x*y| <= 2^^1022
  @ ensures \result == x*y-xy
  @ */
double exactmult(double x, double y, double xy);

/*@ requires
  @     (b == 0   || 2^^(-916) <= |b*b|) &&
  @     (a*c == 0 || 2^^(-916) <= |a*c|) &&
  @     |b| <= 2^^510 && |a| <= 2^^995 && |c| <= 2^^995 &&
  @     |a*c| <= 2^^1021
  @ ensures \result == 0 || |\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 = exactmult(b,b,p);
    dq = exactmult(a,c,q);
    d = (p-q) + (dp-dq);
  }
  return d;
}
```

- Coq proof reduced from 420 to 35 lines by using Gappa.
- Downside: time for replaying the proof script is doubled.

Conclusion

1. Coq is a formal system able to tackle any kind of PO,
   - but its use is tedious, especially for FP programs.
2. Gappa is an automated prover for computer arithmetic,
   - but it only handles (basic) arithmetic properties.

Conclusion

1. Coq is a formal system able to tackle any kind of PO,
   - but its use is tedious, especially for FP programs.
2. Gappa is an automated prover for computer arithmetic,
   - but it only handles (basic) arithmetic properties.

The gappa tactic makes it possible to call Gappa from Coq.

- Considerably reduces the time needed for FP proofs.
- Generates a proper Coq proof (no assumptions).
- Shipped in Coq 8.2.

Questions?

Web: http://www.lri.fr/~melquion/
Email: guillaume.melquiond@inria.fr