SCAN 2014
Würzburg, Germany

Formal verification of tricky numerical computations

Sylvie Boldo

Inria

September 25th, 2014

(joint work with Clément, Filliâtre, Mayero, Melquiond, Weis)

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
- My personal challenge:

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
- My personal challenge:

CORRECTNESS

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
- My personal challenge:

CORRECTNESS

- consider small critical programs, where complex properties about floating-point arithmetic are involved

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
- My personal challenge:

CORRECTNESS

- consider small critical programs, where complex properties about floating-point arithmetic are involved
- How can we get a high guarantee?

↪ formal verification

Motivations

- Scientific Computing, Computer Arithmetic and Validated Numerics
- My personal challenge:

CORRECTNESS

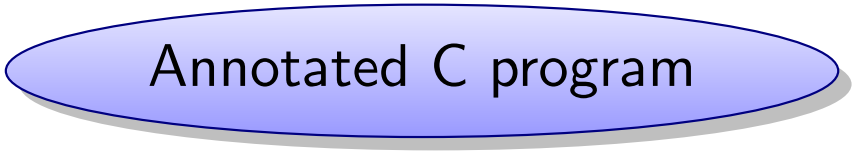
- consider small critical programs, where complex properties about floating-point arithmetic are involved
- How can we get a high guarantee?

↪ formal verification

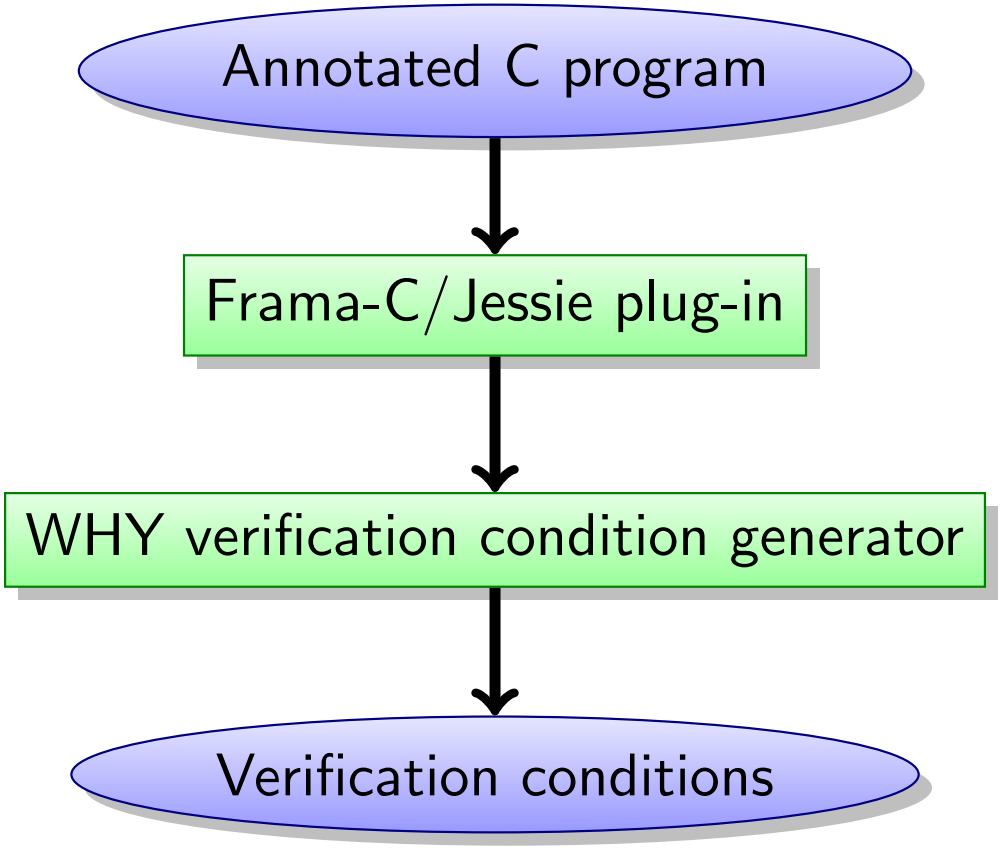
- Convince people of what is formally verified!

Outline

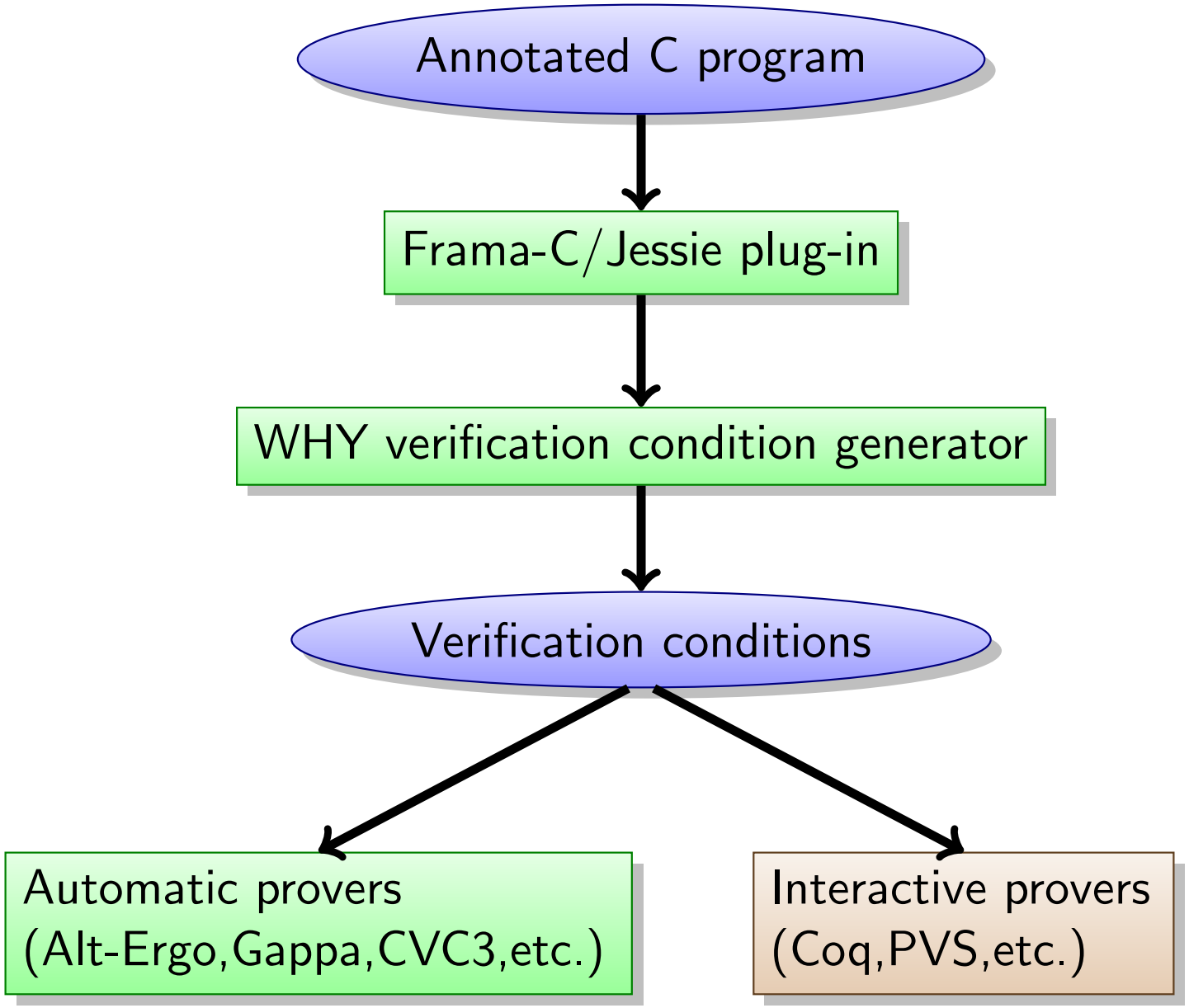
The used toolchain: Frama-C/Jessie/Why



The used toolchain: Frama-C/Jessie/Why



The used toolchain: Frama-C/Jessie/Why



Outline

Annotation language: ACSL

(how a bug differs from a rounding error)

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions
  (and which variables are modified).

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions (and which variables are modified).
- variants and invariants of the loops.

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions (and which variables are modified).
- variants and invariants of the loops.
- assertions

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions (and which variables are modified).
- variants and invariants of the loops.
- assertions
- In annotations, all computations are exact.

Annotation language: ACSL

(how a bug differs from a rounding error)

- ANSI/ISO C Specification Language
- behavioral specification language for C programs
- pre-conditions and post-conditions to functions
  (and which variables are modified).
- variants and invariants of the loops.
- assertions
- In annotations, all computations are exact.

$\Rightarrow$ **For the programmer, the specification is easy to understand.**

ACSL and floating-point numbers

A floating-point number is a triple:

- the floating-point number, really computed by the program, $x \rightarrow x_f$ floating-point part

ACSL and floating-point numbers

A floating-point number is a triple:

- the floating-point number, really computed by the program, $x \rightarrow x_f$ floating-point part
- the value that would have been obtained with exact computations, $x \rightarrow x_e$ exact part

ACSL and floating-point numbers

A floating-point number is a triple:

- the floating-point number, really computed by the program,
  $x \rightarrow x_f$ floating-point part
- the value that would have been obtained with exact computations,
  $x \rightarrow x_e$ exact part
- the value that we ideally wanted to compute
  $x \rightarrow x_m$ model part

ACSL and floating-point numbers

A floating-point number is a triple:

- the floating-point number, really computed by the program,
  $x \rightarrow x_f$ floating-point part `1+x+x*x/2`
- the value that would have been obtained with exact computations,
  $x \rightarrow x_e$ exact part $1 + x + \frac{x^2}{2}$
- the value that we ideally wanted to compute
  $x \rightarrow x_m$ model part $\exp(x)$

ACSL and floating-point numbers

A floating-point number is a triple:

- the floating-point number, really computed by the program,
  $x \rightarrow x_f$ floating-point part `1+x+x*x/2`
- the value that would have been obtained with exact computations,
  $x \rightarrow x_e$ exact part $1 + x + \frac{x^2}{2}$
- the value that we ideally wanted to compute
  $x \rightarrow x_m$ model part $\exp(x)$

$\Rightarrow$ easy to split into method error and rounding error

Outline

Formal proof

The proof is checked in its deep details until the computer agrees with it.

We often use formal proof checkers, meaning programs that only **check** a proof (they may also generate easy demonstrations).

Therefore the checker is a very short program (de Bruijn criteria: the correctness of the system as a whole depends on the correctness of a very small "kernel").

The Coq proof assistant (`http://coq.inria.fr`)

- Based on the Curry-Howard isomorphism.
  (equivalence between proofs and λ-terms)
- Few automations.
- Comprehensive libraries, including on $\mathbb{Z}$ and $\mathbb{R}$.
- Coq kernel mechanically checks each step of each proof.
- The method is to apply successively tactics (theorem application, rewriting, simplifications...) to transform or reduce the goal down to the hypotheses.
- The proof is handled starting from the conclusion.

A Coq formalization of FP arithmetic : Flocq

A FP format is only characterized by a function $\varphi : \mathbb{Z} \to \mathbb{Z}$.

A Coq formalization of FP arithmetic : Flocq

A FP format is only characterized by a function $\varphi : \mathbb{Z} \to \mathbb{Z}$.

For $x \in \mathbb{R}$, we compute e such that $\beta^{e-1} \leq |x| < \beta^e$.
Then x is in the format iff

$$x = \left\lfloor x\beta^{-\varphi(e)} \right\rfloor \beta^{\varphi(e)}$$

In other words: if it can be written with exponent $\varphi(e)$.

Usual Formats

Definition (FIX)

Fixed-point format with exponent $e_{\min}$: $\varphi(e) = e_{\min}$.

Usual Formats

Definition (FIX)

Fixed-point format with exponent $e_{\min}$: $\varphi(e) = e_{\min}$.

Definition (FL*)

Floating-point format with precision p:

- unbounded (FLX): $\varphi(e) = e - p$,

Usual Formats

Definition (FIX)

Fixed-point format with exponent $e_{\min}$: $\varphi(e) = e_{\min}$.

Definition (FL*)

Floating-point format with precision p:

- unbounded (FLX): $\varphi(e) = e - p$,
- bounded with subnormal numbers (FLT): $\varphi(e) = \max(e - p, e_{\min})$,

Usual Formats

Definition (FIX)

Fixed-point format with exponent $e_{\min}$: $\varphi(e) = e_{\min}$.

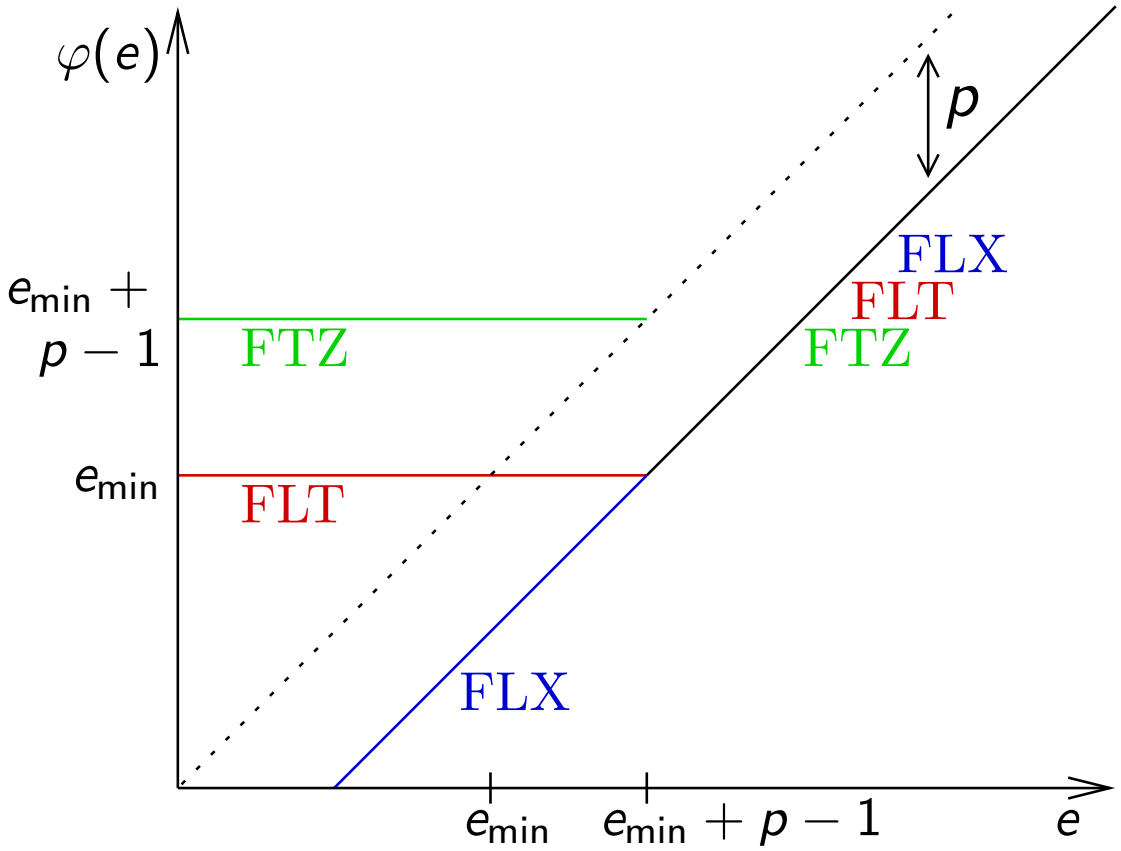
Definition (FL*)

Floating-point format with precision p:

- unbounded (FLX): $\varphi(e) = e - p$,
- bounded with subnormal numbers (FLT): $\varphi(e) = \max(e - p, e_{\min})$,
- bounded without subnormal numbers (FTZ).

A random φ may not allow to define a rounding: we have a `valid` predicate for being a reasonable φ.

Usual Floating-Point Formats



Example of Coq theorem

Theorem (round_NE_abs)

Let φ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.

Example of Coq theorem

Theorem (round_NE_abs)

Let φ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.

```
Lemma round_NE_abs:  forall x :  R,
   round beta fexp ZnearestE (Rabs x) = Rabs (round beta fexp ZnearestE x).
```

Example of Coq theorem

Theorem (round_NE_abs)

Let φ be a format, such that the rounding to nearest, ties to even ($\circ$) can be defined. For all $x \in \mathbb{R}$, $\circ(|x|) = |\circ(x)|$.

```
Lemma round_NE_abs:  forall x :  R,
  round beta fexp ZnearestE (Rabs x) = Rabs (round beta fexp ZnearestE x).
 Proof with auto with typeclass_instances.
 intros x; apply sym_eq.
 unfold Rabs at 2.
 destruct (Rcase_abs x) as [Hx|Hx].
 rewrite round_NE_opp.
 apply Rabs_left1.
 rewrite <- (round_0 beta fexp ZnearestE).
 apply round_le...
 now apply Rlt_le.
 apply Rabs_pos_eq.
 rewrite <- (round_0 beta fexp ZnearestE).
 apply round_le...
 now apply Rge_le.
 Qed.
```

With the stating of the theorem, the tactics, and the name of theorems.

More about Flocq

Flocq: 16 000 lines of Coq, 700 theorems,

- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.

More about Flocq

Flocq: 16 000 lines of Coq, 700 theorems,

- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.

Applications:

- Frama-C/Jessie — C code certifier
- CompCert — certified C compiler

`http://flocq.gforge.inria.fr/`

More about Flocq

Flocq: 16 000 lines of Coq, 700 theorems,

- any radix, any format,
- both axiomatic and computable definitions of rounding,
- effective arithmetic operators,
- numerous theorems.

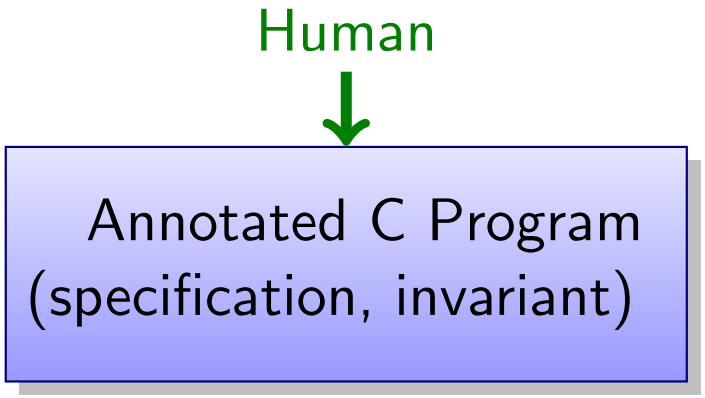
Applications:

- **Frama-C/Jessie** — **C code certifier**
- CompCert — certified C compiler

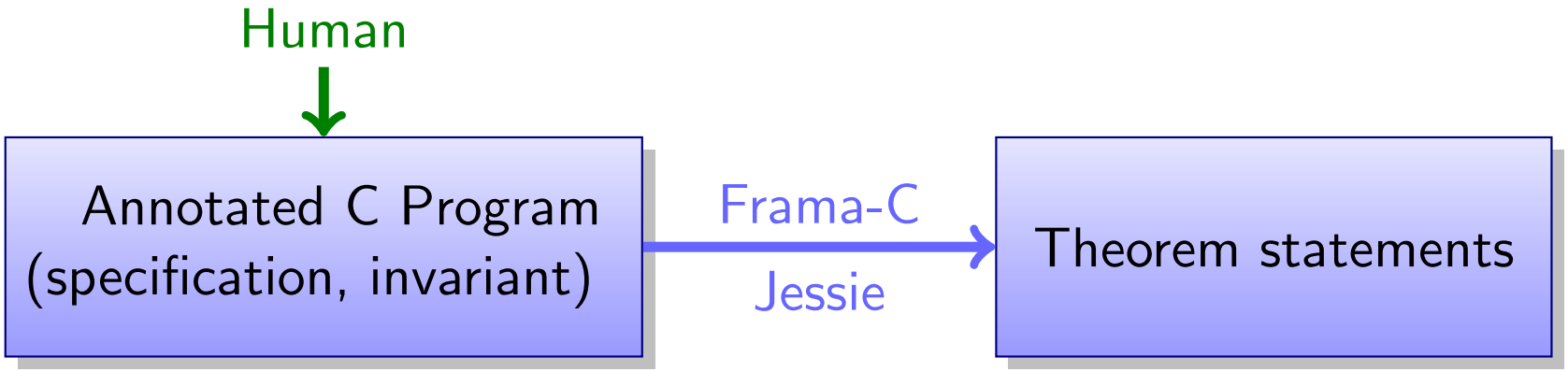
`http://flocq.gforge.inria.fr/`

Methodology for the verification of C programs

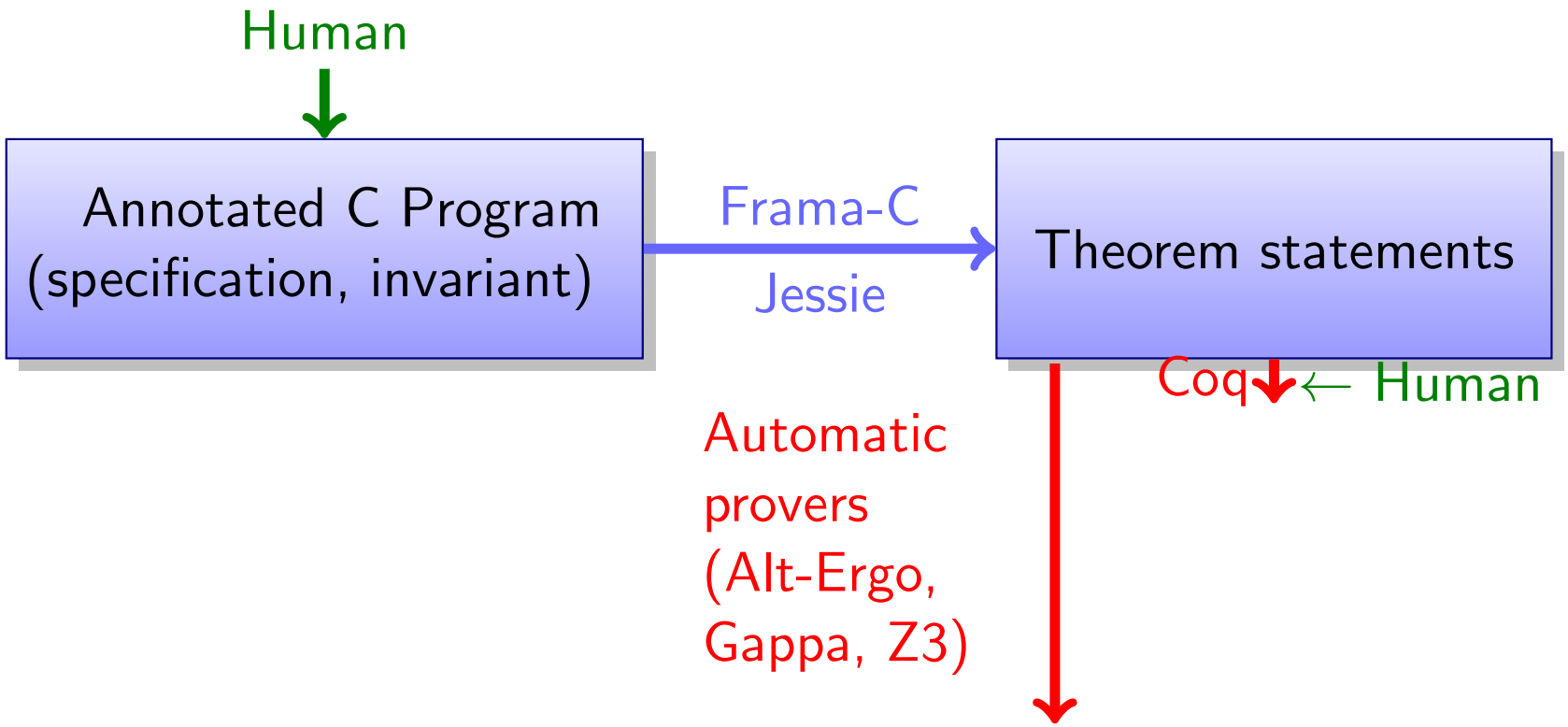
Methodology for the verification of C programs



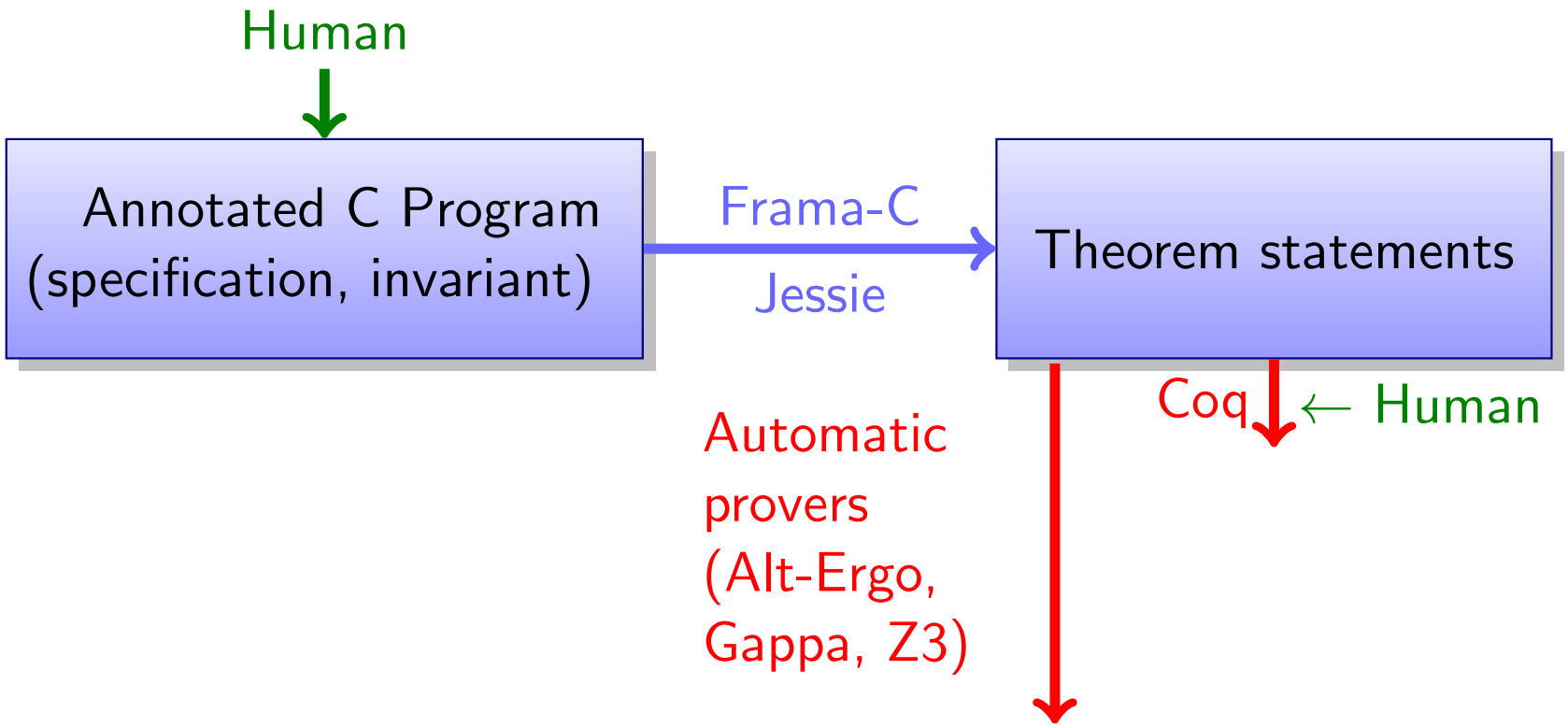
Methodology for the verification of C programs



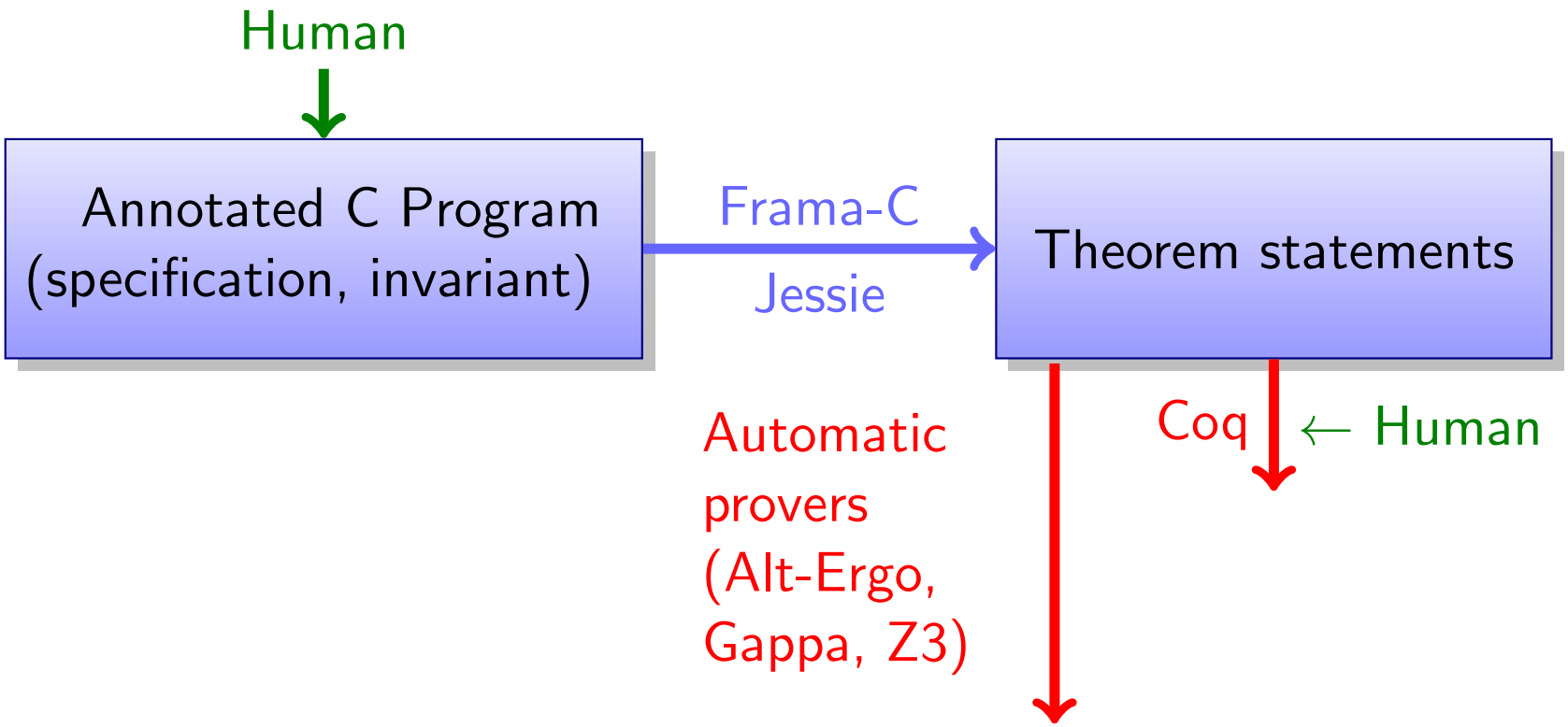
Methodology for the verification of C programs



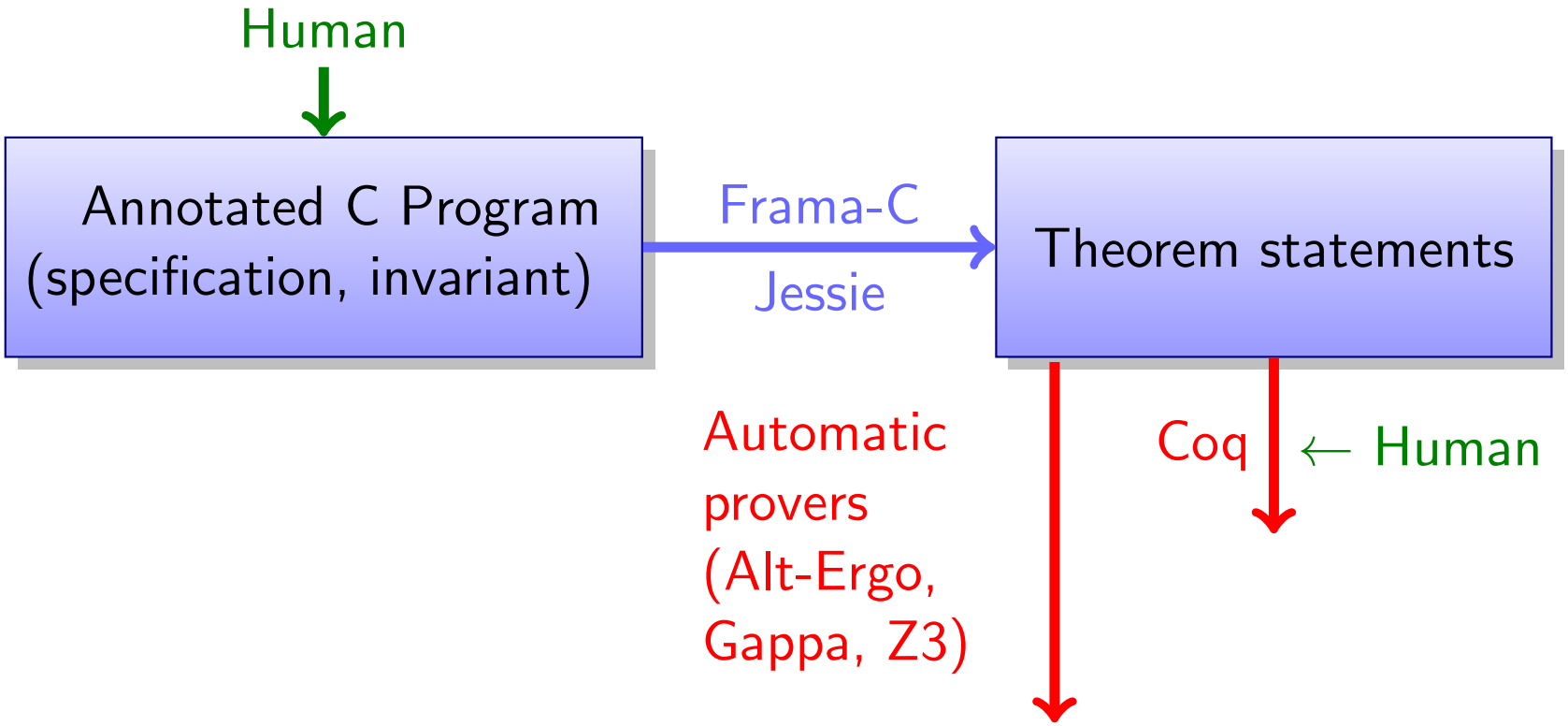
Methodology for the verification of C programs



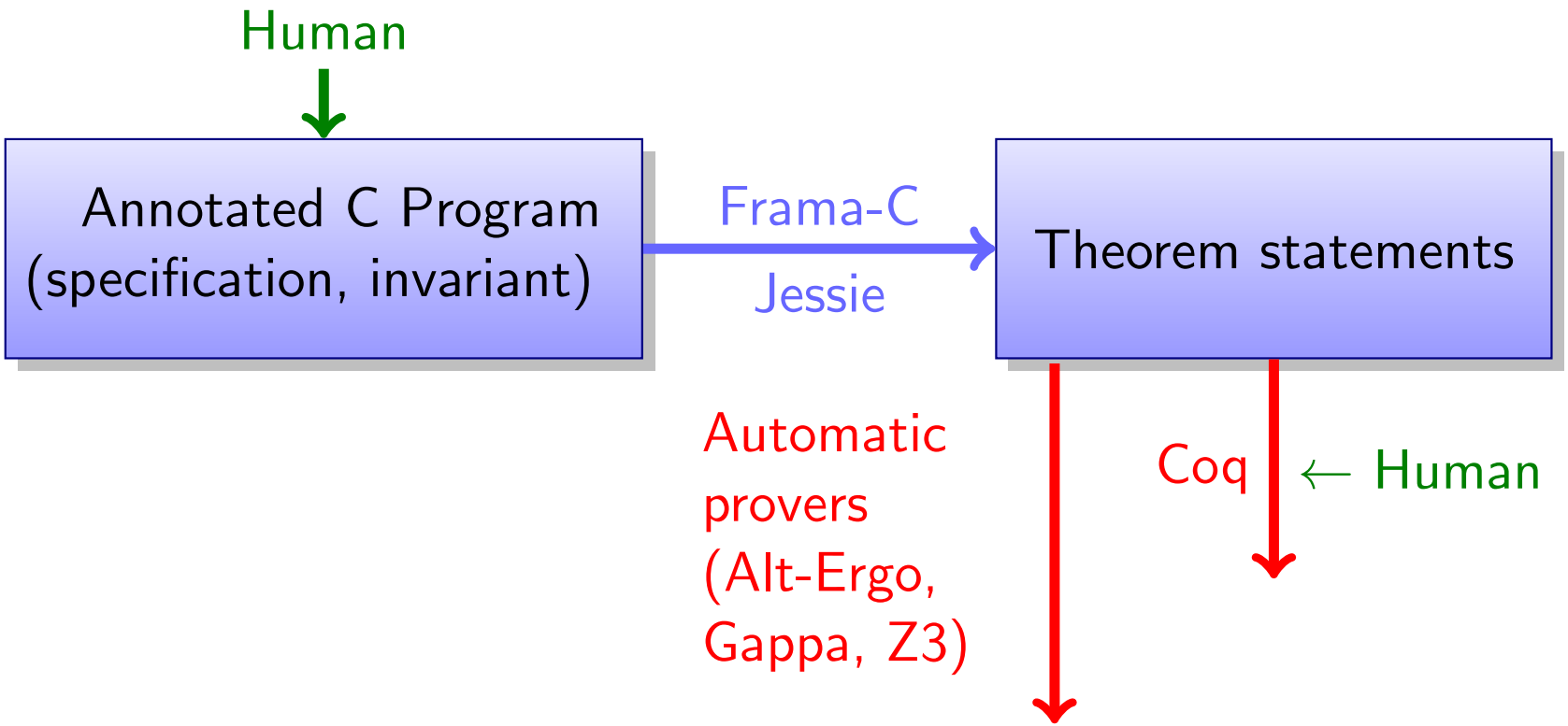
Methodology for the verification of C programs



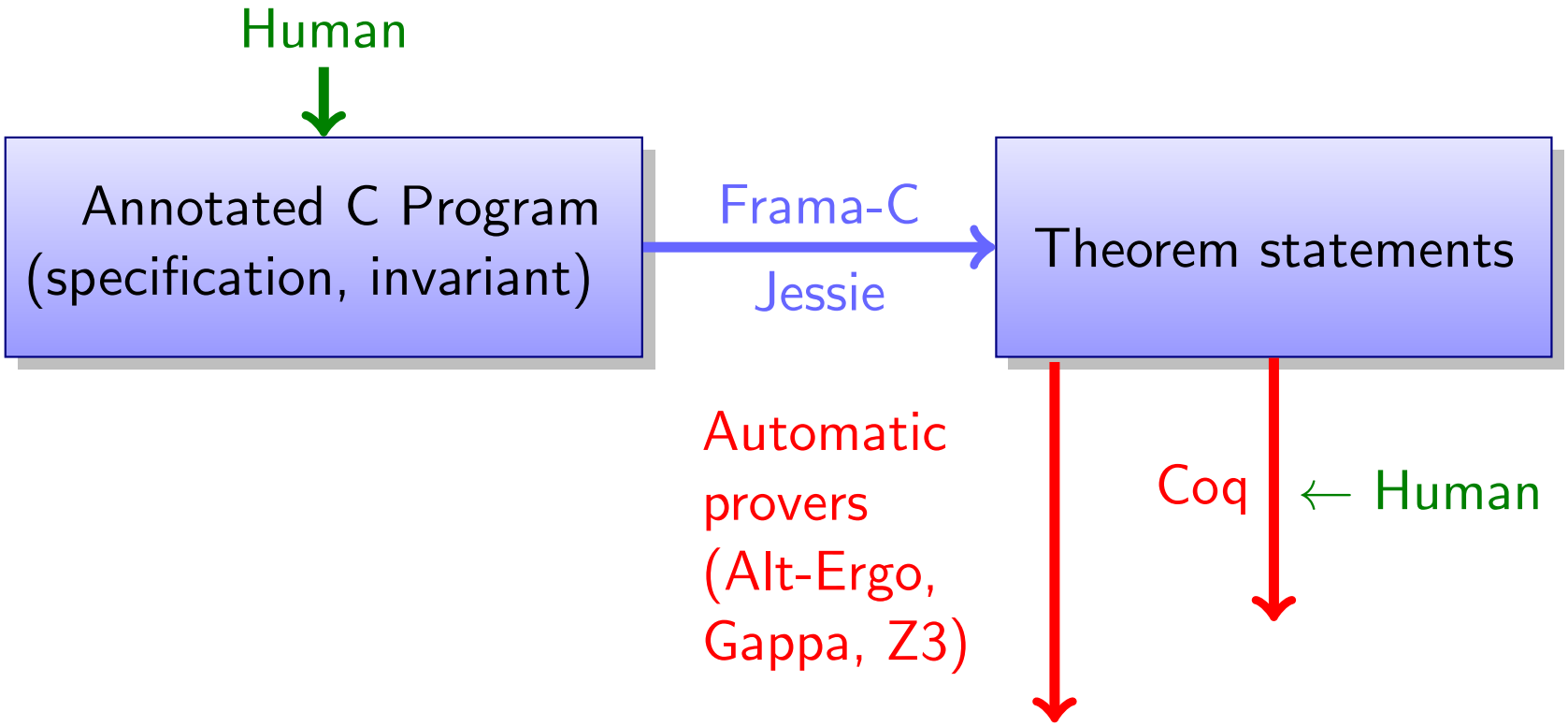
Methodology for the verification of C programs



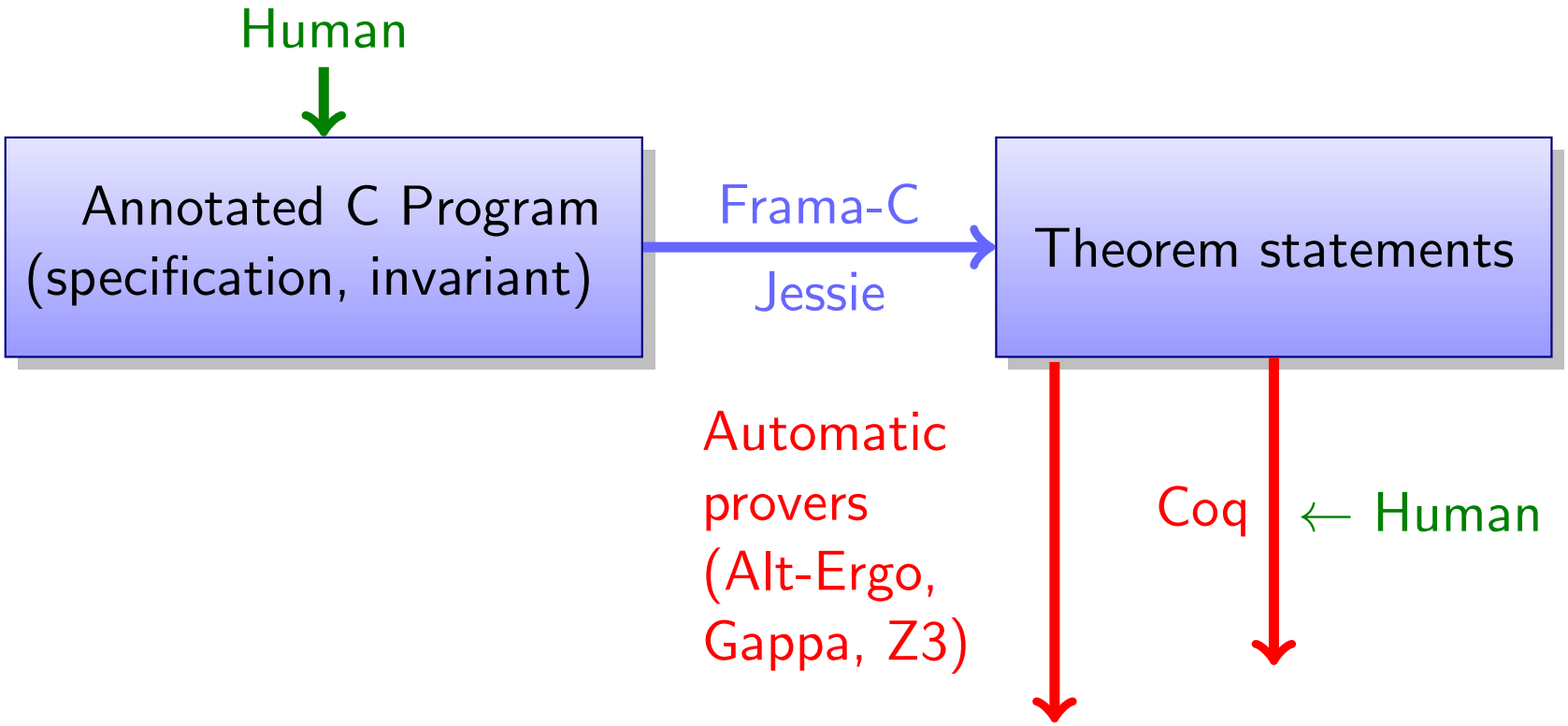
Methodology for the verification of C programs



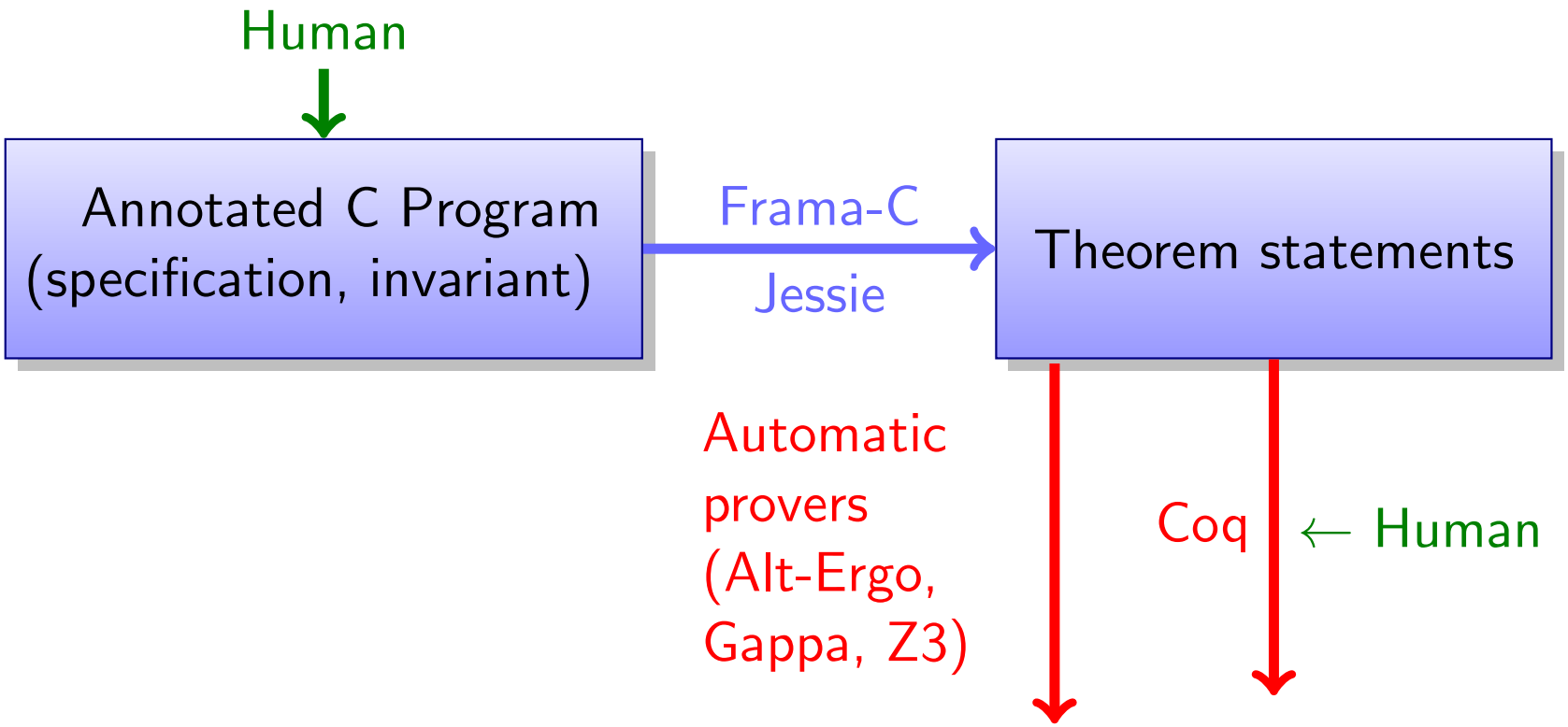
Methodology for the verification of C programs



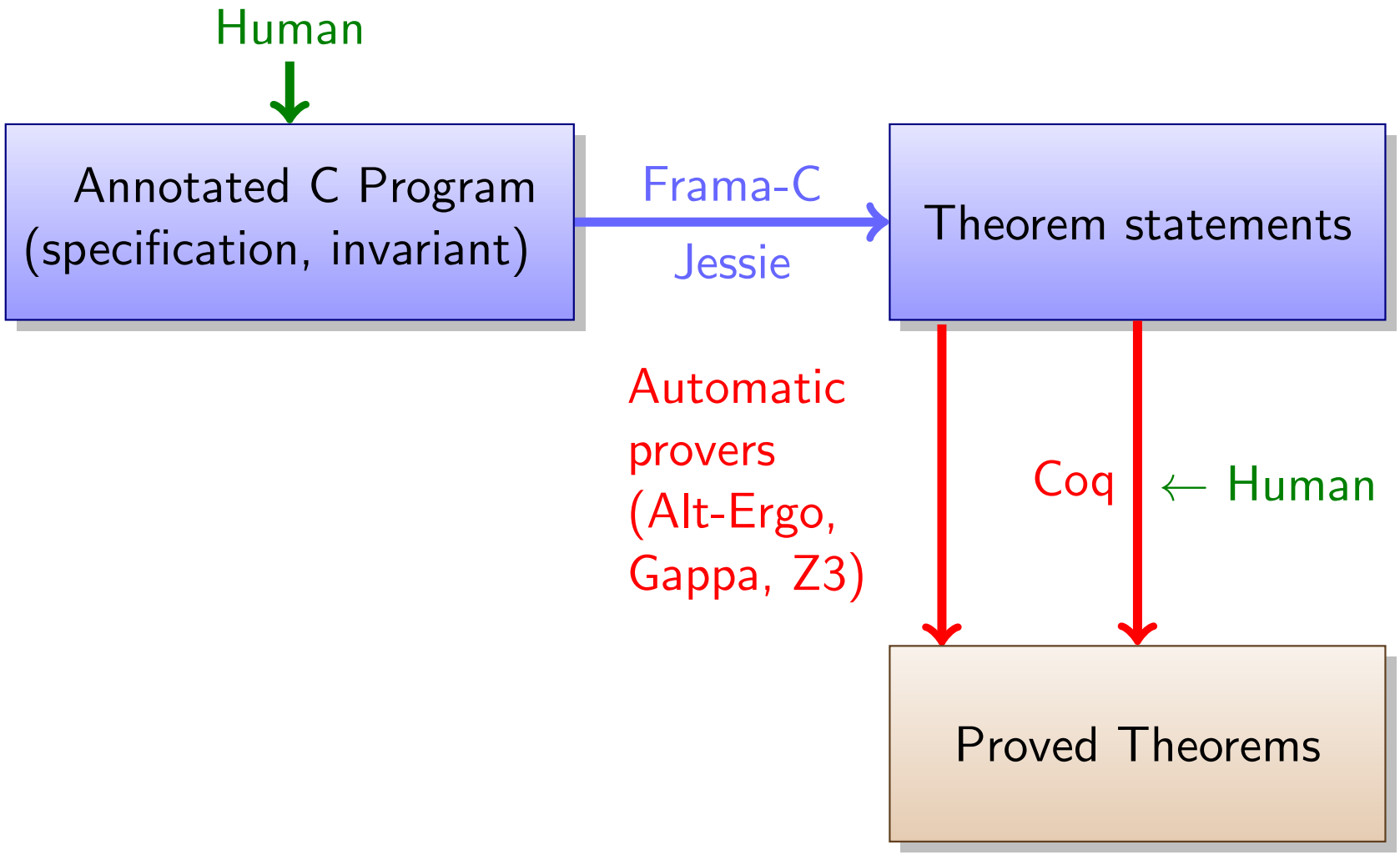
Methodology for the verification of C programs



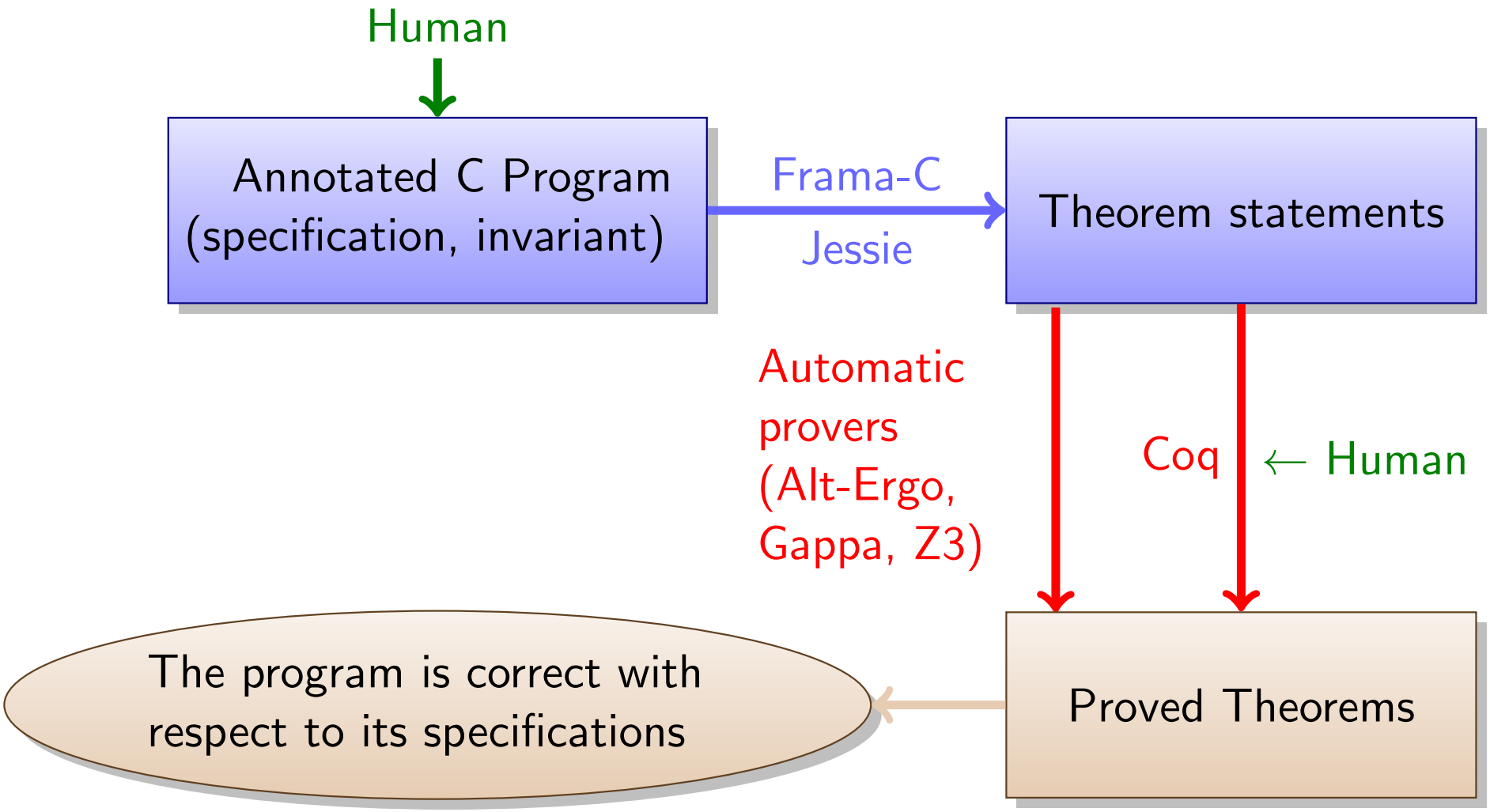
Methodology for the verification of C programs



Methodology for the verification of C programs



Methodology for the verification of C programs



Outline

Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.

Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.

Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.
- Overflow is considered a runtime error.

Examples

- All examples use Frama-C Neon, Why 2.34 and Why3 0.83.
- Non-automatic proof obligations are proved using Coq 8.4pl4.
- Overflow is considered a runtime error.
- Code & proofs available on `http://www.lri.fr/~sboldo/research.html`.

Sterbenz

Theorem (Sterbenz)

If x and y are FP numbers in a given precision such that

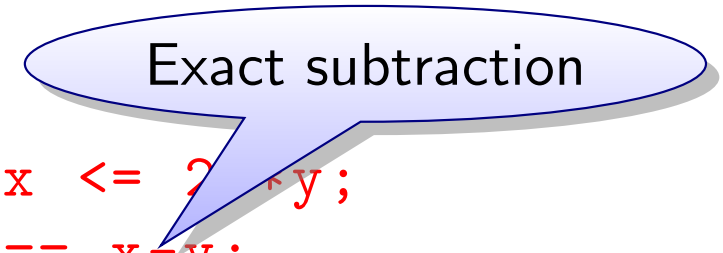
$$\frac{y}{2} \leq x \leq 2y,$$

then $x - y$ fits in a FP number in the same precision and is therefore computed without error.

Sterbenz – program

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

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

Sterbenz – program




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

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

Sterbenz – proofs

Proof obligations	**CVC3**	**Coq**	
			Nb lines
`VC for behavior`		2.34	6
`VC for safety`	0.23		

Outline

Veltkamp/Dekker

Also known as Error-Free-Transformation for the multiplication.

Theorem (Veltkamp/Dekker)

Provided no Overflow and no Underflow occur, there is an algorithm computing the exact error of the multiplication using only FP operations.

Veltkamp/Dekker

Also known as Error-Free-Transformation for the multiplication.

Theorem (Veltkamp/Dekker)

Provided no Overflow and no Underflow occur, there is an algorithm computing the exact error of the multiplication using only FP operations.

Idea:

split your floats in 2, multiply all the parts, add them in the correct order.

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

$xy = \circ(xy)$

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

Overflow

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

If no Underflow

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

Exact error of $\otimes$

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

Split x and y

Veltkamp/Dekker – program

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

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

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

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

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

Multiply all halves and add all the results

Veltkamp/Dekker – proof

Proof obligations		**Coq** Nb lines	Time
`Previous Coq proof (spec + proof)`		2639	
`VC for behavior`	`1. assertion`	3	
	`2. postcondition`	238	
`VC for safety`	`1-9. FP overflow`	1 or 2	
	`10. FP overflow`	37	
	`11. FP overflow`	47	
	`12. FP overflow`	43	
	`13. FP overflow`	64	
	`14. FP overflow`	43	
	`15. FP overflow`	83	
	`16. FP overflow`	49	
	`17. FP overflow`	94	
Total (1,248 lines spec VC excluded)		3351	9 min 02

Outline

Accurate discriminant

It is pretty hard to compute $b^2 - ac$ accurately.

Accurate discriminant

It is pretty hard to compute $b^2 - ac$ accurately.

Theorem (Kahan)

Provided no Overflow and no Underflow occur, there is an algorithm computing the $b^2 - a * c$ within 2 ulps.

Accurate discriminant

It is pretty hard to compute $b^2 - ac$ accurately.

Theorem (Kahan)

Provided no Overflow and no Underflow occur, there is an algorithm computing the $b^2 - a * c$ within 2 ulps.

Idea:

Test whether there is cancellation. If not, then use the naive algorithm. Else, compute the errors of the multiplication, and add everything in the correct order.

Accurate discriminant – program

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

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

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

Accurate discriminant – program

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

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

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

Test of cancellation

When $p \geq q$, it roughly corresponds to $p \geq 2\,q$

Accurate discriminant – program

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

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

Naive algorithm

Accurate discriminant – program

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

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

Compute the multiplication errors

Accurate discriminant – program

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

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

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

Add everything,
$p-q$ being correct.

As $\frac{q}{2} \lesssim p \lesssim 2\,q$

Accurate discriminant – program

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

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

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

Underflow

Accurate discriminant – program

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

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

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

Overflow

Accurate discriminant – program

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

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

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

2 ulps

Accurate discriminant – program

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

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

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

Function calls

⇒ pre-conditions to prove

⇒ post-conditions guaranteed

Accurate discriminant – program

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

double                                          le b, double c) {
  doub
  p=b*b;
  q=a*c;

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

In initial proof, test assumed correct

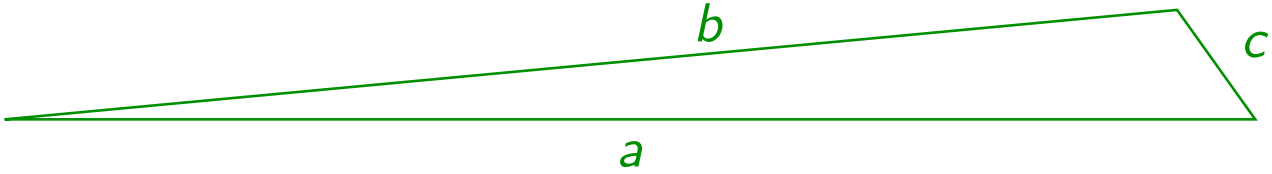
⇒ Additional proof when test is incorrect

Accurate discriminant – proof

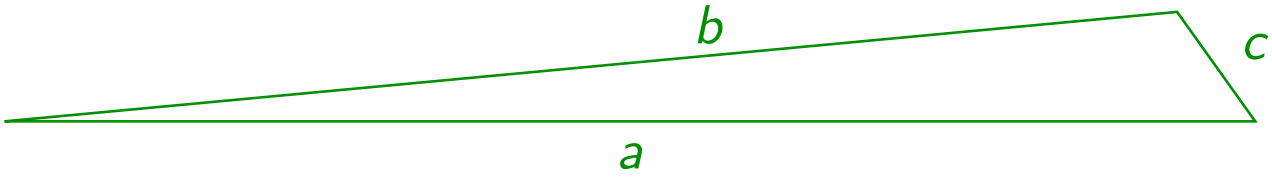
Proof obligations		**Coq**	Time
Previous Coq proof (spec + proof)		3390	
VC for theory realization		88	
Behavior	1. postcondition	61	
	2. postcondition	90	
Safety	1. floating-point overflow	2	
	2. floating-point overflow	2	
	3. floating-point overflow	3	
	4. floating-point overflow	4	
	5. floating-point overflow	4	
	6. precondition for call	2	
	7. precondition for call	9	
	8. precondition for call	1	
	9-13. precondition for call	2	
	14. floating-point overflow	44	
	15. floating-point overflow	45	
Total (1,146 lines spec VC excluded)		3655	5 min 47

Outline

Triangle area



Triangle area

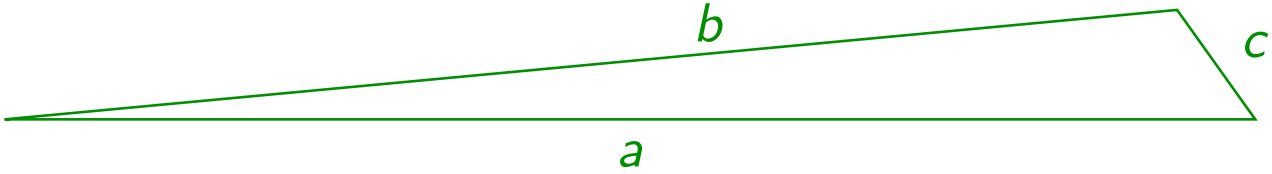


Heron's formula: $\Delta = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$ where $s = \frac{a+b+c}{2}$.

Kahan's formula, for $c \leq b \leq a$:

$$\Delta = \frac{1}{4}\sqrt{(a+(b+c))\,(c-(a-b))\,(c+(a-b))\,(a+(b-c))}.$$

Triangle area



Heron's formula: $\Delta = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$ where $s = \frac{a+b+c}{2}$.

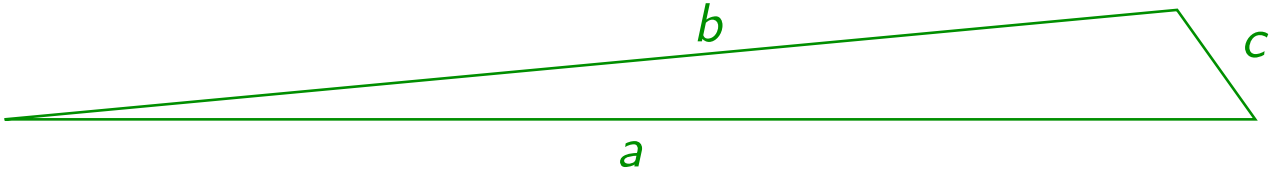
Kahan's formula, for $c \leq b \leq a$:

$$\Delta = \frac{1}{4}\sqrt{(a+(b+c))\,(c-(a-b))\,(c+(a-b))\,(a+(b-c))}.$$

[Kahan, Miscalculating Area and Angles of a Needle-like Triangle]

Area Δ is accurate to within a few units in their last digits.

Triangle area



Heron's formula: $\Delta = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$ where $s = \frac{a+b+c}{2}$.

Kahan's formula, for $c \leq b \leq a$:

$$\Delta = \frac{1}{4}\sqrt{(a+(b+c))\,(c-(a-b))\,(c+(a-b))\,(a+(b-c))}.$$

[Kahan, Miscalculating Area and Angles of a Needle-like Triangle]

Area Δ is accurate to within a few units in their last digits.

[Goldberg, 1991]

The rounding error of area Δ is at most 11 ε, provided $\varepsilon < 0.005$ and subtraction and square roots are accurate.

Triangle area

Theorem (err_Δ_flx_radix2)

With an unbounded exponent range, $\beta = 2$*, and* $\varepsilon \leq \frac{1}{100}$*, The rounding error of area* Δ *is at most* $4.75\varepsilon + 33\varepsilon^2$*.*

Triangle area

Theorem (err_Δ_flx_radix2)

With an unbounded exponent range, $\beta = 2$*, and* $\varepsilon \leq \frac{1}{100}$*, The rounding error of area* Δ *is at most* $4.75\varepsilon + 33\varepsilon^2$*.*

For underflow:

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first: $0 \leq c \ominus (a \ominus b) \leq c \oplus (a \ominus b) \leq a \oplus (b \ominus c) \leq a \oplus (b \oplus c)$

Triangle area

Theorem (err_Δ_flx_radix2)

With an unbounded exponent range, $\beta = 2$, and $\varepsilon \leq \frac{1}{100}$, The rounding error of area Δ is at most $4.75\varepsilon + 33\varepsilon^2$.

For underflow:

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first: $0 \leq c \ominus (a \ominus b) \leq c \oplus (a \ominus b) \leq a \oplus (b \ominus c) \leq a \oplus (b \oplus c)$

Theorem (err_Δ_flt_radix2)

We assume that $\beta = 2$, that $\varepsilon \leq \frac{1}{100}$, and that $2^{\left\lceil \frac{E_i+p-1}{2} \right\rceil - 2} < \Delta$. The rounding error of area Δ (computed in the given order) is at most $4.75\varepsilon + 33\varepsilon^2$.

Triangle area

Theorem (err_Δ_flx_radix2)

With an unbounded exponent range, $\beta = 2$, and $\varepsilon \leq \frac{1}{100}$, The rounding error of area Δ is at most $4.75\varepsilon + 33\varepsilon^2$.

For underflow:

- detect afterwards if a subnormal appeared in the computation
- order the intermediate variables, and multiply the biggest first: $0 \leq c \ominus (a \ominus b) \leq c \oplus (a \ominus b) \leq a \oplus (b \ominus c) \leq a \oplus (b \oplus c)$

Theorem (err_Δ_flt_radix2)

We assume that $\beta = 2$, that $\varepsilon \leq \frac{1}{100}$, and that $2^{\left\lceil \frac{E_i+p-1}{2} \right\rceil - 2} < \Delta$. The rounding error of area Δ (computed in the given order) is at most $4.75\varepsilon + 33\varepsilon^2$.

(and 5.75ε in radix 10 as multiplying by $\frac{1}{4}$ is not exact).

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);


/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);
```

Square root definition

```
/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);
```

```
/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */
```

Heron's formula

(no rounding)

```
/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);
```

```
/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */
```

Kahan's algorithm
with properly ordered values

```
double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

ordered side lengths

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);


/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */
```

overflow condition

```
/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);


/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result                    If no underflow
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);


/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @             <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */
```

Error bound

```
double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Triangle area – program

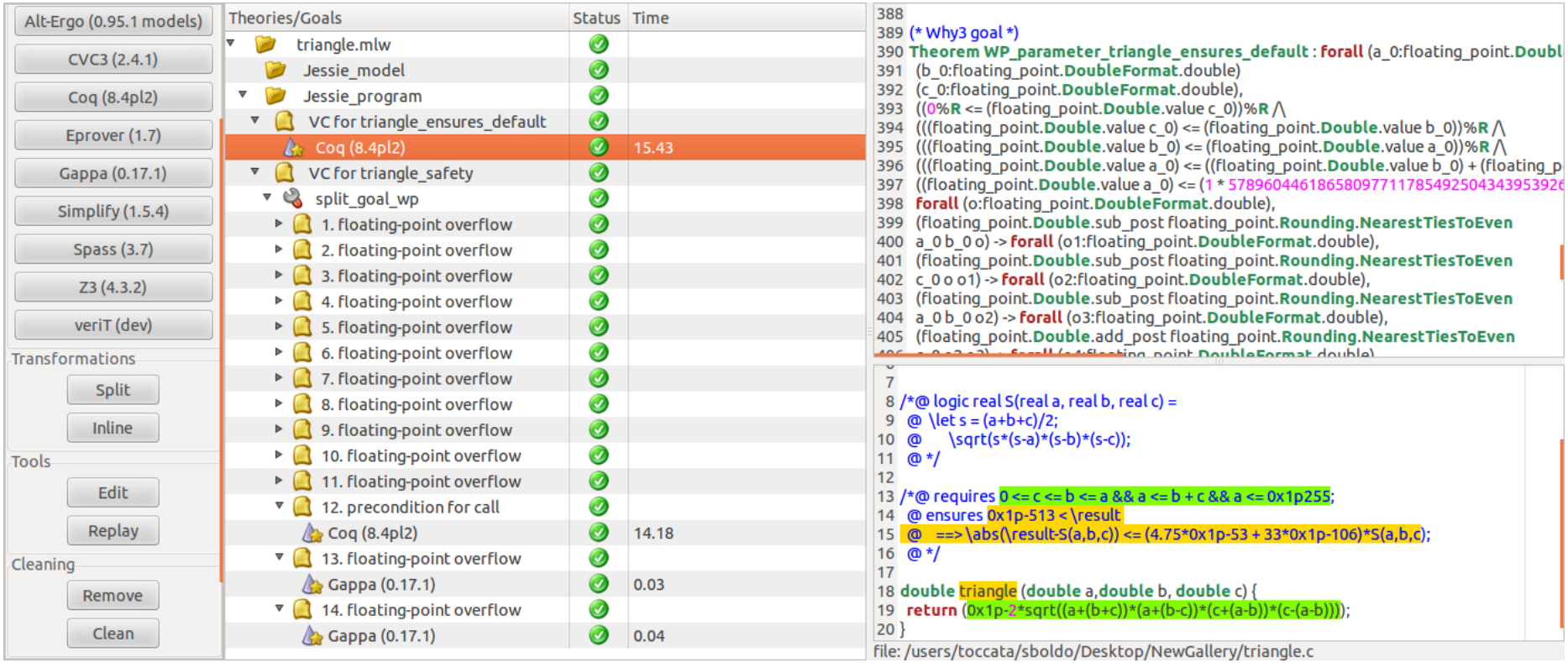
```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);


/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c))
  @              <= (                  33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Function call

Triangle area – proof

Proof obligations		Gappa	Coq	
Previous Coq proof			18.89	2091
Behavior	1. postcondition		16.00	82
Safety	1. FP overflow	0.02		
	2. FP overflow	0.03		
	3. FP overflow	0.03		
	4. FP overflow	0.03		
	5. FP overflow	0.03		
	6. FP overflow	0.00		
	7. FP overflow	0.02		
	8. FP overflow	0.01		
	9. FP overflow	0.00		
	10. FP overflow	0.02		
	11. FP overflow	0.02		
	12. precondition for call		13.22	13
	13. FP overflow	0.03		
	14. FP overflow	0.04		

Triangle area – proof



Outline

The wave equation

Looking for $u : \mathbb{R}^2 \to \mathbb{R}$ regular enough such that:

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t)$$

with given values for the initial position $u_0(x)$ and the initial velocity $u_1(x)$.

$\Rightarrow$ rope oscillation, sound, radar, oil prospection...

Scheme?

We want $u_j^k \approx u(j\Delta x, k\Delta t)$.

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

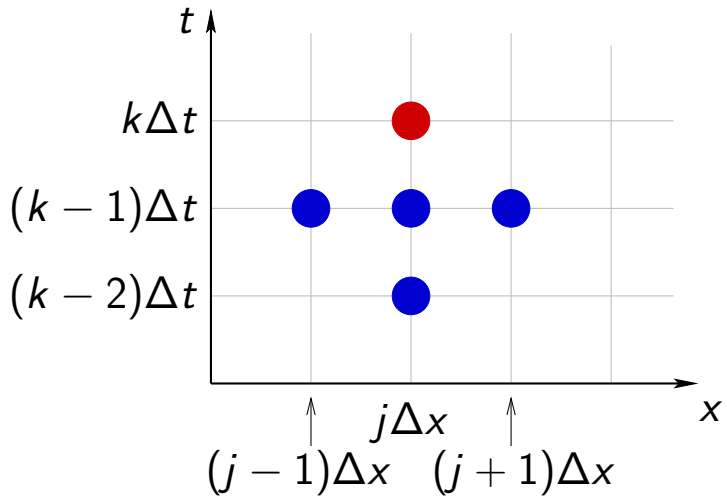
And other horrible formulas to initialize u_j^0 and u_j^1.

Scheme?

We want $u_j^k \approx u(j\Delta x, k\Delta t)$.

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

And other horrible formulas to initialize u_j^0 and u_j^1.



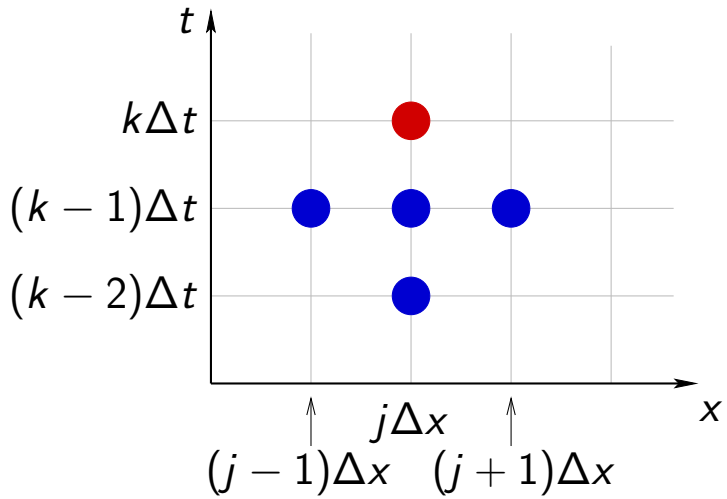
Three-point scheme: u_j^k depends on u_{j-1}^{k-1}, u_j^{k-1}, u_{j+1}^{k-1} and u_j^{k-2}.

Scheme?

We want $u_j^k \approx u(j\Delta x, k\Delta t)$.

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

And other horrible formulas to initialize u_j^0 and u_j^1.

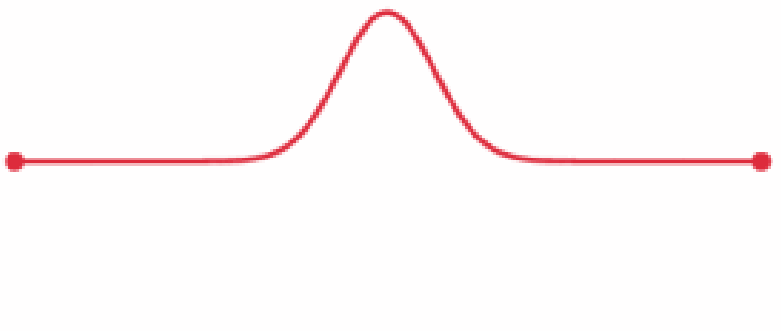


Three-point scheme: u_j^k depends on u_{j-1}^{k-1}, u_j^{k-1}, u_{j+1}^{k-1} and u_j^{k-2}.

Not really tricky computer arithmetic!

Program

```
// initialization of p[i][0] and p[i][1]
for (k=1; k<nk; k++) {
  p[0][k+1] = 0.;
  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.;
}
```



Program

```
// initialization of p[i][0] and p[i][1]
for (k=1; k<nk; k++) {
  p[0][k+1] = 0.;
  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.;
}
```

Two different errors:

- round-off errors
  due to floating-point roundings
- method errors
  the scheme only approximates the exact solution

Rounding error

Remainder:

```
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;
```

If we use a naive technique to bound the rounding errors, we get

Rounding error

Remainder:

```
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;
```

If we use a naive technique to bound the rounding errors, we get

$$|p_i^k - exact(p_i^k)| \leq O\left(2^k 2^{-53}\right)$$

Rounding error

Remainder:

```
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;
```

If we use a naive technique to bound the rounding errors, we get

$$|p_i^k - exact(p_i^k)| \leq O\left(2^k 2^{-53}\right)$$

This is too much because the errors do compensate.

Definition of ε_i^k

Remainder:

```
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;
```

Let ε_i^{k+1} be the rounding error made during these two lines of computations.

We assume a, p_{i-1}^k, p_i^k, p_{i+1}^k and p_i^{k-1} are exact and we look into the rounding error of these two lines. It is called ε_i^{k+1}.

Definition of ε_i^k

Remainder:

```
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;
```

Let ε_i^{k+1} be the rounding error made during these two lines of computations.

We assume a, p_{i-1}^k, p_i^k, p_{i+1}^k and p_i^{k-1} are exact and we look into the rounding error of these two lines. It is called ε_i^{k+1}.

We know (from initializations) that the model values of the $|p_n^m|$ are bounded by 1. We assume that the floating-point values of the $|p_n^m|$ are bounded by 2.

Definition of ε_i^k

Remainder:

```
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;
```

Let ε_i^{k+1} be the rounding error made during these two lines of computations.

We assume a, p_{i-1}^k, p_i^k, p_{i+1}^k and p_i^{k-1} are exact and we look into the rounding error of these two lines. It is called ε_i^{k+1}.

We know (from initializations) that the model values of the $|p_n^m|$ are bounded by 1. We assume that the floating-point values of the $|p_n^m|$ are bounded by 2.

$$|\varepsilon_n^m| \leq 78 \times 2^{-52}$$

Rounding error

$$p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$$

1. We have an analytical expression of the rounding error with known constants α_i^k.

Rounding error

$$p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$$

1. We have an analytical expression of the rounding error with known constants α_i^k.
2. It is not that complicated!
   (we cannot get rid of the pyramidal double summation)

Rounding error

$$p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$$

1. We have an analytical expression of the rounding error with known constants α_i^k.
2. It is not that complicated!
   (we cannot get rid of the pyramidal double summation)
3. The rounding error is bounded by $\bigcirc(k^2\ 2^{-53})$:

$$\left|\mathbf{p_i^k} - \mathbf{exact}\left(\mathbf{p_i^k}\right)\right| \leq \mathbf{78} \times \mathbf{2^{-53}} \times (\mathbf{k}+\mathbf{1}) \times (\mathbf{k}+\mathbf{2})$$

Method error

We measure that u and u_j^k are close when $(\Delta x, \Delta t) \to 0$.

We define $e_j^k \stackrel{\text{def}}{=} \bar{u}_j^k - u_j^k$: convergence error
where $\bar{u}_j^k$ is the value of u at the (j, k) point of the grid.

Method error

We measure that u and u_j^k are close when $(\Delta x, \Delta t) \to 0$.

We define $e_j^k \stackrel{\text{def}}{=} \bar{u}_j^k - u_j^k$: convergence error
where $\bar{u}_j^k$ is the value of u at the (j,k) point of the grid.

We want to bound $\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x}$: the average of the convergence error on all points of the grid at a given time $k_{\Delta t}(t) = \left\lfloor \frac{t}{\Delta t} \right\rfloor \Delta t$.

Method error

We measure that u and u_j^k are close when $(\Delta x, \Delta t) \to 0$.

We define $e_j^k \stackrel{\text{def}}{=} \bar{u}_j^k - u_j^k$: convergence error
where $\bar{u}_j^k$ is the value of u at the (j,k) point of the grid.

We want to bound $\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x}$: the average of the convergence error on all points of the grid at a given time $k_{\Delta t}(t) = \left\lfloor \frac{t}{\Delta t} \right\rfloor \Delta t$.

We want to prove:

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^2 + \Delta t^2)$$

Convergence

We proved that:

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{\substack{t \in [0, t_{\max}] \\ (\Delta x, \Delta t) \to 0 \\ 0 < \Delta x \wedge 0 < \Delta t \wedge \\ \zeta \leq c\frac{\Delta t}{\Delta x} \leq 1 - \xi}} (\Delta x^2 + \Delta t^2).$$

(This is out of the scope of this talk.)

Extraction of the big O constants

The preceding result is a uniform big O defined by:

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

Extraction of the big O constants

The preceding result is a uniform big O defined by:

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

Let (α_3, C_3) be the constants for the order-3 Taylor development of the exact solution and (α_4, C_4) for order-4. The initial support is $[\chi_1; \chi_2]$.

Extraction of the big O constants

The preceding result is a uniform big O defined by:

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

Let (α_3, C_3) be the constants for the order-3 Taylor development of the exact solution and (α_4, C_4) for order-4. The initial support is $[\chi_1; \chi_2]$.

$$\begin{aligned}
\alpha &= \min(\alpha_3, \alpha_4, 1, t_{\max}) \\
s_1 &= \max(1, 2 \cdot C_4 \cdot (c^2 + 1), C_3 \cdot (1 + c^2/2) + 1) \\
s_2 &= s_1^2 \left(\lfloor \chi_2 \rfloor - \lfloor \chi_1 \rfloor + 2 \cdot c \cdot t_{\max} \cdot \left(1 + \frac{1}{\zeta} \right) + 3 \right) \\
s_3 &= \frac{1}{\sqrt{2}} \left(C_3 \cdot (1 + c^2/2) + 1 \right) \cdot (\chi_2 - \chi_1 + 1 + (2 \cdot c + 4)) \\
&\quad + \frac{\sqrt{2}}{2\sqrt{2\xi - \xi^2}} (2 \cdot t_{\max} \cdot s_2 + 2 s_2) \\
C &= \frac{\sqrt{2}}{\sqrt{2\xi - \xi^2}} \cdot 2 \cdot t_{\max} \cdot s_3
\end{aligned}$$

Program verification

- 154 lines of annotations for 32 lines of C
- 150 verification conditions:
  - 44 about the behavior
  - 106 about the safety (runtime errors)

Program verification

- 154 lines of annotations for 32 lines of C
- 150 verification conditions:
  - 44 about the behavior
  - 106 about the safety (runtime errors)

Prover	Behavior VC	Safety VC	Total
Alt-Ergo	18	80	98
CVC3	18	89	107
Gappa	2	20	22
Z3	21	63	84
Automatically proved	**23**	**94**	**117**
Coq	**21**	**12**	**33**
Total	44	106	150

Program verification

- About 90 % of the safety goals (matrix access, Overflow, and so on) are proved automatically.
- 33 theorems are interactively proved using Coq for a total of about 15,000 lines of Coq and 30 minutes of compilation.

Type of proofs	Nb spec lines	Nb lines	Compilation time
Convergence	991	5 275	42 s
Round-off + runtime errors	7 737	13 175	32 min

Outline

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
- formal proofs are possible because algorithms are small

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
- formal proofs are possible because algorithms are small
- (Have you seen long tricky algorithms?)

Conclusion

- always a Coq proof, generic wrt precision and minimal exponent (and often radix)
- but also an annotated C program that handles exceptional behavior (e. g. Overflow, division by zero)
- formal proofs are required because algorithms are tricky
- formal proofs are possible because algorithms are small
- (Have you seen long tricky algorithms?)

- not applicable on big (naive) industrial algorithms

Conclusion

- Very high guarantee

Conclusion

- Very high guarantee
- not only rounding errors:

Conclusion

- Very high guarantee
- not only rounding errors:
  - all other errors such as pointer dereferencing or division by zero

Conclusion

- Very high guarantee
- not only rounding errors:
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties

Conclusion

- Very high guarantee
- not only rounding errors:
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties
  - any property can be checked

Conclusion

- Very high guarantee
- not only rounding errors:
  - all other errors such as pointer dereferencing or division by zero
  - link with mathematical properties
  - any property can be checked
- expressive annotation language (as expressive as Coq)
  $\Rightarrow$ exactly the specification you want

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

- Solution 1: cover all cases.

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

- Solution 1: cover all cases.

  only use forward analysis with a slightly larger bound
  (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

- Solution 1: cover all cases.

  only use forward analysis with a slightly larger bound
  (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)

- Solution 2: look into the assembly, and prove what is compiled.

Limits: compilation

- We assume all `double` operations are direct 64-bits roundings.
- On recent processors, we have x86 extended registers (80-bits long) and FMA ($\circ(ax + b)$ with one single rounding).

$\Rightarrow$ several possible results!

- Solution 1: cover all cases.

  only use forward analysis with a slightly larger bound
  (it covers, 64-bit, 80-bit, double roundings and all uses of FMA)

- Solution 2: look into the assembly, and prove what is compiled.

- Solution 3: use a certified compiler, then compilation is specified.

Perspectives

- a better handling of exceptional behaviors

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

$\Rightarrow$ basic blocks to build upon

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

$\Rightarrow$ basic blocks to build upon

- prove libraries with computational contents
  (e.g. computational geometry)

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

$\Rightarrow$ basic blocks to build upon

- prove libraries with computational contents (e.g. computational geometry)

- go deeper into numerical analysis

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

$\Rightarrow$ basic blocks to build upon

- prove libraries with computational contents (e.g. computational geometry)

- go deeper into numerical analysis

$\Rightarrow$ e.g. finite elements

Perspectives

- a better handling of exceptional behaviors
- prove and generalize well-known facts/algorithms/programs from the computer arithmetic community

$\Rightarrow$ basic blocks to build upon

- prove libraries with computational contents (e.g. computational geometry)

- go deeper into numerical analysis

$\Rightarrow$ e.g. finite elements

$\Rightarrow$ e.g. stability

This is not a slide.

Big O = big pain

Usually, the big O uses one variable and $f(\mathbf{x}) = O_{\|\mathbf{x}\|\to 0}(g(\mathbf{x}))$ means

$$\exists \alpha, C > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \quad \|\mathbf{x}\| \leq \alpha \Rightarrow |f(\mathbf{x})| \leq C \cdot |g(\mathbf{x})|.$$

Big O = big pain

Usually, the big O uses one variable and $f(\mathbf{x}) = O_{\|\mathbf{x}\|\to 0}(g(\mathbf{x}))$ means

$$\exists \alpha, C > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \quad \|\mathbf{x}\| \leq \alpha \Rightarrow |f(\mathbf{x})| \leq C \cdot |g(\mathbf{x})|.$$

Here 2 variables: $\boldsymbol{\Delta}\mathbf{x}$ (grid sizes, tends to 0), and $\mathbf{x}$ (time and space).
(Think about Taylor expansions)

Big O = big pain

Usually, the big O uses one variable and $f(\mathbf{x}) = O_{\|\mathbf{x}\|\to 0}(g(\mathbf{x}))$ means

$$\exists \alpha, C > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \quad \|\mathbf{x}\| \leq \alpha \Rightarrow |f(\mathbf{x})| \leq C \cdot |g(\mathbf{x})|.$$

Here 2 variables: $\mathbf{\Delta x}$ (grid sizes, tends to 0), and $\mathbf{x}$ (time and space).
(Think about Taylor expansions)

$$\forall \mathbf{x}, \exists \alpha, C > 0, \quad \forall \mathbf{\Delta x} \in \mathbb{R}^2, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|$$

does not work.

Uniform big O

We used a uniform big O:

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

where variables $\mathbf{x}$ and $\mathbf{\Delta x}$ are restricted to subsets of $\mathbb{R}^2$.
(for example such that $\Delta t > 0$)
$\Rightarrow$ Taylor expansions

Proof idea 1/3: consistency

The truncation error is defined as how much the exact solution solves the numerical scheme:

$$\varepsilon_j^{k-1} = \frac{\bar{u}_j^k - 2\bar{u}_j^{k-1} + \bar{u}_j^{k-2}}{\Delta t^2} - c^2 \frac{\bar{u}_{j+1}^{k-1} - 2\bar{u}_j^{k-1} + \bar{u}_{j-1}^{k-1}}{\Delta x^2} - s_j^{k-1}$$

Proof idea 1/3: consistency

The truncation error is defined as how much the exact solution solves the numerical scheme:

$$\varepsilon_j^{k-1} = \frac{\bar{u}_j^k - 2\bar{u}_j^{k-1} + \bar{u}_j^{k-2}}{\Delta t^2} - c^2 \frac{\bar{u}_{j+1}^{k-1} - 2\bar{u}_j^{k-1} + \bar{u}_{j-1}^{k-1}}{\Delta x^2} - s_j^{k-1}$$

The consistency is the boundedness of the truncation error:

$$\left\| \varepsilon_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^2 + \Delta t^2)$$

By Taylor series and many computations.

Proof idea 2/3: stability

We define a discrete energy by

$$E_h(c)(u_h)^{k+\frac{1}{2}} \stackrel{\text{def}}{=} \frac{1}{2}\left\|\frac{u_h^{k+1}-u_h^k}{\Delta t}\right\|_{\Delta x}^2 + \frac{1}{2}\left\langle u_h^k, u_h^{k+1}\right\rangle_{A_h(c)}$$

kinetic energy potential energy

$$\left\langle v_h, w_h\right\rangle_{A_h(c)} \stackrel{\text{def}}{=} \left\langle A_h(c)\, v_h, w_h\right\rangle_{\Delta x} \text{ and } \left(A_h(c)\, v_h\right)_j \stackrel{\text{def}}{=} -c^2\,\frac{v_{j+1}-2v_j+v_{j-1}}{\Delta x^2}.$$

Proof idea 2/3: stability

We define a discrete energy by

$$E_h(c)(u_h)^{k+\frac{1}{2}} \stackrel{\text{def}}{=} \frac{1}{2}\left\|\frac{u_h^{k+1}-u_h^k}{\Delta t}\right\|_{\Delta x}^2 + \frac{1}{2}\left\langle u_h^k, u_h^{k+1}\right\rangle_{A_h(c)}$$

kinetic energy potential energy

$$\left\langle v_h, w_h\right\rangle_{A_h(c)} \stackrel{\text{def}}{=} \left\langle A_h(c)\, v_h, w_h\right\rangle_{\Delta x} \text{ and } \left(A_h(c)\, v_h\right)_j \stackrel{\text{def}}{=} -c^2\,\frac{v_{j+1}-2v_j+v_{j-1}}{\Delta x^2}.$$

Note that this energy is constant if $f = 0$.
We prove an overestimation and an underestimation of this energy.
$\Rightarrow$ u_h does not diverge.

Proof idea 3/3: convergence

The convergence error is solution of the same discrete scheme with inputs

$$u_{0,j} = 0, \qquad u_{1,j} = \frac{e_j^1}{\Delta t}, \qquad \text{and} \qquad s_j^k = \varepsilon_j^{k+1}.$$

+ proofs about the initializations.

Proof idea 3/3: convergence

The convergence error is solution of the same discrete scheme with inputs

$$u_{0,j} = 0, \qquad u_{1,j} = \frac{e_j^1}{\Delta t}, \qquad \text{and} \qquad s_j^k = \varepsilon_j^{k+1}.$$

+ proofs about the initializations.

All these proofs require the existence of ζ and ξ in $]0,1[$ with $\zeta \leq 1 - \xi$ and we require that $\zeta \leq \frac{c\Delta t}{\Delta x} \leq 1 - \xi$ (CFL conditions).