Automated Methods for Verifying Floating-point Algorithms

Guillaume Melquiond

Inria Saclay–Île-de-France

LRI, Université Paris Sud, CNRS

2014-02-06

Why Floating-point Arithmetic?

The real world is much more continuous than one could hope,
so real numbers tend to creep in all the applications.

Why Floating-point Arithmetic?

The real world is much more continuous than one could hope, so real numbers tend to creep in all the applications.

How to compute with them?

- Use a subset, e.g. rational or algebraic numbers.

Why Floating-point Arithmetic?

The real world is much more continuous than one could hope, so real numbers tend to creep in all the applications.

How to compute with them?

- Use a subset, e.g. rational or algebraic numbers.
- Compute with arbitrary precision.

Why Floating-point Arithmetic?

The real world is much more continuous than one could hope, so real numbers tend to creep in all the applications.

How to compute with them?

- Use a subset, e.g. rational or algebraic numbers.
- Compute with arbitrary precision.
- Approximate operations, e.g. floating-point numbers.

Why Floating-point Arithmetic?

The real world is much more continuous than one could hope, so real numbers tend to creep in all the applications.

How to compute with them?

- Use a subset, e.g. rational or algebraic numbers.
- Compute with arbitrary precision.
- Approximate operations, e.g. floating-point numbers.

Speed of FP operations is high and deterministic,
but all bets are off with respect to the quality of FP results:
precision is known, but accuracy is not.

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:
   - preconditions of functions are still satisfied;

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:
   - preconditions of functions are still satisfied;
   - control-flow changes are innocuous;

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:
    - preconditions of functions are still satisfied;
    - control-flow changes are innocuous;
    - accuracy of the computed values is good enough.

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:
   - preconditions of functions are still satisfied;
   - control-flow changes are innocuous;
   - accuracy of the computed values is good enough.

Verifying Floating-point Algorithms

People tend to verify FP algorithms in two steps:

1. Prove correctness assuming that all operators are infinitely-precise.
2. Check that limited precision does not have much impact:
   - preconditions of functions are still satisfied;
   - control-flow changes are innocuous;
   - accuracy of the computed values is good enough.

There exist numerous automated tools for this job.
But what if your algorithm is intricate or you need a formal proof?

Scope and Constraints

Scope

- real numbers and basic operators: $+$, $\times$, $\div$, $\sqrt{\cdot}$;
- radix-2 fixed- and FP arithmetic (no multi-precision);
- logical formulas (no control flow).

Scope and Constraints

Scope

- real numbers and basic operators: $+$, $\times$, $\div$, $\sqrt{\cdot}$;
- radix-2 fixed- and FP arithmetic (no multi-precision);
- logical formulas (no control flow).

Features

- compute range and format of expressions;
- bound forward errors.

Scope and Constraints

Scope

- real numbers and basic operators: $+$, $\times$, $\div$, $\sqrt{\cdot}$;
- radix-2 fixed- and FP arithmetic (no multi-precision);
- logical formulas (no control flow).

Features

- compute range and format of expressions;
- bound forward errors.

Constraints

- handle complicated formulas (possibly with user help),
- generate Coq proofs that fit into Flocq's formalism.

Outline

Why is FP Arithmetic Amenable to Formal Proof?

IEEE-754 standard for FP arithmetic

Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result.

Why is FP Arithmetic Amenable to Formal Proof?

IEEE-754 standard for FP arithmetic

Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result.

- Concise specification, suitable for program verification.

Why is FP Arithmetic Amenable to Formal Proof?

IEEE-754 standard for FP arithmetic

Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result.

- Concise specification, suitable for program verification.
- It is all about real numbers.

Exceptional Values

Floating-point computations can lead to exceptional behaviors:

- invalid operations: $\sqrt{-1}$,
- overflow: $2 \times 2 \times \cdots \times 2$.

Exceptional Values

Floating-point computations can lead to exceptional behaviors:

- invalid operations: $\sqrt{-1}$,
- overflow: $2 \times 2 \times \cdots \times 2$.

When proving a FP algorithm, the very first step is to prove that

- exceptional behaviors cannot arise, or
- they are properly handled.

Exceptional Values

Floating-point computations can lead to exceptional behaviors:

- invalid operations: $\sqrt{-1}$,
- overflow: $2 \times 2 \times \cdots \times 2$.

When proving a FP algorithm, the very first step is to prove that

- exceptional behaviors cannot arise, or
- they are properly handled.

Today's talk is not about floating-point exceptions.
Let us assume that they are proved not to occur.

(This can be achieved by computing the range of expressions.)

FP Numbers and Real Numbers, the Flocq Way

Since there are no exceptional behaviors,
floating-point numbers can be embedded into real numbers.

FP Numbers and Real Numbers, the Flocq Way

Since there are no exceptional behaviors,
floating-point numbers can be embedded into real numbers.

Representable numbers

$$\mathbb{F} = \{m \cdot \beta^e \in \mathbb{R} \mid m, e \in \mathbb{Z} \land |m| < \beta^p \land e \geq e_{\min}\}$$

with β, p, and $e_{\min}$ depending on the format.

FP Numbers and Real Numbers, the Flocq Way

Since there are no exceptional behaviors,
floating-point numbers can be embedded into real numbers.

Representable numbers

$$\mathbb{F} = \{m \cdot \beta^e \in \mathbb{R} \mid m, e \in \mathbb{Z} \wedge |m| < \beta^p \wedge e \geq e_{\min}\}$$

with β, p, and $e_{\min}$ depending on the format.

Rounding operators

The result of an addition $a \oplus b$ is $\circ(a + b)$
with $\circ : \mathbb{R} \to \mathbb{F}$ a monotonic function that is the identity on $\mathbb{F}$.
$\circ(\cdot)$ depends on the destination format and the rounding direction.

The Gappa Tool

Gappa 1.1: 11k lines of C++, 8k lines of Coq, GPL'd.

The Gappa Tool

Gappa 1.1: 11k lines of C++, 8k lines of Coq, GPL'd.

Example (Cody-Waite argument reduction for exp)

```
x = float<ieee_64,ne>(dummyx); # x is a double

Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(float<ieee_64,ne>(x*InvLog2));
t1 float<ieee_64,ne>= x - k*Log2h;

# prove that t1 is computed exactly
{ x in [0.7, 800] -> t1 = x - k*Log2h }

Log2h ~ 1/InvLog2; # user hint
```

Outline

1. Introduction
2. Interval arithmetic and forward error analysis
   - Preliminaries
   - Interval arithmetic
   - Forward error analysis
   - Example: fast sine
3. Dealing with more intricate algorithms
4. The Gappa tool

What We Want to Prove

- Bounds on program expressions:
  $\forall x_1, \ldots, x_m \in \mathbb{R},\ e_1 \in I_1 \wedge \ldots \wedge e_n \in I_n \Rightarrow e \in J$
  with $I_1, \ldots, I_n, J$ intervals with nonsymbolic bounds.

What We Want to Prove

- Bounds on program expressions:
  $\forall x_1, \ldots, x_m \in \mathbb{R},\ e_1 \in I_1 \wedge \ldots \wedge e_n \in I_n \Rightarrow e \in J$
  with $I_1, \ldots, I_n, J$ intervals with nonsymbolic bounds.
- Bounds on forward errors:
  $\forall x_1, \ldots, x_m \in \mathbb{R},\ e_1 \in I_1 \wedge \ldots \wedge e_n \in I_n \Rightarrow \tilde{e} - e \in K$
  with $\tilde{e}$ and e two expressions with close values.

A Variety of Forward Errors

Example (Addition)

Let u and v be approximated by $\tilde{u}$ and $\tilde{v}$.
What is the error between $\circ(\tilde{u} + \tilde{v})$ and $u + v$?

A Variety of Forward Errors

Example (Addition)

Let u and v be approximated by $\tilde{u}$ and $\tilde{v}$.
What is the error between $\circ(\tilde{u} + \tilde{v})$ and $u + v$?

Three errors are involved:

- between $\tilde{u}$ and u,
- between $\tilde{v}$ and v,
- round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$.

A Variety of Forward Errors

Example (Addition)

Let u and v be approximated by $\tilde{u}$ and $\tilde{v}$.
What is the error between $\circ(\tilde{u} + \tilde{v})$ and $u + v$?

Three errors are involved:

- between $\tilde{u}$ and u,
- between $\tilde{v}$ and v,
- round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$.

Each error bound might be either

- absolute: $\tilde{u} - u \in I$, or
- relative: $(\tilde{u} - u)/u \in I$.

A Variety of Round-off Errors

The round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$ is

- absolutely bounded if $\tilde{u}$ and $\tilde{v}$ are bounded,

A Variety of Round-off Errors

The round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$ is

- absolutely bounded if $\tilde{u}$ and $\tilde{v}$ are bounded,
- relatively bounded for FP formats with gradual underflow,

A Variety of Round-off Errors

The round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$ is

- absolutely bounded if $\tilde{u}$ and $\tilde{v}$ are bounded,
- relatively bounded for FP formats with gradual underflow,
- relatively bounded if $\tilde{u} + \tilde{v}$ is far enough from 0,

A Variety of Round-off Errors

The round-off error between $\circ(\tilde{u} + \tilde{v})$ and $\tilde{u} + \tilde{v}$ is

- absolutely bounded if $\tilde{u}$ and $\tilde{v}$ are bounded,
- relatively bounded for FP formats with gradual underflow,
- relatively bounded if $\tilde{u} + \tilde{v}$ is far enough from 0,
- zero if $\tilde{u} + \tilde{v}$ is in a suitable fixed-point format,

A Variety of Round-off Errors

The round-off error between $\circ(\tilde{u}+\tilde{v})$ and $\tilde{u}+\tilde{v}$ is

- absolutely bounded if $\tilde{u}$ and $\tilde{v}$ are bounded,
- relatively bounded for FP formats with gradual underflow,
- relatively bounded if $\tilde{u}+\tilde{v}$ is far enough from 0,
- zero if $\tilde{u}+\tilde{v}$ is in a suitable fixed-point format,
- zero if $\tilde{u}/\tilde{v} \in [-2, -1/2]$ for FP formats with gradual underflow.

Interval Arithmetic

Interval arithmetic extends operations on real numbers to operations on closed connected subsets of real numbers.

Application

Instead of proving $\forall x \in [a, b],\ f(x) \in [c, d]$,
you can prove $F([a, b]) \subseteq [c, d]$,
assuming that F is an interval extension of f.

Interval Arithmetic

Interval arithmetic extends operations on real numbers to operations on closed connected subsets of real numbers.

Application

Instead of proving $\forall x \in [a, b],\ f(x) \in [c, d]$,
you can prove $F([a, b]) \subseteq [c, d]$,
assuming that F is an interval extension of f.

Evaluating F is easy; it involves operations on bounds only:

$$x \in [a, b] \wedge y \in [c, d] \Rightarrow x + y \in [a + c, b + d].$$

This makes interval arithmetic suitable for automatically proving bounds on real-valued expressions.

Interval Arithmetic and Dependencies

Independent expressions

If $a \in [3, 5]$ and $b \in [1, 2]$ are independent, then

$$a - b \in [3 - 2, 5 - 1] = [1, 4]$$

is the optimal enclosure.

Interval Arithmetic and Dependencies

Independent expressions

If $a \in [3,5]$ and $b \in [1,2]$ are independent, then

$$a - b \in [3-2, 5-1] = [1,4]$$

is the optimal enclosure.

Correlated expressions

If we have $a \in [1,100]$, interval arithmetic gives

$$(a+\varepsilon) - a \in [1+\varepsilon, 100+\varepsilon] - [1,100] = [-99+\varepsilon, 99+\varepsilon]$$

while the optimal enclosure is $[\varepsilon, \varepsilon]$.

Interval Arithmetic and Dependencies

Various methods solve the dependency issue:

- octogons,
- ellipsoids,
- zonotopes,
- Taylor/Chebyshev models,
- decision procedures, e.g. simplex or CAD.

Interval Arithmetic and Dependencies

Various methods solve the dependency issue:

- octogons,
- ellipsoids,
- zonotopes,
- Taylor/Chebyshev models,
- decision procedures, e.g. simplex or CAD.

Unfortunately they are much costlier than interval arithmetic at execution time, and even worse at formalization time.

Leveraging Forward Error Analysis

Forward error analysis offers a simpler way to deal with dependencies.

- "the absolute error of the sum is the sum of the absolute errors"

$$(\tilde{u} + \tilde{v}) - (u + v) = (\tilde{u} - u) + (\tilde{v} - v)$$

Leveraging Forward Error Analysis

Forward error analysis offers a simpler way to deal with dependencies.

- “the absolute error of the sum is the sum of the absolute errors”

$$(\tilde{u} + \tilde{v}) - (u + v) = (\tilde{u} - u) + (\tilde{v} - v)$$

- “the relative error of the product is the sum of the relative errors”

$$\frac{\tilde{u}\tilde{v}}{uv} - 1 = \varepsilon_u + \varepsilon_v + \varepsilon_u \varepsilon_v$$

with $\varepsilon_u = \tilde{u}/u - 1$ and $\varepsilon_v = \tilde{v}/v - 1$

Leveraging Forward Error Analysis

Forward error analysis offers a simpler way to deal with dependencies.

- "the absolute error of the sum is the sum of the absolute errors"

$$(\tilde{u} + \tilde{v}) - (u + v) = (\tilde{u} - u) + (\tilde{v} - v)$$

- "the relative error of the product is the sum of the relative errors"

$$\frac{\tilde{u}\tilde{v}}{uv} - 1 = \varepsilon_u + \varepsilon_v + \varepsilon_u \varepsilon_v$$

with $\varepsilon_u = \tilde{u}/u - 1$ and $\varepsilon_v = \tilde{v}/v - 1$

- "the relative error of rounding operators is bounded"

$$\left| \frac{\circ(u)}{u} - 1 \right| \leq 2^{-p} \text{ if } |u| \geq \ldots$$

Leveraging Forward Error Analysis

Forward error analysis:

- $(\tilde{u} + \tilde{v}) - (u + v) = (\tilde{u} - u) + (\tilde{v} - v)$
- $(\tilde{u}\tilde{v})/(uv) - 1 = \varepsilon_u + \varepsilon_v + \varepsilon_u\varepsilon_v$

This inductive rewriting works fine as long as

- errors are not correlated,
- expressions have the same inductive structure with correlated sub-expressions in the same places.

Because of the two-step verification process, the above often holds.

Example: Sine Around Zero

How to efficiently compute $\sin x$ for $|x| \leq 1$
with a relative accuracy bounded by $103 \cdot 2^{-16}$?

Example: Sine Around Zero

How to efficiently compute $\sin x$ for $|x| \leq 1$
with a relative accuracy bounded by $103 \cdot 2^{-16}$?

Example (Toy sine)

```
float toy_sin(float x) {
  if (fabsf(x) < 0x1p-5f) return x;
  return x * (1.0f - x * x * 0x28e9p-16f);
}
```

Example: Sine Around Zero

How to efficiently compute $\sin x$ for $|x| \leq 1$
with a relative accuracy bounded by $103 \cdot 2^{-16}$?

Example (Toy sine)

```
float toy_sin(float x) {
  if (fabsf(x) < 0x1p-5f) return x;
  return x * (1.0f - x * x * 0x28e9p-16f);
}
```

An actual implementation of sin would

- use more than just 2 polynomials, and/or
- perform an argument reduction.

But the proof process is the same!

Approximating a Mathematical Function

How to compute an accurate FP approximation of $g(x)$ for any x?

Approximating a Mathematical Function

How to compute an accurate FP approximation of $g(x)$ for any x?

1. Find an approximation $\hat{g}$ of g that uses only real operations that can be approximated by your floating-point unit.

   Bound the method error $\hat{g}(x)/g(x) - 1$.

Approximating a Mathematical Function

How to compute an accurate FP approximation of $g(x)$ for any x?

1. Find an approximation $\hat{g}$ of g that uses only real operations that can be approximated by your floating-point unit.

   Bound the method error $\hat{g}(x)/g(x) - 1$.

2. Write $\tilde{g}$ that implements $\hat{g}$ with floating-point operations.

   Bound the round-off error $\tilde{g}(x)/\hat{g}(x) - 1$.

Approximating a Mathematical Function

How to compute an accurate FP approximation of $g(x)$ for any x?

1. Find an approximation $\hat{g}$ of g that uses only real operations that can be approximated by your floating-point unit.
   Bound the method error $\hat{g}(x)/g(x) - 1$.
2. Write $\tilde{g}$ that implements $\hat{g}$ with floating-point operations.
   Bound the round-off error $\tilde{g}(x)/\hat{g}(x) - 1$.
3. Compose both bounds to get $\tilde{g}(x)/g(x) - 1$.

Approximating a Mathematical Function

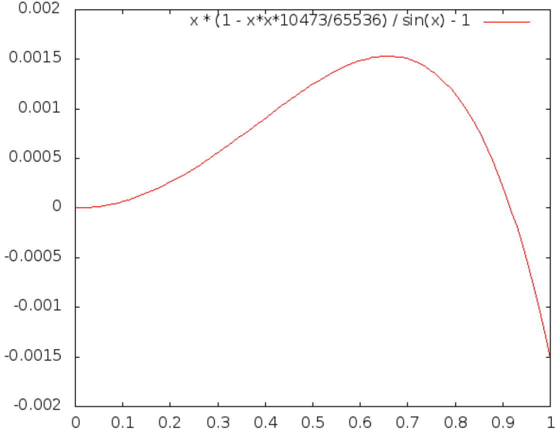
How to compute an accurate FP approximation of $g(x)$ for any x?

1. Find an approximation $\hat{g}$ of g that uses only real operations that can be approximated by your floating-point unit.
   Bound the method error $\hat{g}(x)/g(x) - 1$.
2. Write $\tilde{g}$ that implements $\hat{g}$ with floating-point operations.
   Bound the round-off error $\tilde{g}(x)/\hat{g}(x) - 1$.
3. Compose both bounds to get $\tilde{g}(x)/g(x) - 1$.

Proving correctness is just a matter of computing tight bounds for these expressions.

Method Error (Relative)

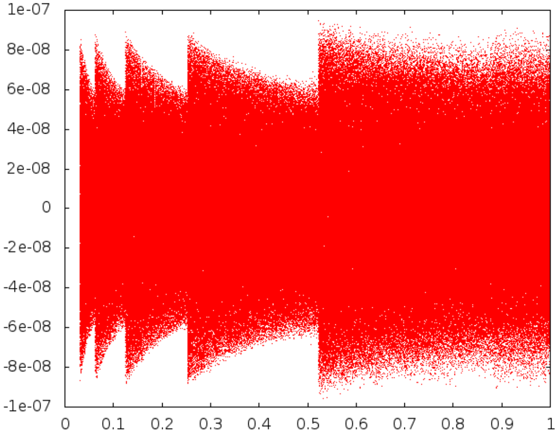
Method error: $\frac{x \cdot (1 - x^2 \cdot 10473 \cdot 2^{-16})}{\sin x} - 1.$



Interval analysis knows how to bound such an expression.

Binary32 Round-off Error (Relative)

Round-off error: $\frac{\circ(x \cdot \circ(1 - \circ(\circ(x^2) \cdot 10473 \cdot 2^{-16})))}{x \cdot (1 - x^2 \cdot 10473 \cdot 2^{-16})} - 1.$



Gappa knows how to bound such an expression.
(And how to compose method and round-off errors.)

Correctness Statement in Coq

```
Notation fsub x y :=
  (round radix2 binary32_fmt rndNE (x - y)).
Notation fmul x y :=
  (round radix2 binary32_fmt rndNE (x * y)).

Definition fsin x :=
  if Rle_lt_dec (pow2 (-5)) (Rabs x) then
    fmul x (fsub 1 (fmul (fmul x x)
      (10473 * pow2 (-16))))
  else x.

Lemma sine_spec : forall x, Rabs x <= 1 ->
  Rabs (fsin x - sin x) <= 103*pow2 (-16) *
    Rabs (sin x).
```

Proof Sketch in Coq

```
Lemma sine_spec : forall x, Rabs x <= 1 ->
  Rabs (fsin x - sin x) <= 103 * pow2 (-16) *
    Rabs (sin x).
Proof.
intros x Bx. unfold fsin.
case Rle_lt_dec ; intros Bx'.
- (* |x| >= 1/32, degree-3 approx *)
  assert (Rabs (x * (1 - x * x * (10473*pow2 (-16))) -
       sin x) <= 102*pow2 (-16) * Rabs (sin x)).
    (* bound the method error *)
    interval with (i_bisect_diff x).
  (* bound the round-off and total errors *)
  gappa.
- (* |x| < 1/32, degree-1 approx *)
  destruct (MVT_cor2 sin cos).
  interval.
Qed.
```

Gappa Script, as Written by a Human

Example (Relative error for a toy sin implementation)

```
@rnd = float<ieee_32,ne>;
x = rnd(dummyx); # x is a float

# floating-point implementation
y rnd= x * (1 - x*x * 0x28E9p-16);
# infinitely-precise computation
My    = x * (1 - x*x * 0x28E9p-16);

{ |x| in [1b-5,1] /\
  # relative method error
  |My -/ sin_x| <= 1.55e-3 ->

  # relative total error
  |y -/ sin_x| <= 1.551e-3 }
```

Outline

1. Introduction
2. Interval arithmetic and forward error analysis
3. Dealing with more intricate algorithms
   - Example: Cody-Waite argument reduction
   - Example: Integer division on Itanium
4. The Gappa tool

Intricate Algorithms

For some algorithms, bounding errors is not sufficient, as they might rely on various tricks:

- exact computations,
- error compensations,
- convergent iterations,
- and so on.

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \leq 800$.

Argument reduction: replace x by a value close to 0,
so that exp can be approximated by a small polynomial.

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \leq 800$.

Argument reduction: replace x by a value close to 0,
so that exp can be approximated by a small polynomial.

- Idea 1: use $\exp x = 2^k \exp(x - k \log 2)$ with k an integer.

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \leq 800$.

Argument reduction: replace x by a value close to 0,
so that exp can be approximated by a small polynomial.

- Idea 1: use $\exp x = 2^k \exp(x - k \log 2)$ with k an integer.
- Issue: how to compute $x - k \log 2$ accurately?

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \le 800$.

Argument reduction: replace x by a value close to 0,
so that exp can be approximated by a small polynomial.

- Idea 1: use $\exp x = 2^k \exp(x - k \log 2)$ with k an integer.
- Issue: how to compute $x - k \log 2$ accurately?
- Idea 2: use $\log 2 = \ell_h + \ell_l + \varepsilon$ with ε close to negligible.

$$\exp x = 2^k \exp((x - k\ell_h) - k\ell_l) \ \exp(-k\varepsilon).$$

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \leq 800$.

Argument reduction: replace x by a value close to 0, so that exp can be approximated by a small polynomial.

- Idea 1: use $\exp x = 2^k \exp(x - k \log 2)$ with k an integer.
- Issue: how to compute $x - k \log 2$ accurately?
- Idea 2: use $\log 2 = \ell_h + \ell_l + \varepsilon$ with ε close to negligible.

$$\exp x = 2^k \exp((x - k\ell_h) - k\ell_l)\ \exp(-k\varepsilon).$$

- Implementation: evaluate $(x - k\ell_h) - k\ell_l$ with FP arithmetic.

$$\exp x = 2^k \exp(\circ(\ldots))\ \exp(\delta - k\varepsilon).$$

Cody-Waite Argument Reduction

Goal: compute $\exp x$ for $|x| \le 800$.

Argument reduction: replace x by a value close to 0,
so that exp can be approximated by a small polynomial.

- Idea 1: use $\exp x = 2^k \exp(x - k \log 2)$ with k an integer.
- Issue: how to compute $x - k \log 2$ accurately?
- Idea 2: use $\log 2 = \ell_h + \ell_l + \varepsilon$ with ε close to negligible.

$$\exp x = 2^k \exp((x - k\ell_h) - k\ell_l)\ \exp(-k\varepsilon).$$

- Implementation: evaluate $(x - k\ell_h) - k\ell_l$ with FP arithmetic.

$$\exp x = 2^k \exp(\circ(\ldots))\ \exp(\delta - k\varepsilon).$$

- Issue: how much is δ?

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp, part 1)

```
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
```

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp, part 1)

```
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
```

Proof.

1. $|x| \leq 800$, so $|k| < 2048$, so k fits on 11 bits.

□

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp, part 1)

```
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
```

Proof.

1. $|x| \leq 800$, so $|k| < 2048$, so k fits on 11 bits.
2. ℓ_h fits on 42 bits, so $\circ(k\ell_h) = k\ell_h$.

□

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp, part 1)

```
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
```

Proof.

1. $|x| \leq 800$, so $|k| < 2048$, so k fits on 11 bits.
2. ℓ_h fits on 42 bits, so $\circ(k\ell_h) = k\ell_h$.
3. $\ell_h^{-1} \approx$ InvLog2, so $x \approx k\ell_h$.

□

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp, part 1)

```
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
```

Proof.

1. $|x| \leq 800$, so $|k| < 2048$, so k fits on 11 bits.
2. ℓ_h fits on 42 bits, so $\circ(k\ell_h) = k\ell_h$.
3. $\ell_h^{-1} \approx$ `InvLog2`, so $x \approx k\ell_h$.
4. So $\circ(x - \circ(k\ell_h)) = x - k\ell_h$ by Sterbenz.

□

Exact Computations

For intricate algorithms, ranges of expressions are not enough.

You also need to know how many bits you need to represent them.

Cody-Waite Argument Reduction

Example (Cody-Waite argument reduction for exp)

```
@rnd = float<ieee_64,ne>;
x = rnd(dummyx); # x is a double

# Cody-Waite argument reduction
Log2h = 0xb.17217f7d1cp-4; # 42 bits out of 53
Log2l = 0xf.79abc9e3b398p-48;
InvLog2 = 0x1.71547652b82fep0;
k = int<ne>(rnd(x*InvLog2));
t1 rnd= x - k*Log2h;
t2 rnd= t1 - k*Log2l;

# exact values
T1 = x - k*Log2h;
T2 = T1 - k*Log2l;

{ x in [0.3, 800] ->

  t1 = T1 /\
  T1 in [-0.35,0.35] /\
  t2 - T2 in ? }

Log2h ~ 1/InvLog2;

# try harder!
T1 $ x;
```

Integer Division on Itanium

Intel Itanium processors have no hardware divisor.
How to efficiently perform a division with just add and mul?

Integer Division on Itanium

Intel Itanium processors have no hardware divisor.
How to efficiently perform a division with just add and mul?

Example (Division of 16-bit unsigned integers on Itanium)

```
// Inputs:  dividend a in f6, divisor b in f7, 1 + 2^-17 in f9
      frcpa.s1    f8,p6=f6,f7 ;;
(p6) fma.s1      f6=f6,f8,f0
(p6) fnma.s1     f7=f7,f8,f9 ;;
(p6) fma.s1      f8=f7,f6,f6 ;;
      fcvt.fx.trunc.s1 f8=f8
// Output:  ⌊a/b⌋ in f8
```

Integer Division on Itanium

Example (Division of 16-bit unsigned integers on Itanium)

$$
\begin{array}{rcl}
y_0 & \approx & 1/b \qquad [\texttt{frcpa}] \\
q_0 & = & \circ(a \times y_0) \\
e_0 & = & \circ(1 + 2^{-17} - b \times y_0) \\
q_1 & = & \circ(e_0 \times q_0 + q_0) \\
q & = & \lfloor q_1 \rfloor
\end{array}
$$

with $\circ(\cdot)$ rounding to nearest on the extended 82-bit format.

Correctness of the division

$$\forall a, b \in [\![1; 65535]\!], \quad q = \lfloor a/b \rfloor.$$

Correctness Statement in Coq

```
Notation fma x y z :=
  (round radix2 register_fmt rndNE (x * y + z)).

Axiom frcpa : R -> R.
Axiom frcpa_spec : forall x : R,
  1 <= Rabs x <= 65536 ->
  generic_format radix2 (FLT_exp _ 11) (frcpa x) /\
  Rabs (frcpa x - 1/x) <= 4433*pow2 (-21) * Rabs(1/x).

Definition div_u16 a b :=
  let y0 := frcpa b in
  let q0 := fma a y0 0 in
  let e0 := fnma b y0 (1 + pow2 (-17)) in
  let q1 := fma e0 q0 q0 in
  Zfloor q1.

Lemma div_u16_spec : forall a b,
  (1 <= a <= 65535)%Z ->
  (1 <= b <= 65535)%Z ->
  div_u16 a b = (a / b)%Z.
```

Proof Sketch

Theorem (Exclusion zones)

Given a and b positive integers.
If $0 \leq a \times (q_1/(a/b) - 1) < 1$, *then* $\lfloor q_1 \rfloor = \lfloor a/b \rfloor$.

Proof Sketch

Theorem (Exclusion zones)

Given a and b positive integers.
If $0 \le a \times (q_1/(a/b) - 1) < 1$, *then* $\lfloor q_1 \rfloor = \lfloor a/b \rfloor$.

Notice the relative error between the FP value q_1 and the real a/b.
So proving the correctness is just a matter of bounding this error.

Proof Sketch Continued

Bounding the method error $\hat{q}_1 - a/b$ and the round-off error $q_1 - \hat{q}_1$ and composing them does not work at all.

Proof Sketch Continued

Bounding the method error $\hat{q}_1 - a/b$ and the round-off error $q_1 - \hat{q}_1$ and composing them does not work at all.

What the developers knew when designing the algorithm:

- If not for 2^{-17}, the code would perform a Newton iteration:
  $\hat{q}_1/(a/b) - 1 = -\varepsilon_0^2$ with $\varepsilon_0 = y_0/(1/b) - 1$.
- By taking into account 2^{-17},
  $\hat{q}_1/(a/b) - 1 = -\varepsilon_0^2 + (1 + \varepsilon_0) \cdot 2^{-17}$.

Proof Sketch, the Coq Version

```
Lemma div_u16_spec : forall a b,
  (1 <= a <= 65535)%Z -> (1 <= b <= 65535)%Z ->
  div_u16 a b = (a / b)%Z.
Proof.
intros a b Ba Bb.
apply Zfloor_imp.
cut (0 <= b * q1 - a < 1).
  lra.
set (err := (q1 - a / b) / (a / b)).
replace (b * q1 - a) with (a * err) by field.
set (y0 := frcpa b).
set (Mq0 := a * y0 + 0).
set (Me0 := 1 + pow2 (-17) - b * y0).
set (Mq1 := Me0 * Mq0 + Mq0).
set (eps0 := (y0 - 1 / b) / (1 / b)).
assert ((Mq1 - a / b) / (a / b) =
  -(eps0 * eps0) + (1 + eps0) * pow2 (-17)) by field.
generalize (frcpa_spec b) (FIX_format_Z2R radix2 a)
  (FIX_format_Z2R radix2 b).
gappa.
Qed.
```

Convergent Algorithms

If you know some clever property about an algorithm,
don't expect automatic tools to infer it, just tell them about it.

The Gappa Script, as Written by a Human

Example (Division of 16-bit unsigned integers on Itanium)

```
@rnd = float<x86_80,ne>;

# algorithm with no rounding operators
q0 = a * y0;
e0 = 1 + 1b-17 - b * y0;
q1 = q0 + e0 * q0;

# notations for relative errors
eps0 = (y0 - 1 / b) / (1 / b);
err = (q1 - a / b) / (a / b);

{ # a and b are integers
  @FIX(a, 0) /\ a in [1,65535] /\
  @FIX(b, 0) /\ b in [1,65535] /\
  # specification of frcpa
  @FLT(y0, 11) /\ |eps0| <= 0.00211373 /\
  # Newton's iteration, almost
  err = -(eps0 * eps0) + (1 + eps0) * 1b-17 ->

  # the separation hypothesis is satisfied
  err in [0,1] /\ a * err in [0,0.99999] /\
  # all the computations are exact
  rnd(q0) = q0 /\ rnd(e0) = e0 /\ rnd(q1) = q1 }

# try harder!
rnd(q1) = q1 $ 1 / b;
```

Outline

A Few Words About Gappa

Starting from a formula, Gappa saturates a set of theorems to infer new properties until it encounters a contradiction.

A Few Words About Gappa

Starting from a formula, Gappa saturates a set of theorems to infer new properties until it encounters a contradiction.

Supported properties

$$
\begin{array}{lcll}
\mathtt{BND}(x, I) & \equiv & x \in I & \\
\mathtt{ABS}(x, I) & \equiv & |x| \in I & \\
\mathtt{REL}(x, y, I) & \equiv & \exists \varepsilon \in I, & x = y \cdot (1 + \varepsilon) \\
\mathtt{FIX}(x, e) & \equiv & \exists m \in \mathbb{Z}, & x = m \cdot 2^e \\
\mathtt{FLT}(x, p) & \equiv & \exists m, e \in \mathbb{Z}, & x = m \cdot 2^e \wedge |m| < 2^p \\
\mathtt{NZR}(x) & \equiv & x \neq 0 & \\
\mathtt{EQL}(x, y) & \equiv & x = y &
\end{array}
$$

A Few Words About Gappa

Starting from a formula, Gappa saturates a set of theorems to infer new properties until it encounters a contradiction.

Supported properties

$$
\begin{array}{lcll}
\texttt{BND}(x, I) & \equiv & x \in I & \\
\texttt{ABS}(x, I) & \equiv & |x| \in I & \\
\texttt{REL}(x, y, I) & \equiv & \exists \varepsilon \in I, & x = y \cdot (1 + \varepsilon) \\
\texttt{FIX}(x, e) & \equiv & \exists m \in \mathbb{Z}, & x = m \cdot 2^e \\
\texttt{FLT}(x, p) & \equiv & \exists m, e \in \mathbb{Z}, & x = m \cdot 2^e \wedge |m| < 2^p \\
\texttt{NZR}(x) & \equiv & x \neq 0 & \\
\texttt{EQL}(x, y) & \equiv & x = y &
\end{array}
$$

To prove div_u16, Gappa tried to apply 17k theorems.
The final proof infers ∼80 properties.

Proof Process

Given a logical formula about some expressions $e_1, \ldots, e_n$, Gappa performs the following steps:

Proof Process

Given a logical formula about some expressions $e_1, \ldots, e_n$,
Gappa performs the following steps:

1. Recursively and symbolically instantiate all the theorems that might lead to deducing a fact about some expression e_i.
(backward reasoning)

Proof Process

Given a logical formula about some expressions $e_1, \ldots, e_n$,
Gappa performs the following steps:

1. Recursively and symbolically instantiate all the theorems that might lead to deducing a fact about some expression e_i.
   (backward reasoning)
2. Iteratively and numerically instantiate all these theorems. Keep track of them when they produce a new fact.
   (forward reasoning)

Proof Process

Given a logical formula about some expressions $e_1, \ldots, e_n$, Gappa performs the following steps:

1. Recursively and symbolically instantiate all the theorems that might lead to deducing a fact about some expression e_i.
   (backward reasoning)
2. Iteratively and numerically instantiate all these theorems. Keep track of them when they produce a new fact.
   (forward reasoning)
3. Once a full proof trace is obtained, minimize it by simplifying or removing as many theorem instances as possible.

Proof Process

Given a logical formula about some expressions $e_1, \ldots, e_n$,
Gappa performs the following steps:

1. Recursively and symbolically instantiate all the theorems that might lead to deducing a fact about some expression e_i.
   (backward reasoning)
2. Iteratively and numerically instantiate all these theorems. Keep track of them when they produce a new fact.
   (forward reasoning)
3. Once a full proof trace is obtained, minimize it by simplifying or removing as many theorem instances as possible.
4. Generate a formal proof from the trace.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.
- Not so naive interval arithmetic:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\mathrm{lower}(|U - V|), \mathrm{upper}(U + V)]$.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.
- Not so naive interval arithmetic:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\mathrm{lower}(|U - V|), \mathrm{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.
- Not so naive interval arithmetic:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\text{lower}(|U - V|), \text{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- Forward error analysis:
  $\tilde{u} \times \tilde{v} - u \times v = (\tilde{u} - u) \times v + u \times (\tilde{v} - v) + (\tilde{u} - u) \times (\tilde{v} - v)$.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.
- Not so naive interval arithmetic:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\text{lower}(|U - V|), \text{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- Forward error analysis:
  $\tilde{u} \times \tilde{v} - u \times v = (\tilde{u} - u) \times v + u \times (\tilde{v} - v) + (\tilde{u} - u) \times (\tilde{v} - v)$.
- Precision handling:
  $\text{FLT}(x, p) \wedge \text{FLT}(y, q) \Rightarrow \text{FLT}(x \times y, p + q)$.

Theorem Database

- Naive interval arithmetic:
  $u \in [\underline{u}, \overline{u}] \wedge v \in [\underline{v}, \overline{v}] \Rightarrow u + v \in [\underline{u} + \underline{v}, \overline{u} + \overline{v}]$.
- Not so naive interval arithmetic:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\text{lower}(|U - V|), \text{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- Forward error analysis:
  $\tilde{u} \times \tilde{v} - u \times v = (\tilde{u} - u) \times v + u \times (\tilde{v} - v) + (\tilde{u} - u) \times (\tilde{v} - v)$.
- Precision handling:
  $\text{FLT}(x, p) \wedge \text{FLT}(y, q) \Rightarrow \text{FLT}(x \times y, p + q)$.
- And so on.

Theorem Database

Category	Thm
Interval arithmetic	21
Representability	14
Relative error	15
Rewriting rules	45
FP/FXP arithmetic	25
Miscellaneous	27
Total	147

Conclusion

Gappa does not work on large programs,
only on short straight-line algorithms.

Conclusion

Gappa does not work on large programs,
only on short straight-line algorithms.

It is nowhere as powerful as the dumbest decision procedures.

Conclusion

Gappa does not work on large programs,
only on short straight-line algorithms.

It is nowhere as powerful as the dumbest decision procedures.

But with a bit of help from the user,
it can make short work of intricate algorithms.

Conclusion

Gappa does not work on large programs,
only on short straight-line algorithms.

It is nowhere as powerful as the dumbest decision procedures.

But with a bit of help from the user,
it can make short work of intricate algorithms.

And it generates formal proofs!

Questions?

Gappa: `http://gappa.gforge.inria.fr/`