Automating the Verification of Floating-point Algorithms

Guillaume Melquiond

Inria Saclay–Île-de-France

LRI, Université Paris Sud, CNRS

2014-07-18

This Talk's Content

Disclaimer

- I will not talk much about SMT.

This Talk's Content

Disclaimer

- I will not talk much about SMT.
- I will not talk about floating-point arithmetic in general, but only about its use in a peculiar context: mathematical libraries (libm).

This Talk's Content

Disclaimer

- I will not talk much about SMT.
- I will not talk about floating-point arithmetic in general, but only about its use in a peculiar context: mathematical libraries (libm).
- I will focus on a specific procedure for verifying floating-point algorithms: the Gappa tool.

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.

FP Arithmetic Quick Reference Card

IEEE-754 FP numbers

- Exceptional values: ± 0, $\pm\infty$, Not-a-Number.
- Finite values:
  $\mathbb{F} = \{m \cdot \beta^e \in \mathbb{R} \mid m, e \in \mathbb{Z} \wedge |m| < \beta^p \wedge e_{\mathrm{min}} \leq e \leq e_{\mathrm{max}}\}$
  with β, p, e_{min}, e_{max} parameters of the format.

FP Arithmetic Quick Reference Card

IEEE-754 FP numbers

- Exceptional values: ± 0, $\pm\infty$, Not-a-Number.
- Finite values:
  $\mathbb{F} = \{m \cdot \beta^e \in \mathbb{R} \mid m, e \in \mathbb{Z} \wedge |m| < \beta^p \wedge e_{\min} \le e \le e_{\max}\}$
  with β, p, $e_{\min}$, $e_{\max}$ parameters of the format.

FP arithmetic operations

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.

- $a \oplus b = \circ(a + b)$, $a \otimes b = \circ(a \times b)$, and so on.
- Rounding to nearest: $\forall y \in \mathbb{F}, \quad |\circ(x) - x| \le |y - x|$.

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?

Outline

1. Introduction
   - Context
   - Gallery
   - Gappa
   - Decidability
2. Interval arithmetic and forward error analysis
3. Dealing with more intricate algorithms
4. The Gappa tool
5. Conclusion

Example 1: Toy Elementary Function

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

Example 1: Toy Elementary Function

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

Verification condition for accuracy

$$\forall x \in \mathbb{F}_{32}, \quad |x| \leq 1 \quad \Rightarrow \quad \left| \frac{\texttt{toy\_sin}\ x}{\sin x} - 1 \right| \leq 103 \cdot 2^{-16}.$$

Example 1: Toy Elementary Function

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

Verification condition for accuracy

$$\forall x \in \mathbb{F}_{32}, \quad |x| \leq 1 \quad \Rightarrow \quad \left| \frac{\texttt{toy\_sin}\ x}{\sin x} - 1 \right| \leq 103 \cdot 2^{-16}.$$

Optional hypothesis: $\left| \frac{x - 10473 \cdot 2^{-16} x^3}{\sin x} - 1 \right| \leq 102 \cdot 2^{-16}$.

Example 1: Toy Elementary Function

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

Verification condition for accuracy

$$\forall x \in \mathbb{F}_{32}, \quad |x| \le 1 \quad \Rightarrow \quad \left| \frac{\texttt{toy\_sin}\ x}{\sin x} - 1 \right| \le 103 \cdot 2^{-16}.$$

Optional hypothesis: $\left| \frac{x - 10473 \cdot 2^{-16} x^3}{\sin x} - 1 \right| \le 102 \cdot 2^{-16}$.

- Relative errors.

Example 2: Cody-Waite Argument Reduction for Exp

```
double exp(double x) {
  if (fabs(x) >= 800) return ...;
  double k = nearbyint(x * 0x1.71547652b82fep0);
  double t1 = x  - k * 0xb.17217f7d1cp-4;
  double t2 = t1 - k * 0xf.79abc9e3b398p-48;
  ...
}
```

Example 2: Cody-Waite Argument Reduction for Exp

```
double exp(double x) {
  if (fabs(x) >= 800) return ...;
  double k = nearbyint(x * 0x1.71547652b82fep0);
  double t1 = x  - k * 0xb.17217f7d1cp-4;
  double t2 = t1 - k * 0xf.79abc9e3b398p-48;
  ...
}
```

Verification conditions

- $|t_2| \le 0.35$,
- $|t_2 - (x - k \cdot \mathtt{0xb.17217f7d1cf79abc9e3b398p\text{-}4})| \le 2^{-55}$.

Example 2: Cody-Waite Argument Reduction for Exp

```
double exp(double x) {
  if (fabs(x) >= 800) return ...;
  double k = nearbyint(x * 0x1.71547652b82fep0);
  double t1 = x  - k * 0xb.17217f7d1cp-4;
  double t2 = t1 - k * 0xf.79abc9e3b398p-48;
  ...
}
```

Verification conditions

- $|t_2| \le 0.35$,
- $|t_2 - (x - k \cdot \texttt{0xb.17217f7d1cf79abc9e3b398p-4})| \le 2^{-55}$.

- Vanishing round-off errors.

Example 3: Itanium Division of 16-bit Unsigned Integers

```
// 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
```

Example 3: Itanium Division of 16-bit Unsigned Integers

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

with $\circ(\cdot)$ rounding to nearest on Itanium's 82-bit format.

Example 3: Itanium Division of 16-bit Unsigned Integers

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

with $\circ(\cdot)$ rounding to nearest on Itanium's 82-bit format.

Verification condition

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

Example 3: Itanium Division of 16-bit Unsigned Integers

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

with $\circ(\cdot)$ rounding to nearest on Itanium's 82-bit format.

Verification condition

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

- Vanishing round-off errors.
- Polynomial manipulations.

Example 4: Knuth' TwoSum Algorithm

```
s = a + b
t = s - a
e = (a - (s - t)) + (b - t)
```

Example 4: Knuth' TwoSum Algorithm

```
s = a + b
t = s - a
e = (a - (s - t)) + (b - t)
```

Verification condition

Assuming no overflow occurs, $s + e = a + b$.

Example 4: Knuth' TwoSum Algorithm

```
s = a + b
t = s - a
e = (a - (s - t)) + (b - t)
```

Verification condition

Assuming no overflow occurs, $s + e = a + b$.

- Pointless infinitely-precise values.
- Pointless round-off errors.

Scope and Constraints of Gappa

Scope

- Only real numbers: no exceptional values.
- Basic arithmetic operations: $+$, $\times$, $\div$, $\sqrt{\cdot}$.
- Radix-2 fixed- and FP arithmetic (no multi-precision).
- Logical formulas (no control flow).

Scope and Constraints of Gappa

Scope

- Only real numbers: no exceptional values.
- Basic arithmetic operations: $+$, $\times$, $\div$, $\sqrt{\cdot}$.
- 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 of Gappa

Scope

- Only real numbers: no exceptional values.
- Basic arithmetic operations: $+$, $\times$, $\div$, $\sqrt{\cdot}$.
- 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 some user help).
- Generate Coq proofs that fit into Flocq's formalism.
- Answer instantly.

Why No Exceptional Values?

- Safety of most programs relies on their absence.
  In that case, range computation is sufficient.

Why No Exceptional Values?

- Safety of most programs relies on their absence.
  In that case, range computation is sufficient.

- Their propagation is purely combinatorial anyway.
  Just use your preferred SAT method.

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
```

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
```

Generated Coq proof: 664 lines, 55 reasoning steps.

Rounding Operators and Decidability

Effective rounding

$$\circ(x) = \mathrm{ulp}(x) \cdot \lfloor x/\mathrm{ulp}(x) \rceil,$$

with ulp(x) the distance between the 2 FP numbers surrounding x.

Note: ulp is piecewise constant, e.g. 4091 ranges for binary64.

Rounding Operators and Decidability

Effective rounding

$$\circ(x) = \mathrm{ulp}(x) \cdot \lfloor x/\mathrm{ulp}(x) \rceil,$$

with ulp(x) the distance between the 2 FP numbers surrounding x.

Note: ulp is piecewise constant, e.g. 4091 ranges for binary64.

Decidability of FP arithmetic

For bounded quantifications, the following theories are decidable:

- $(\mathbb{R}, +, \circ(\cdot))$,
- $(\mathbb{F}, \oplus, \otimes, \ldots)$.

Outline

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.

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

Leveraging Forward Error Analysis

Some well-known results of the standard model of FP arithmetic:

- "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

Some well-known results of the standard model of FP arithmetic:

- "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

Some well-known results of the standard model of FP arithmetic:

- "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 round-off error is bounded."
$$\left| \frac{\circ(u)}{u} - 1 \right| \leq 2^{-p} \text{ if } |u| \geq \ldots$$

Leveraging Forward Error Analysis

Rewriting system:

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

Sufficient as long as

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

Because of the 2-step design/verification process,
these hypotheses often hold.

Example 1: Toy Elementary Function

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

Example 1: Toy Elementary Function

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 1: Toy Elementary Function

How to efficiently compute $\sin x$ for $|x| \le 1$
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

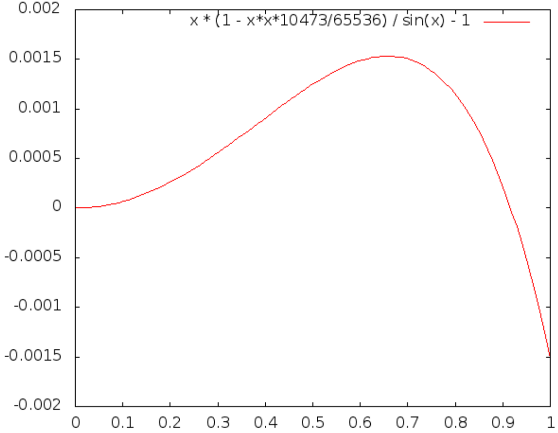
How to compute an accurate FP approximation of $g(x)$ for any x?

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)

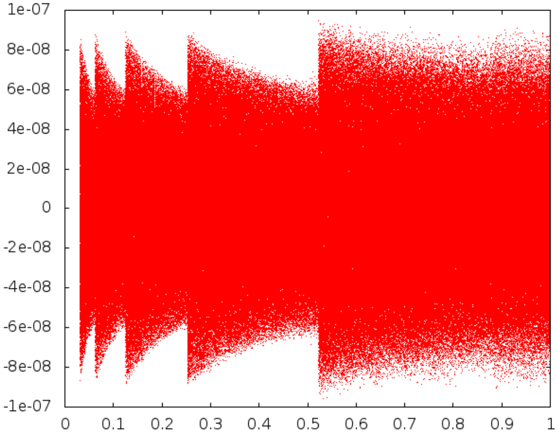
Method error: $\frac{x \cdot (1 - x^2 \cdot 10473 \cdot 2^{-16})}{\sin x} - 1.$



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.)

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

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.

Example 2: Cody-Waite Argument Reduction for Exp

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.

Example 2: Cody-Waite Argument Reduction for Exp

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.

Example 2: Cody-Waite Argument Reduction for Exp

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?

Example 2: Cody-Waite Argument Reduction for Exp

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.

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

Example 2: Cody-Waite Argument Reduction for Exp

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.

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

Example 2: Cody-Waite Argument Reduction for Exp

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?

Example 2: Cody-Waite Argument Reduction for Exp

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;
```

Example 2: Cody-Waite Argument Reduction for Exp

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.

$\square$

Example 2: Cody-Waite Argument Reduction for Exp

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 $\circ(k\ell_h) = k\ell_h$.

Example 2: Cody-Waite Argument Reduction for Exp

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. $\ell_h^{-1} \approx \mathtt{InvLog2}$, so $x \approx k\ell_h$.

$\square$

Example 2: Cody-Waite Argument Reduction for Exp

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 $t_1 = \circ(x - \circ(k\ell_h)) = x - k\ell_h$ by Sterbenz' lemma.

□

Example 2: 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;
```

Example 3: Itanium Division of 16-bit Unsigned Integers

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

Example 3: Itanium Division of 16-bit Unsigned Integers

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

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

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

Correctness of the division

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

Proof Sketch

Theorem (Exclusion zones)

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

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.
There is some error compensation.

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.
There is some error compensation.

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 2^{-17} into account,
  $\hat{q}_1/(a/b) - 1 = -\varepsilon_0^2 + (1 + \varepsilon_0) \cdot 2^{-17}$.

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;
```

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.
```

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 $e_1, \dots, e_n$,
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:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\mathrm{lower}(|U - V|), \mathrm{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- 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:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\mathrm{lower}(|U - V|), \mathrm{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- 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:
  $\mathrm{FLT}(x, p) \wedge \mathrm{FLT}(y, q) \Rightarrow \mathrm{FLT}(x \times y, p + q)$.

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:
  $|u| \in U \wedge |v| \in V \Rightarrow |u + v| \in [\mathrm{lower}(|U - V|), \mathrm{upper}(U + V)]$.
- Floating- and fixed-point arithmetic properties:
  $u \in 2^{-1074} \cdot \mathbb{Z} \Rightarrow \exists \varepsilon \in [-2^{-53}, 2^{-53}],\ \circ(u) = u \times (1 + \varepsilon)$.
- 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:
  $\mathrm{FLT}(x, p) \wedge \mathrm{FLT}(y, q) \Rightarrow \mathrm{FLT}(x \times y, p + q)$.
- 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

Outline

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs
  that handle more than just FP arithmetic, e.g. arrays.

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs
  that handle more than just FP arithmetic, e.g. arrays.
- Properly support equality.

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs
  that handle more than just FP arithmetic, e.g. arrays.
- Properly support equality.
- Benefit from native decision procedures, e.g. linear arithmetic.

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs
  that handle more than just FP arithmetic, e.g. arrays.
- Properly support equality.
- Benefit from native decision procedures, e.g. linear arithmetic.

Integration choices

- Do not implement rewriting rules that can be inferred
  by linear arithmetic.

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs
  that handle more than just FP arithmetic, e.g. arrays.
- Properly support equality.
- Benefit from native decision procedures, e.g. linear arithmetic.

Integration choices

- Do not implement rewriting rules that can be inferred
  by linear arithmetic.

Implementation features

- Incremental backward reasoning.

AltErgo + Gappa

Joint work with Sylvain Conchon and Cody Roux

Motivations

- Make it possible to verify programs that handle more than just FP arithmetic, e.g. arrays.
- Properly support equality.
- Benefit from native decision procedures, e.g. linear arithmetic.

Integration choices

- Do not implement rewriting rules that can be inferred by linear arithmetic.

Implementation features

- Incremental backward reasoning.
- Pattern-matching modulo equality.

Example 4: Knuth' TwoSum Algorithm

```
s = a + b
t = s - a
e = (a - (s - t)) + (b - t)
```

Assuming no overflow occurs, $s + e = a + b$.

Example 4: Knuth' TwoSum Algorithm

```
s = a + b
t = s - a
e = (a - (s - t)) + (b - t)
```

Assuming no overflow occurs, $s + e = a + b$.

This example is out of the reach of Gappa.
Yet its bounded instance is in the decidable fragment of FP arithmetic.

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/`