Recent Advances in Floating-point Static Analyses

(in a very general sense)

Eva Darulova

Uppsala university

Finite precision

- models of the physical world, (control) algorithms, etc. assume **real-valued** arithmetic

$$x_1, x_2, x_3 \in \mathbb{R}$$

$$-x_1 * x_2 - 2x_2x_3 - x_1 - x_3$$

- exact computation not always feasible (e.g. for sine) or is expensive
- computer implementations need finite precision, e.g. **floating-point arithmetic**

```
def rigidBody(x1: Double, x2: Double, x3: Double): Double =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```

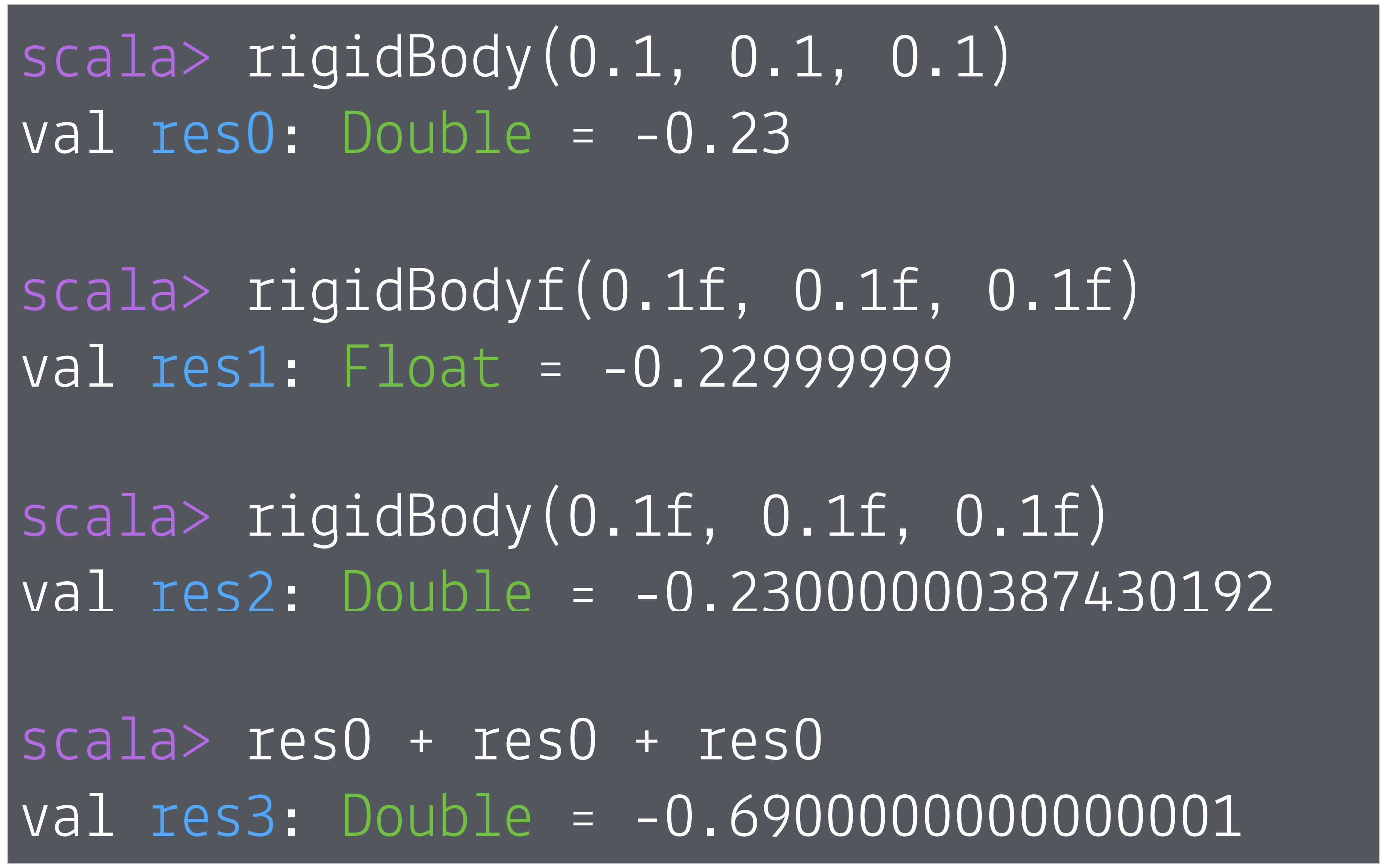
```
def rigidBodyf(x1: Float, x2: Float, x3: Float): Float =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```

Finite precision

- computer implementations need **finite precision**, e.g. floating-point arithmetic
- finite precision introduces **rounding errors**

```
def rigidBody(x1: Double, x2: Double, x3: Double): Double =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```

```
def rigidBodyf(x1: Float, x2: Float, x3: Float): Float =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```



```
scala> rigidBody(0.1, 0.1, 0.1)
val res0: Double = -0.23

scala> rigidBodyf(0.1f, 0.1f, 0.1f)
val res1: Float = -0.22999999

scala> rigidBody(0.1f, 0.1f, 0.1f)
val res2: Double = -0.2300000387430192

scala> res0 + res0 + res0
val res3: Double = -0.6900000000000001
```

Finite precision

- computer implementations need **finite precision**, e.g. floating-point arithmetic
- finite precision introduces **rounding errors**
- rounding **breaks mathematical identities**

```
def rigidBody(x1: Double, x2: Double, x3: Double): Double =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```

```
def rigidBodyf(x1: Float, x2: Float, x3: Float): Float =
  -x1 * x2 - 2 * x2 * x3 - x1 - x3
```

```
def rigidBodyf2(x1: Float, x2: Float, x3: Float): Float =
  (-x1 * x2 - (x1 + x3)) - (x2 * 2 * x3)
```

```
scala> rigidBody(0.1, 0.1, 0.1)
val res0: Double = -0.23

scala> rigidBodyf(0.1f, 0.1f, 0.1f)
val res1: Float = -0.22999999

scala> rigidBody(0.1f, 0.1f, 0.1f)
val res2: Double = -0.2300000387430192

scala> res0 + res0 + res0
val res3: Double = -0.6900000000000001

scala> rigidBodyf2(0.1f, 0.1f, 0.1f)
val res4: Float = -0.23

scala> rigidBody(0.1, 0.1, 0.1/0.0)
val res4: Double = -Infinity
```

Dealing with errors

Xavier Leroy:

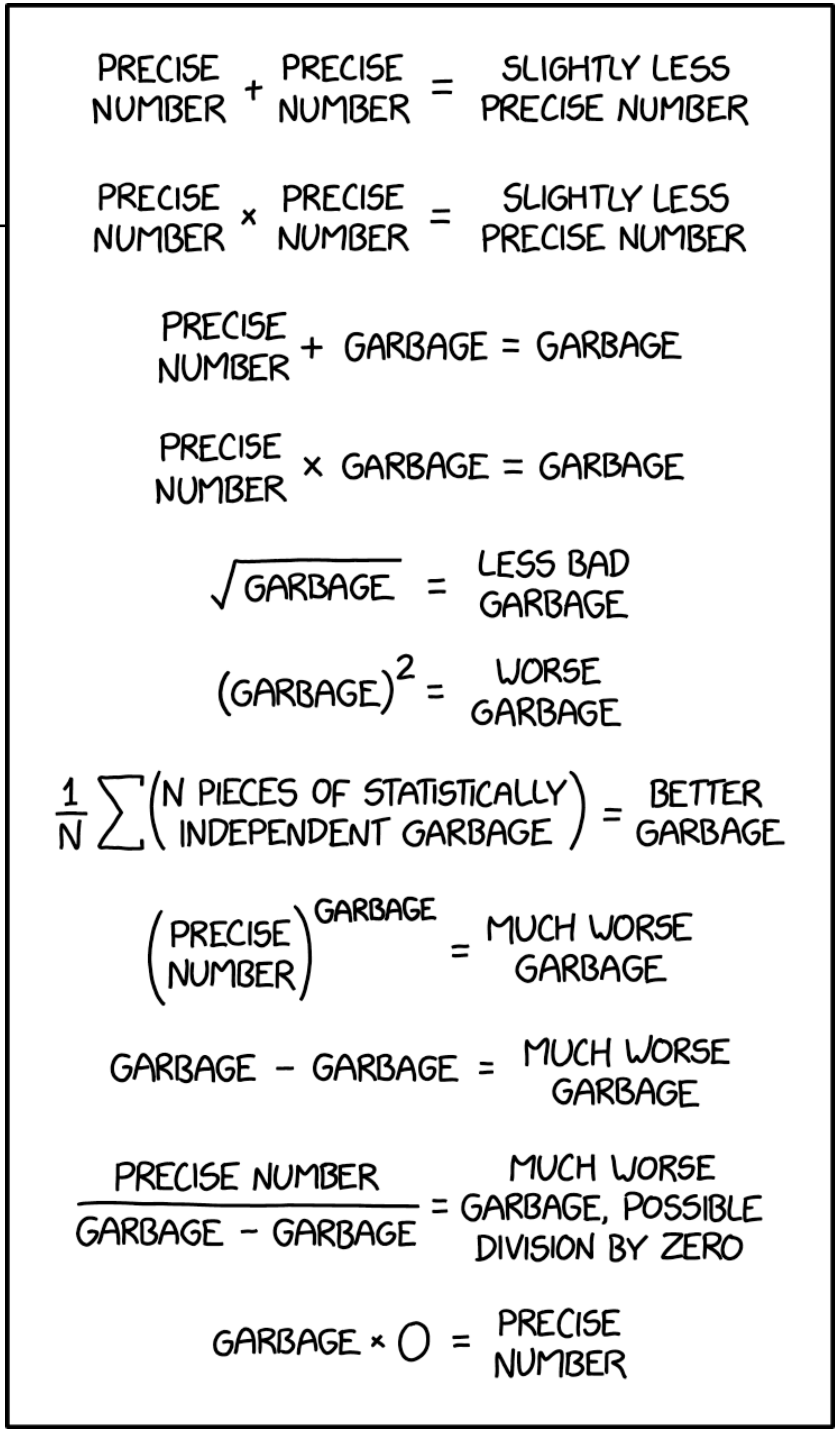
> "It makes us nervous to fly an airplane since we know they OPERATE using floating-point arithmetic."

Verified squared: does critical software deserve verified tools? Talk at POPL, 2011.

Need: **Rigorous correctness guarantees**

Are we there yet?

Spoiler: No



https://xkcd.com/2295/

This talk: where are we and why is it so hard?

Background on floating-point arithmetic *(real quick)*

Floating-points in KeY

Deductive Verification of Floating-Point Java Programs in KeY, TACAS'21 and STTT'23

Tutorial on rounding error analysis *(by example)*

Recent work in rounding error analysis

Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs, SAS'23

This talk: where are we and why is it so hard?

Background on floating-point arithmetic *(real quick)*

Floating-points in KeY

Deductive Verification of Floating-Point Java Programs in KeY, TACAS’21 and STTT’23

Tutorial on rounding error analysis *(by example)*

Recent work in rounding error analysis

Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs, SAS’23

IEEE 754 floating-point standard

Representation: $m \cdot 2^e$

- base 2 (base 10 also possible)
- m: mantissa, $|m| < 1$
- e: exponent

precision	m bits	e bits	ϵ	δ
half (16)	11	5	$2^{-11} \approx$ 4.88e-04	2^{-25}
single (32)	24	8	$2^{-24} \approx$ 5.96e-08	2^{-150}
double (64)	53	11	$2^{-53} \approx$ 1.11e-16	2^{-1075}

Arithmetic operations: computed as if with real arithmetic and then **rounded**

- Different rounding modes: **to nearest** (default), to 0, to +/- Infinity
- **Abstraction** for arithmetic operations and rounding to nearest:

$$\tilde{op} = op(1 + e) + d \quad \text{where } |e| \leq \epsilon, |d| \leq \delta$$

Special values

Representation of normal values: $m \cdot 2^e$

Special values: `+Infinity`, `-Infinity`, `+0.0`, `-0.0`, NaN (Not-a-Number)

- underflow → `+0.0` or `-0.0`
- overflow → `Infinity` or `-Infinity`
- `1.0 / 0.0` → `Infinity`
- `sqrt(-42.0)` → `NaN`
- `NaN * 0.0` → `NaN`
- `NaN == NaN` → `false`

typically, special values signal an error

Consequence of rounding and special values

Floating-point arithmetic is commutative, but **not associative or distributive:**

```
x + (y + z) != (x + y) + z

x * (y * z) != (x * y) * z

x * (y + z) != (x * y) + (x * z)
```

Other real-valued identities also do not hold:

```
x / 10 != x * 0.1

x == y ⇏ 1/x == 1/y

x != x
```

When analyzing code, need to **follow exact order of computation.**

This talk

Background on floating-point arithmetic *(real quick)*

Floating-points in KeY

Deductive Verification of Floating-Point Java Programs in KeY, TACAS’21 and STTT’23

joint work with Rosa Abbasi, Mattias Ulbrich, Jonas Schiffl, Wolfgang Ahrendt

Tutorial on rounding error analysis *(by example)*

Recent work in rounding error analysis

Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs, SAS’23

Goal: prove absence of *runtime errors* and *special values*

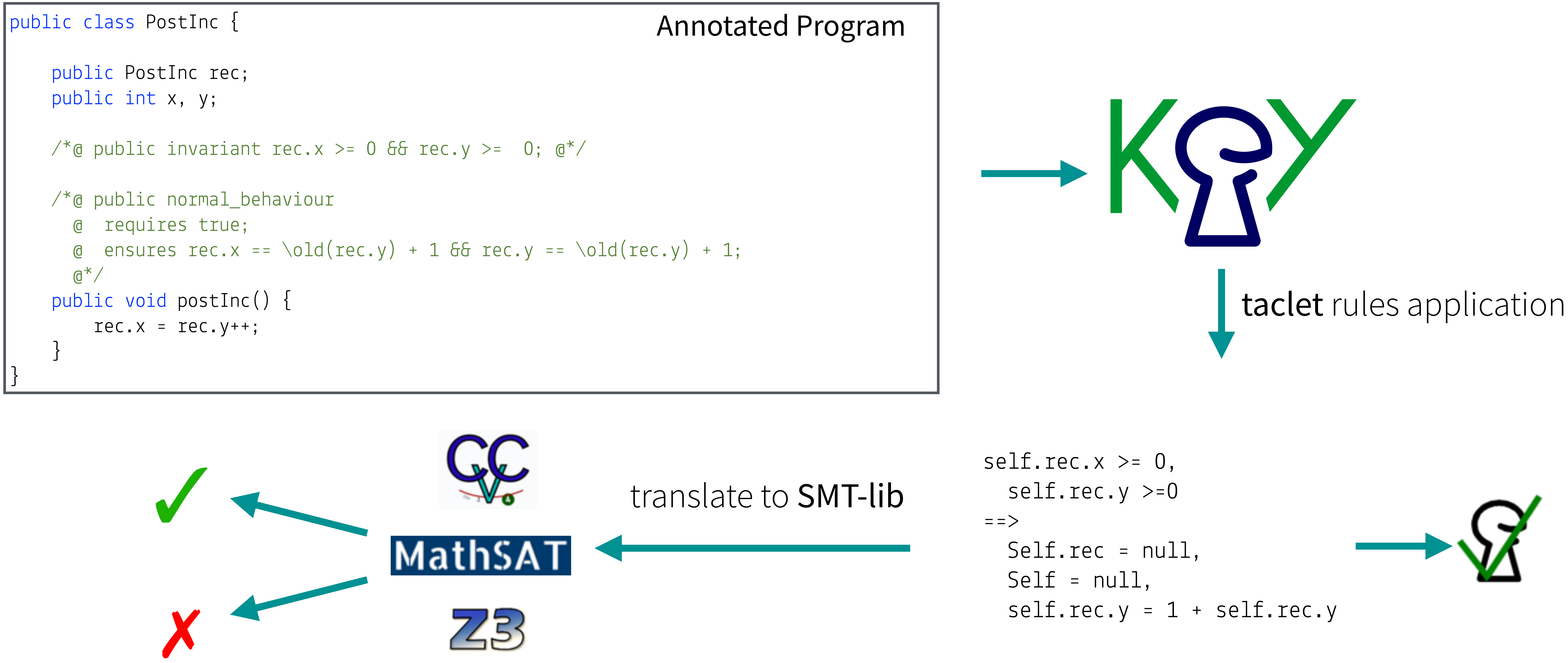
in Java programs

```
public class Circuit {

    double maxVoltage;
    double frequency;
    double resistance;
    double inductance;

    public Complex computeImpedance() {
        return new Complex(resistance, 2.0 * Math.PI * frequency * inductance);
    }

    public Complex computeCurrent() {
        return new Complex(maxVoltage, 0.0).divide(computeImpedance());
    }

    public double computeInstantCurrent(double time) {
        Complex current = computeCurrent();
        double maxCurrent = Math.sqrt(current.getRealPart() * current.getRealPart() +
         current.getImaginaryPart() * current.getImaginaryPart());
        double theta = Math.atan(current.getImaginaryPart() / current.getRealPart());

        return maxCurrent * Math.cos((2.0 * Math.PI * frequency * time) + theta);
    }
}
```

Does the program divide by zero?

Does the program overflow?

KeY workflow



Annotated Program

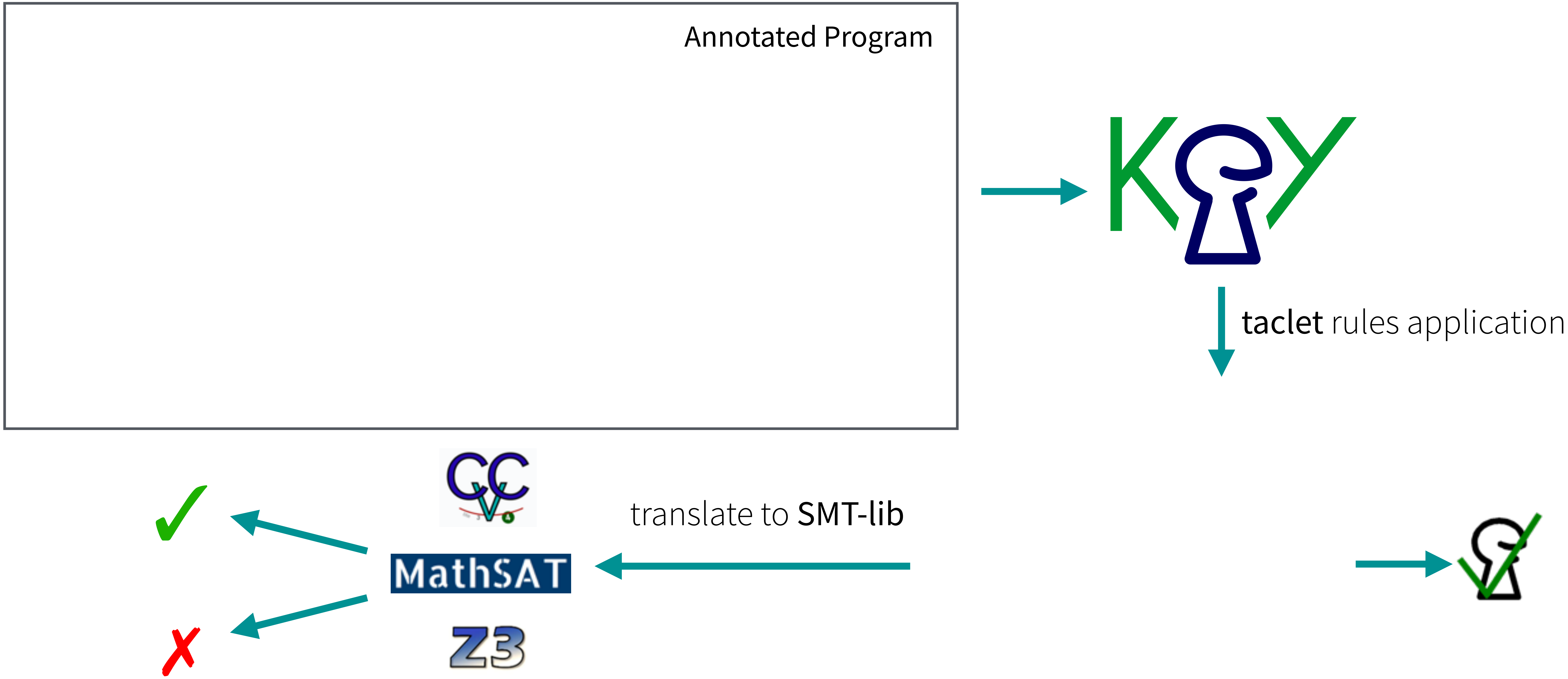
```
public class PostInc {

    public PostInc rec;
    public int x, y;

    /*@ public invariant rec.x >= 0 && rec.y >=  0; @*/

    /*@ public normal_behaviour
      @  requires true;
      @  ensures rec.x == \old(rec.y) + 1 && rec.y == \old(rec.y) + 1;
      @*/
    public void postInc() {
        rec.x = rec.y++;
    }
}
```

taclet rules application

```
self.rec.x >= 0,
  self.rec.y >=0
==>
  Self.rec = null,
  Self = null,
  self.rec.y = 1 + self.rec.y
```

translate to **SMT-lib**

Basic extension for floating-points

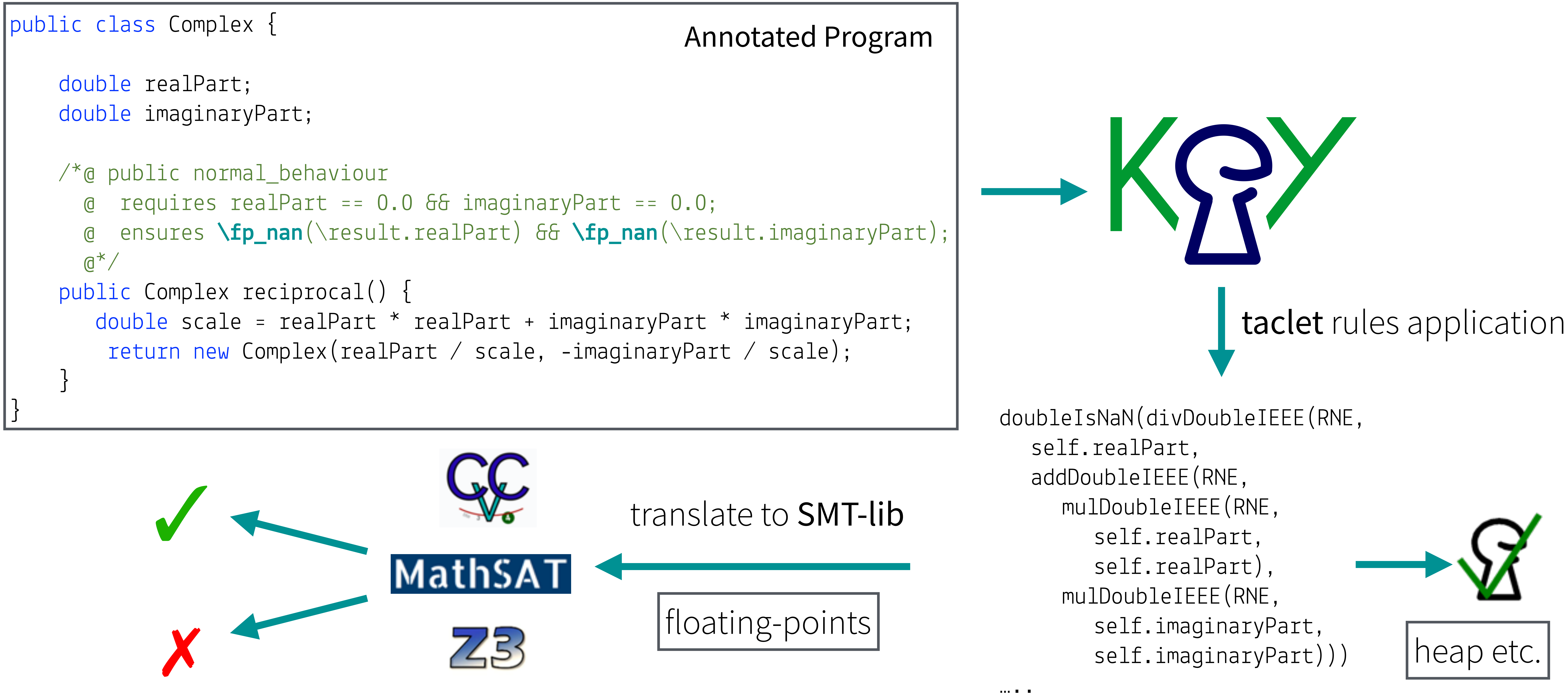


Annotated Program

taclet rules application

translate to **SMT-lib**

Basic extension for floating-points



Annotated Program

```
public class Complex {

    double realPart;
    double imaginaryPart;

    /*@ public normal_behaviour
      @  requires realPart == 0.0 && imaginaryPart == 0.0;
      @  ensures \fp_nan(\result.realPart) && \fp_nan(\result.imaginaryPart);
      @*/
    public Complex reciprocal() {
       double scale = realPart * realPart + imaginaryPart * imaginaryPart;
        return new Complex(realPart / scale, -imaginaryPart / scale);
    }
}
```

taclet rules application

```
doubleIsNaN(divDoubleIEEE(RNE,
   self.realPart,
   addDoubleIEEE(RNE,
      mulDoubleIEEE(RNE,
         self.realPart,
         self.realPart),
      mulDoubleIEEE(RNE,
         self.imaginaryPart,
         self.imaginaryPart)))
…..
```

heap etc.

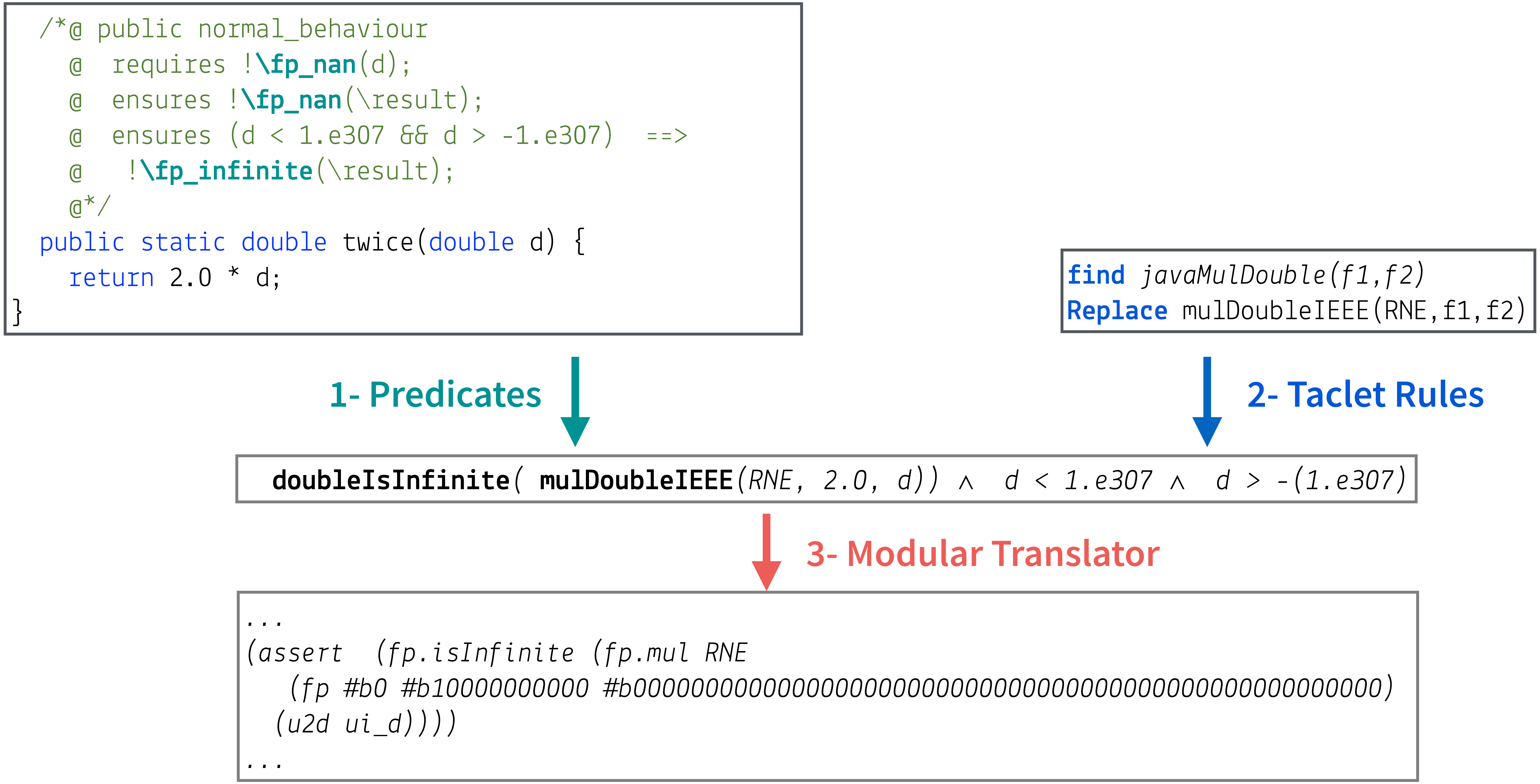
translate to **SMT-lib**

floating-points

✓

✗

Basic extension



```
  /*@ public normal_behaviour
    @  requires !\fp_nan(d);
    @  ensures !\fp_nan(\result);
    @  ensures (d < 1.e307 && d > -1.e307)  ==>
    @   !\fp_infinite(\result);
    @*/
  public static double twice(double d) {
    return 2.0 * d;
}
```

```
find javaMulDouble(f1,f2)
Replace mulDoubleIEEE(RNE,f1,f2)
```

1- Predicates

2- Taclet Rules

```
doubleIsInfinite( mulDoubleIEEE(RNE, 2.0, d)) ∧  d < 1.e307 ∧  d > -(1.e307)
```

3- Modular Translator

```
...
(assert  (fp.isInfinite (fp.mul RNE
   (fp #b0 #b10000000000 #b0000000000000000000000000000000000000000000000000000)
  (u2d ui_d))))
...
```

Library functions

no SMT-lib equivalent for transcendental functions:

- encode transcendental functions as **uninterpreted functions**
- **axiomatize** them

  in SMT queries, or

  in KeY as taclet rules

```
public class Circuit {

    double maxVoltage;
    double frequency;
    double resistance;
    double inductance;

    /*@ public normal_behaviour
      @ requires this.maxVoltage > 1.0  && this.maxVoltage < 12.0 &&
      @  this.frequency > 1.0 && this.frequency < 100.0 &&
      @  time > 0.0 && time < 300.0;
      @ ensures \fp_nice(\result);
      @*/
    public double computeInstantVoltage(double time) {
        return maxVoltage * Math.cos(2.0 * Math.PI * frequency * time);
    }
}
```

Axioms

- capture **high-level properties** of library functions
- comply with the specifications in the IEEE 754 standard
- e.g. encode **value ranges** and allow one to show that a function application is **not NaN**

```
Axiom: !fp_nan(a) ∧ !fp_infinite(a) → -1.0 ≤ cos(a) ≤ 1.0
```

```
/*@ public normal_behaviour
  @ requires this.maxVoltage > 1.0  && this.maxVoltage < 12.0 &&
  @  this.frequency > 1.0 && this.frequency < 100.0 &&
  @  time > 0.0 && time < 300.0;
  @ ensures \fp_nice(\result);
  @*/
public double computeInstantVoltage(double time) {
    return maxVoltage * Math.cos(2.0 * Math.PI * frequency * time);
}
```

Axiomatization

in SMT queries

function definitions and axioms are added to the SMT-LIB translation

axioms are expressed as **quantified** floating-point formulas

```
(assert (forall ((a Float64)) (=>
(and (not (fp.isNaN a)) (not (fp.isInfinite a)))
(and (fp.leq (cosDouble a)
(fp #b0 #b01111111111 #b0000...000000))
(fp.geq (cosDouble a)
(fp #b1 #b01111111111 #b0000...000000)) ))))
```

via taclet rules in KeY

axioms are encoded as taclets in KeY

fully automated, or user can choose which rule to apply

no quantified formulas

```
find cos(a)
add ¬fp_nan(a) ∧ ¬fp_infinite(a) → -1.0 ≤ cos(a) ≤ +1.0 ⟹
```

We can now prove

The **absence of special values** using **`fp_nan`**, **`fp_infinite`**, **`fp_nice`**

counter-example

```
/*@ public normal_behavior
  @  requires \fp_nice(arg0.x) && \fp_nice(arg0.y) && \fp_nice(arg1) && \fp_nice(arg2);
  @  ensures !\fp_nan(\result.x) && !\fp_nan(\result.y) && !\fp_nan(\result.width) && !\fp_nan(\result.height);
  @ also
  @ public normal_behavior
  @  requires -5.53 <= arg0.x && arg0.x <= -3.38 && -5.53 <= arg0.y && arg0.y <= -3.38 &&
  @    3.1 < arg0.width && arg0.width <= 3.7332 && 3.0000001 < arg0.height && arg0.height <=4.0004 &&
  @    3.0003001 < arg1 && arg1 <= 4.0024 && -6.4000003 < arg2 && arg2 <= 3.0001;
  @  ensures !\fp_nan(\result.x) && !\fp_nan(\result.y) && !\fp_nan(\result.width) &&!\fp_nan(\result.height);
  @*/
public Rectangle scale(Rectangle arg0, double arg1, double arg2){
    Area v1 = new Area(arg0);
    AffineTransform v2 = AffineTransform.getScaleInstance(arg1, arg2);
    Area v3 = v1.createTransformedArea(v2);
    Rectangle v4 = v3.getRectangle2D();
    return v4;
}
```

valid

We can now prove

The absence of special values with **transcendentals**

```
public class Circuit {
    double maxVoltage, frequency, resistance, inductance;
    // ...

    /*@ public normal_behavior
      @ requires 1.0<this.maxVoltage && this.maxVoltage<12.0 && 1.0<this.frequency && this.frequency<100.0 &&
      @  1.0<this.resistance && this.resistance<50.0 && 0.001<this.inductance && this.inductance<0.004 &&
      @  0.0 < time && time < 300.0;
      @ ensures !\fp_nan(\result) && !\fp_infinite(\result);
      @*/
    public double instantCurrent(double time) {
        Complex curr = computeCurrent();
        double maxCurrent = Math.sqrt(curr.getRealPart() * curr.getRealPart() +
            curr.getImaginaryPart() * curr.getImaginaryPart());
        double theta = Math.atan(curr.getImaginaryPart() / curr.getRealPart());
        return maxCurrent * Math.cos((2.0 * Math.PI * frequency * time) + theta);
    }
}
```

need to use `fp.sqrt`

axioms as taclet rules

We can now prove

Functional properties that are expressible in floating-point arithmetic

precomputed cosine

```
public class Rotation {
    final static double cos90 = 6.123233995736766E-17;
    final static double sin90 = 1.0;

    public static double[] rotate(double[] vec) {  // rotates a 2D vector by 90 degrees
        double x = vec[0] * cos90 - vec[1] * sin90;
        double y = vec[0] * sin90 + vec[1] * cos90;
        return new double[]{x, y};
    }

    /*@ public normal_behaviour
      @  requires (\forall int i; 0 <= i && i < vec.length;
      @   vec[i] > 1.0 && vec[i] < 2.0) && vec.length == 2;
      @  ensures  \result[0] < 1.0E-15 && \result[1] < 1.0E-15;
      @*/
    public static double[] computeError(double[] vec) {
        double[] temp = rotate(rotate(rotate(rotate(vec))));
        return new double[] { Math.abs(temp[0] - vec[0]), Math.abs(temp[1] - vec[1])};
    }
}
```

We can now prove

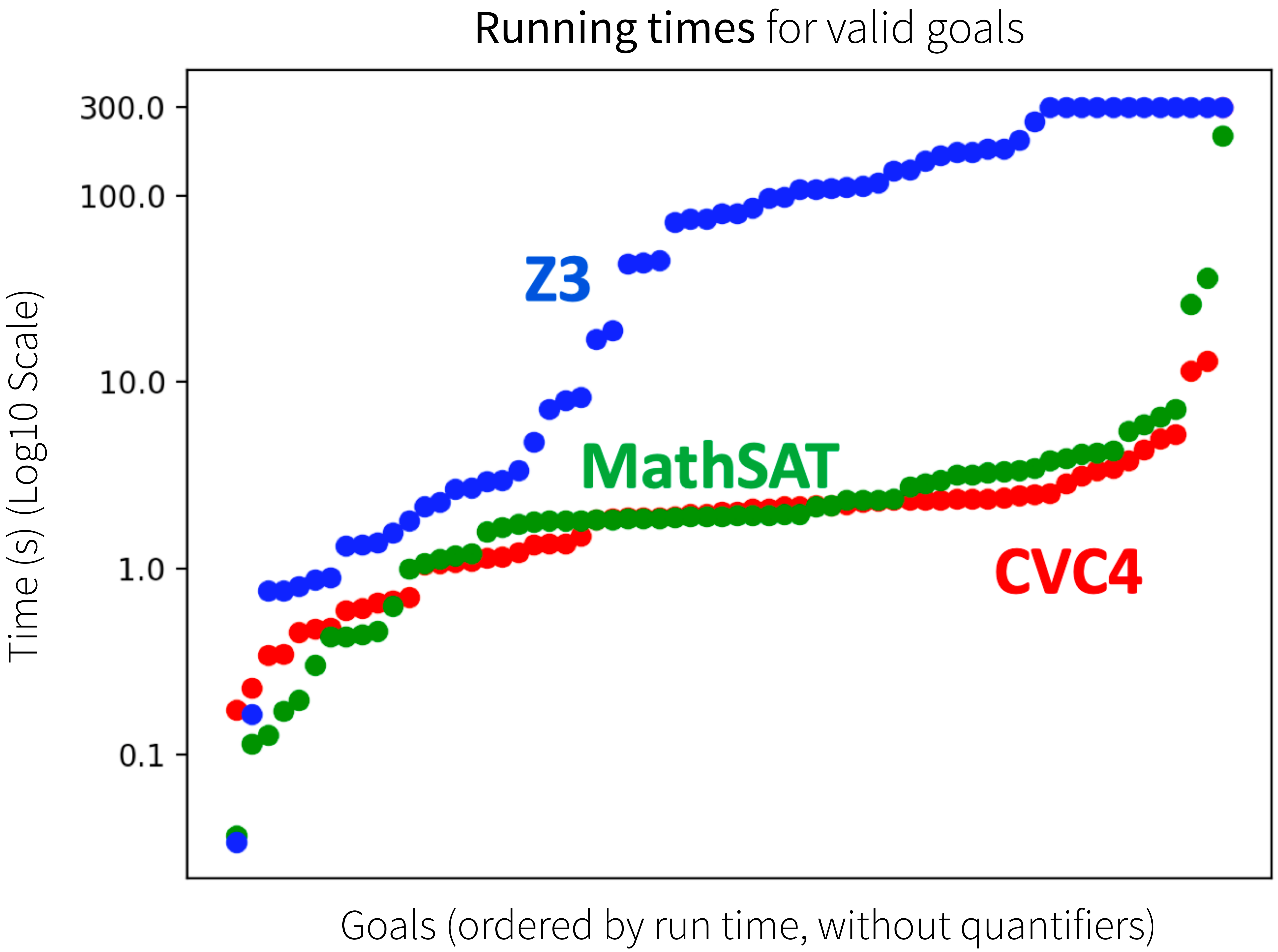
Loop invariants

- invariant generated by external tool [1]
- validated by KeY

```
/*@ public normal_behavior
  @ requires 0.0f <= u && u <= 0.0f && 2.0f <= v && v <= 3.0f;
  @ diverges true;
  @*/
public float pendulum-approx(float u, float v) {

    /*@ loop_invariant -1.1f <= u && u <= 1.2f  &&
      @  -3.2f <= v && v <= 3.1f &&
      @  (-0.11f*u) + (0.01f*v) + (1.0f*u*u) + (0.03f*u*v)
      @  + (0.12f*v*v) <= 1.15f;
      @*/
    while (true) {
        u = u + 0.01f * v;
        v = v + 0.01f * (-0.5f * v - 9.81f *
            (u - (u * u * u) / 6.0f +
            (u * u * u * u * u) / 120.0f));
    }
    return u;
}
```

[1] *Counterexample- and Simulation-Guided Floating-Point Loop Invariant Synthesis*. A. Izycheva, E. Darulova and Helmut Seidl. SAS'20

Solver performance



- Best **running time**: **CVC4**
- Most **goals** validated: **MathSAT**

Floating-point solvers have improved!

Axiomatization performance

Experiment	Quantified Axioms	# Goals	CVC4		Z3		MathSAT	
			# Goals Decided	Avg.	# Goals Decided	Avg.	# Goals Decided	Avg.
Axioms in SMT	✓	10	9	33.2	4	63.4	-	-
Axioms as Taclets	✗	10	10	33.4	5	74.2	8	0.9

- **axiomatization in KeY avoids quantified formulas:** both CVC4 and Z3 prove more goals
- `fp.sqrt` vs axiomatization:

  axiomatization mostly cheaper, but weaker

KeY float

This talk

Background on floating-point arithmetic *(real quick)*

Floating-points in KeY

Deductive Verification of Floating-Point Java Programs in KeY, TACAS’21 and STTT’23

Tutorial on rounding error analysis [1] *(by example)*

Recent work in rounding error analysis

Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs, SAS’23

---

[1] Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. A. Solovyev, C. Jacobsen, Z. Rakamaric, G. Gopalakrishnan. FM’15

Bounding rounding errors

real-valued specification: $f(y,z) = y^2 + z^2 \quad \text{where} \quad y \in [10.0, 20.0], z \in [20.0, 80.0]$

floating-point implementation: $\tilde{f}(\tilde{y},\tilde{z}) = \tilde{y} \,\tilde{*}\, \tilde{y} + \tilde{z} \,\tilde{*}\, \tilde{z} \quad \text{where} \quad \tilde{y} = y + u_y,\ \tilde{z} = z + u_z$

Goal: compute absolute rounding error bound:

$$\max_{y,z \in Y,Z} \left| f(y,z) - \tilde{f}(\tilde{y},\tilde{z}) \right|$$

- main challenge: **accurate** bounds
- over-approximation of the true errors; impossible to get exact errors in general
- (too) complex to reason about: combines real-valued and floating-point reasoning

  cannot be simply phrased as SMT-query
- (aside: easier than relative errors)

Abstracting floating-point arithmetic

$$f(y, z) = y^2 + z^2$$

too complex: $\max\limits_{y,z \in Y,Z} \left| f(y, z) - \tilde{f}(\tilde{y}, \tilde{z}) \right|$

use abstraction of floating-point arithmetic $\tilde{op} = op(1 + e) + d \quad \text{where } |e| \leq \epsilon, |d| \leq \delta$

to compute abstraction

$$\hat{f}(y, z, \mathbf{e}, \mathbf{d}) = \Big(((y(1{+}e_1){+}d_1)^2(1{+}e_2){+}d_2){+}((z(1{+}e_3){+}d_3)^2(1{+}e_4){+}d_4) \Big)(1{+}e_5)$$

- now only real-valued
- but still too complex to reason about automatically
- apply Taylor approximation; standard approach in maths and physics to simplify equations

Taylor approximation

general first-order Taylor approximation:

$$f(\mathbf{x}) = f(\underset{\uparrow}{\mathbf{a}}) + \sum_{i=1}^{k} \frac{\partial f}{\partial x_i}(\mathbf{a})(x_i - a_i) + 1/2 \sum_{i,j=1}^{k} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p})(x_i - a_i)(x_j - a_j)$$

choose suitable point

first-order derivative

remainder bounds error

compute Taylor approximation around $(y,\ z,\ \mathbf{0},\ \mathbf{0})$

$$\begin{aligned} \hat{f}(y, z, \mathbf{e}, \mathbf{d}) = \underset{\downarrow}{\hat{f}(y, z, \mathbf{0}, \mathbf{0})} + &\left.\frac{\partial \hat{f}}{\partial e_1}\right|_{y,z,\mathbf{0}} e_1 + \left.\frac{\partial \hat{f}}{\partial e_2}\right|_{y,z,\mathbf{0}} e_2 + \left.\frac{\partial \hat{f}}{\partial e_3}\right|_{y,z,\mathbf{0}} e_3 + \\ = f(y, z) \quad &\left.\frac{\partial \hat{f}}{\partial d_1}\right|_{y,z,\mathbf{0}} d_1 + \left.\frac{\partial \hat{f}}{\partial d_2}\right|_{y,z,\mathbf{0}} d_2 + R_2(y, z, \mathbf{e}, \mathbf{d}) \end{aligned}$$

approximate bound on roundoff error:

$$\max_{y,z \in I} \left| \hat{f}(y, z, \mathbf{e}, \mathbf{d}) - f(y, z) \right| = \max_{y,z \in I} \left| \left.\frac{\partial \hat{f}}{\partial e_1}\right|_{y,z,\mathbf{0}} e_1 + \ldots + \left.\frac{\partial \hat{f}}{\partial d_2}\right|_{y,z,\mathbf{0}} d_2 + R_2(y, z, \mathbf{e}, \mathbf{d}) \right|$$

Bounding rounding errors

$$\max_{y,z\in I}\left|\hat{f}(y,z,\mathbf{e},\mathbf{d}) - f(y,z)\right| = \max_{y,z\in I}\left| \left.\frac{\partial \hat{f}}{\partial e_1}\right|_{y,z,\mathbf{0}} e_1 + \ldots + \left.\frac{\partial \hat{f}}{\partial d_2}\right|_{y,z,\mathbf{0}} d_2 + R_2(y,z,\mathbf{e},\mathbf{d})\right|$$

- compute floating-point abstraction
- compute derivatives symbolically

} abstraction/simplification

- bound derivates over interval input domain

  with interval arithmetic or branch-and-bound

} solving/optimization

- supports arithmetic and transcendental functions (as library functions, but derivatives are well-defined)
- does **not** support function calls **modularly**

  requires *inlining* of functions

This talk

Background on floating-point arithmetic *(real quick)*

Floating-points in KeY

Deductive Verification of Floating-Point Java Programs in KeY, TACAS’21 and STTT’23

Tutorial on rounding error analysis *(by example)*

Recent work in rounding error analysis

Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs, SAS’23

joint work with Rosa Abbasi

Modular rounding error analysis

$$g(x) = x^2 \quad \text{where} \quad x \in [0.0, 100.0]$$

$$f(y, z) = g(y) + g(z) \quad \text{where} \quad y \in [10.0, 20.0], z \in [20.0, 80.0] \quad \longrightarrow \quad y, z \in [0.0, 100.0]$$

- consider simplified case first: no input errors
- **step 1:** compute an **error specification** for each procedure

$$\max_{x \in I} |g(x) - \tilde{g}(x)| \leq |\beta_g|$$

$$\max_{y,z \in J,K} |f(y, z) - \tilde{f}(y, z)| \leq |\beta_f|$$

reuse error spec

- **step 2:** instantiate error specifications for each procedure at their call-sites **with appropriate contexts**
- **goal:** abstract **enough** but not too much

constant error bound $\overset{?}{\longleftrightarrow}$ pre-compute only derivatives

looses too much **accuracy** — modest **performance**

Step 1: Roundoff error specification

$$g(x) = x^2 \quad \text{where} \quad x \in [0.0, 100.0]$$

$$f(y, z) = g(y) + g(z) \quad \text{where} \quad y \in [10.0, 20.0], z \in [20.0, 80.0]$$

- **extend rounding error model** with procedures: replaced with (real-valued) symbolic variables

$$\hat{g}(x, e_1, d_1) = x^2(1 + e_1) + d_1$$

$$\hat{f}(y, z, e_2, \beta_g(y), \beta_g(z)) = \Big(g(y) + \beta_g(y) + g(z) + \beta_g(z)\Big)(1 + e_2)$$

real-valued result

absolute error

depends on input parameter

- **values** of symbolic variables only needed at instantiation time

Step 1: Roundoff error abstraction

$$\hat{g}(x, e_1, d_1) = x^2(1 + e_1) + d_1$$

$$\hat{f}(y, z, e_2, \beta_g(y), \beta_g(z)) = \Big(g(y) + \beta_g(y) + g(z) + \beta_g(z)\Big)(1 + e_2)$$

- proceed as before with Taylor approximation:

$$\beta_g = \left.\frac{\partial \hat{g}}{\partial e_1}\right|_{x,\mathbf{0}} e_1 + \left.\frac{\partial \hat{g}}{\partial d_1}\right|_{x,\mathbf{0}} d_1$$

$$\beta_f = \left.\frac{\partial \hat{f}}{\partial e_2}\right|_{y,z,\mathbf{0}} e_2 + \left.\frac{\partial \hat{f}}{\partial \beta_g(y)}\right|_{y,z,\mathbf{0}} \beta_g(y) + \left.\frac{\partial \hat{f}}{\partial \beta_g(z)}\right|_{y,z,\mathbf{0}} \beta_g(z) + R_2(y, z, e_2, \beta_g(y), \beta_g(z))$$

treated symbolically

- pre-evaluate part of the Taylor approximations at abstraction stage already

Step 2: Instantiation

- **instantiate** error terms using interval analysis recursively

$$\begin{aligned}\beta_g &= \epsilon \max|x^2| + \delta,\\ \beta_f &= \epsilon \max|g(y) + g(z)| + (1 + 2\epsilon)\max|\beta_g(y) + \beta_g(z)|\end{aligned}$$

using intervals of y and z

- also check that intervals of error specifications are respected
- tradeoff: parts of β_g have been computed with (potentially) **wider ranges**, but **only once**
- **correctness:** inlining error specs without pre-computation yields the same error expression

Input errors

- use triangle inequality to split error:

$$|f(\mathbf{x}) - \tilde{f}(\tilde{\mathbf{x}})| = |f(\mathbf{x}) - f(\tilde{\mathbf{x}}) + f(\tilde{\mathbf{x}}) - \tilde{f}(\tilde{\mathbf{x}})| \leq \underbrace{|f(\mathbf{x}) - f(\tilde{\mathbf{x}})|}_{\text{propagation error}} + \underbrace{|f(\tilde{\mathbf{x}}) - \tilde{f}(\tilde{\mathbf{x}})|}_{\text{round-off error}}$$

- compute error specification in two parts:

$$\max_{x \in I} |g(x) - \tilde{g}(\tilde{x})| \leq |\gamma_g| + |\beta_g|$$

$$\max_{y,z \in J,K} |f(y,z) - \tilde{f}(\tilde{y},\tilde{z})| \leq |\gamma_f| + |\beta_f|$$

$$\gamma_g = g(\tilde{x}) - g(x) \quad \text{where} \quad \tilde{x} = x + u_x$$

$$\gamma_f = f(\tilde{y},\tilde{z}) - f(y,z) \quad \text{where} \quad \tilde{y} = y + u_y, \tilde{z} = z + u_z$$

- compute Taylor approximation, but w.r.p. inputs

Evaluation

benchmark	# top level	# procedure calls	# arithmetic ops	# arith. ops inlined
`matrix`	5	15	26	371
`matrixXL`	6	33	44	911
`matrixXS`	4	6	17	101
`complex`	15	152	98	699
`complexXL`	16	181	127	1107
`complexXS`	13	136	72	464

- `matrix`: library procedures on 3×3 matrices with determinant and Cramer’s rule
- `complex`: library procedures on complex numbers, used for computing properties of RL circuits
- `XL/XS`: larger or smaller versions

Performance-Accuracy wrt. state-of-the-art

case study	procedure	HUGO		Daisy		FPTaylor	
		err	time(s)	error	time(s)	err	time(s)
matrix	solveEquationX	4.14e-15	**3.9**	1.07e-15	10.5	**3.83e-16**	539.7
	solveEquationY	4.68e-15		1.55e-15		**6.11e-16**	
	solveEquationZ	5.16e-15		1.90e-15		**4.96e-16**	
	solveEquationsVector	4.73e-15		2.09e-16		**1.83e-16**	
matrixXL	solveEquationsVectorXL	4.78e-15	**5.9**	2.53e-16	24.2	**2.27e-16**	1342.0
matrixXS			**3.5**		4.0		158.9
complex	computeCurrentRe	6.12e-10	**239.7**	4.90e-10	439.1	**9.65e-14**	TO
	computeCurrentIm	6.71e-10		2.46e-11		**2.42e-13**	
	computeInstantCurrent	**3.34e-03**		5.57e+01		-	
	approxEnergy	**1.00e-01**		1.67e+03		-	
	computeRadiusVector	1.47e-11		**6.20e-14**		7.26e-14	
	computeDivideVector	2.39e-10		8.26e-14		**3.85e-14**	
	computeReciprocalRadiusV.	3.12e-14		3.89e-14		**4.67e-15**	
complexXL	approxEnergyXL	**2.00e-01**	**969.3**	3.34e+03	1315.1	-	TO
complexXS			181.7		**13.4**		140.7

Are we there yet?

Rounding error analysis

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor [1], Daisy [2], PRECiSA [3]

[1] Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. A. Solovyev, C. Jacobsen, Z. Rakamaric, G. Gopalakrishnan. FM'15

[2] Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper), TACAS'18

[3] An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, L. Titolo, M.A. Feliú, M.M. Moscato, C.A. Muñoz. VMCAI'18

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor, Daisy, PRECiSA
- ☐ conditionals [1, 2]

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0)
    ...
  else
    ...
}
```

real-valued and floating-point executions
may/will diverge

---

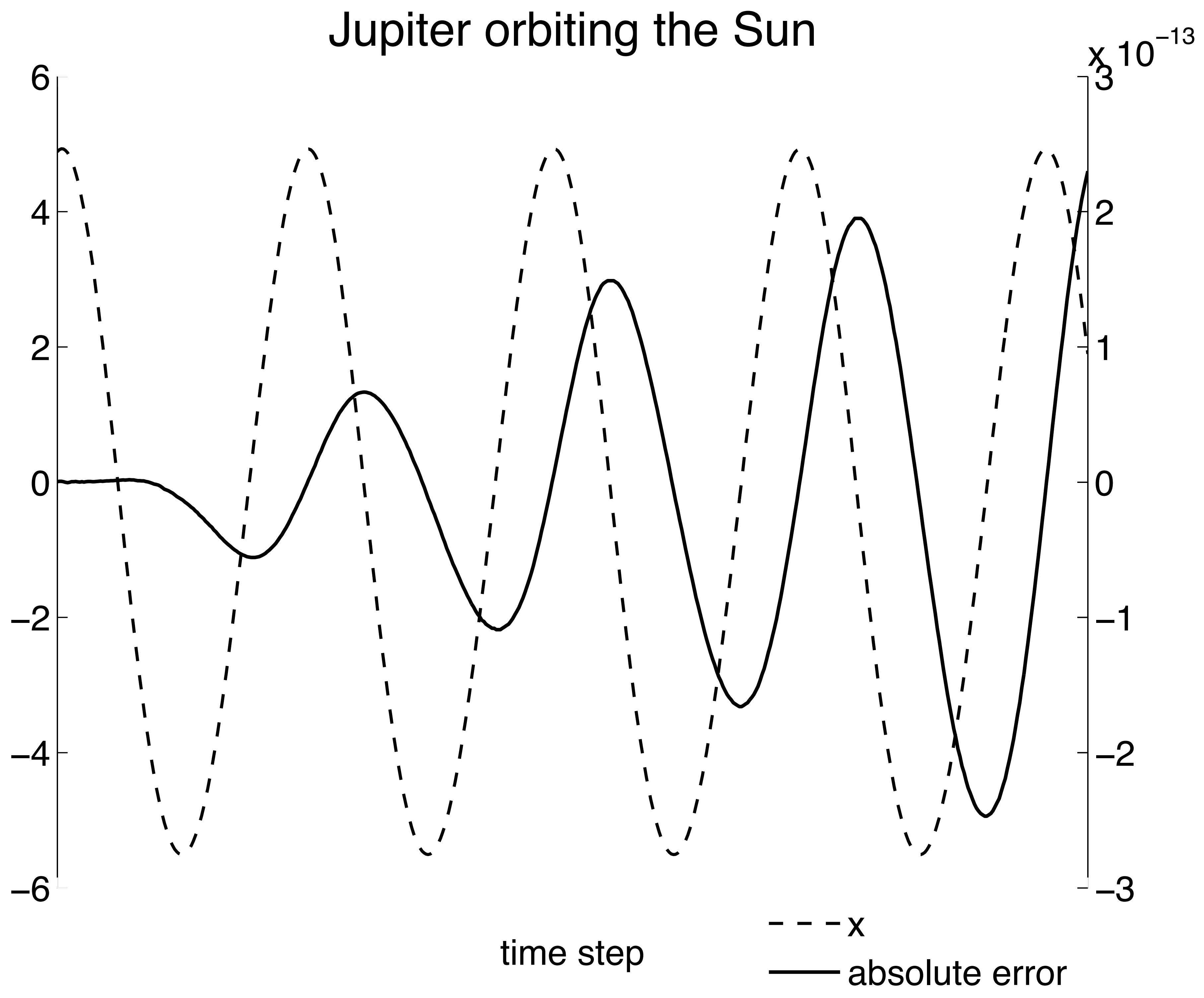
[1] Towards a Compiler for Reals. E. Darulova, V. Kuncak, TOPLAS'17

[2] An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, L. Titolo, M.A. Feliú, M.M. Moscato, C.A. Muñoz. VMCAI'18

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor, Daisy, PRECiSA
- ☐ conditionals
- ☐ loops [1, 2, 3]



[1] Towards a Compiler for Reals. E. Darulova, V. Kuncak, TOPLAS’17

[2] An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, L. Titolo, M.A. Feliú, M.M. Moscato, C.A. Muñoz. VMCAI’18

[3] Scaling up Roundoff Analysis of Functional Data Structure Programs. A. Isychev, E. Darulova. SAS’23

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor, Daisy, PRECiSA
- ☐ conditionals
- ☐ loops
- ☐ scalability [1, 2, 3]

imperative code:

```
double heat1d(double (*xm)[N], double (*xp)[N], double* x0) {
  int i,j;
  for (j=1; j<N; j++) {
    for (i=2; i<(N-j); i++) {
      xm[j][i]=0.25*xm[j-1][i+1]+0.5*xm[j-1][i]+0.25*xm[j-1][i-1];
      xp[j][i]=0.25*xp[j-1][i-1]+0.5*xp[j-1][i]+0.25*xp[j-1][i+1];
    }
    xm[j][0] = 0.25*xm[j-1][1] + 0.5*xm[j-1][0] + 0.25*x0[j-1];
    xp[j][0] = 0.25*xp[j-1][1] + 0.5*xp[j-1][0] + 0.25*x0[j-1];
    x0[j] = 0.25*xm[0][j-1] + 0.5*x0[j-1] + 0.25*xp[0][j-1];
  }
  return x0[N-1];
}
```

[1]Scalable yet Rigorous Floating-point Error Analysis. A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, P. Panchekha SC'20

[2] A Two-Phase Approach for Conditional Floating-Point Verification. D. Lohar, C. Jeangoudoux, J. Sobel, E. Darulova, M. Christakis. TACAS'21

[3] Scaling up Roundoff Analysis of Functional Data Structure Programs. A. Isychev, E. Darulova. SAS'23

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor, Daisy, PRECiSA
- ☐ conditionals
- ☐ loops
- ☐ scalability [1, 2, 3]

analysis of our functional DSL scales better [3]:

```
def heat1d(ax: Vector): Real = {
  require(1.0 <= ax && ax <= 2.0 && ax.size(33))

  if (ax.length() <= 1) {
    ax.head
  } else {
    val coef = Vector(List(0.25, 0.5, 0.25))
    val updCoefs = ax.slideReduce(3,1)(v => (coef*v).sum())
    heat1d(updCoefs)
  }
}
```

[1]Scalable yet Rigorous Floating-point Error Analysis. A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, P. Panchekha SC'20

[2] A Two-Phase Approach for Conditional Floating-Point Verification. D. Lohar, C. Jeangoudoux, J. Sobel, E. Darulova, M. Christakis. TACAS'21

[3] Scaling up Roundoff Analysis of Functional Data Structure Programs. A. Isychev, E. Darulova. SAS'23

Are we there yet?

Rounding error analysis

- ☑ for (short) pure arithmetic computations: FPTaylor, Daisy, PRECiSA
- ☐ conditionals
- ☐ loops
- ☐ scalability

Deductive verification (in KeY)

- ☑ with automation: absence of runtime errors
- ☐ ???