A Compiler for Sound Floating-Point using Affine Arithmetic

Joao Rivera, ETH Zürich
Franz Franchetti, Carnegie Mellon University
Markus Püschel, ETH Zürich
double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n; i++) {
        xi = x;
        yi = y;
        x = 1.0 - a*xi*xi + yi;
        y = b*xi;
    }
    return x;
}
double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n; i++) {
        xi = x;
        yi = y;
        x = 1.0 - a*xi*xi + yi;
        y = b*xi;
    }
    return x;
}"
double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n ; i++) {
        xi = x;
        yi = y;
        x = 1.0 - a*xi*xi + yi ;
        y = b*xi ;
    }
    return x;
}
double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n; i++) {
        xi = x;
        yi = y;
        x = 1.0 - a*xi*xi + yi;
        y = b*xi;
    }
    return x;
}

How accurate is the result?
We don’t know...

FP is not sound with respect to real arithmetic.

**Sound Floating-Point**
Accuracy guarantees of the final results.
How to make the code sound?

Ideal approach:
Static round-off error analysis.

double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n; i++) {
        xi = x;  yi = y;
        x = 1.0 - a*xi*xi + yi;
        y = b*xi;
    }
    return x;
}
How to make the code sound?

**Ideal approach:**

Certified Roundoff Error Bounds Using Semidefinite Programming

VICTOR MAGRON, CNRS Verimag
GEORGE CONSTANTINIDES and ALASTAIR DONALDSON, Imperial College London

Roundoff errors cannot be avoided when implementing numerical programs with finite precision. The ability to reason about rounding is especially important if one wants to explore a range of potential representations for instances, for FPGAs or custom hardware implementations. This problem becomes challenging when the program does not employ solely linear operations as non-linearities are inherent in many interesting

Towards a Compiler for Reals

EVA DARULOVA, Max Planck Institute for Software Systems
VIKTOR KUNCAK, Ecole Polytechnique Federale de Lausanne

Numerical software, common in scientific computing or precision approximation of the real arithmetic in which most algorithms, the roundoff errors introduced by finite-precision arithmetic and measurement and other input errors further increase the uncore tools are needed to help users select suitable data types and evaul safety-critical applications.

We present a source-to-source compiler called Rosa which tal error specifications and synthesizes code over an appropriate fio main challenge of such a compiler is a fully automated, sound a estimation. We introduce a unified technique for bounding round point arithmetic of various precisions. The technique can handle no symbolic invariants for unbounded loops and quantify the effects c evaluate Rosa on a number of benchmarks from scientific comput: it to state-of-the-art in automated error estimation, show that it accuracy and performance.

Static Analysis of Finite Precision Computations

Eric Goubault and Sylvie Putot
CEA LIST, Laboratory for the Modelling and Analysis of Interacting Systems, Point quant 94, Gif-sur-Yvette, F-91191 France, firstname.lastname@cea.fr

Abstract. We define several abstract semantics for the static analysis of finite precision computations, that bound not only the ranges of values taken by numerical variables of a program, but also the difference with the result of the same sequence of operations in an idealized real number semantics. These domains point out with more or less detail (control point, block, function for instance) sources of numerical errors in the pro- gram and the way they were propagated by further computations, thus sensitivity to uses of abstrac
tly relational

Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions

ALEXEY SOLOYEV, MAREK S. BARANOWSKI, IAN BRIGGS, CHARLES JACOBSEN, ZVONIMIR RAKAMARIC, and GANESH GOPALAKRISHNAN, School of Computing, University of Utah, Salt Lake City, UT, USA

Rigorous estimation of maximum floating-point round-off errors is an important capability central to many formal verification tools. Unfortunately, available techniques for this task often provide very pessimistic over-estimates, causing unnecessary verification failure. We have developed a new approach called Symbolic Taylor Expansions that avoids these problems, and implemented a new tool called FPTaylor embodying this approach. Key to our approach is the use of rigorous global optimization, instead of the more familiar interval arithmetic, affine arithmetic, and/or SMT solvers. FPTaylor emits per-instance analysis certificates in the form of HOL. Light proofs that can be machine checked.

In this article, we present the basic ideas behind Symbolic Taylor Expansions in detail. We also survey as well as thoroughly evaluate six tool families, namely, Gappa (two tool options studied), Floatmat, PRECiSA, Real2Float, Rosa, and FPTaylor (two tool options studied) on 24 examples, running on the same machine, and
How to make the code sound?

Ideal approach:
Static round-off error analysis.

```
double henon_map(double x, double y, int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n; i++) {
        xi = x;  yi = y;
        x = 1.0 - a*xi*xi + yi ;
        y = b*xi ;
    }
    return x;
}
```

**Limitations**
- Only for very simple programs.
- Loops are problematic.
- Overestimation.
How to make the code sound?

Analysis at runtime:
Rewrite code to account for ranges, e.g., using interval arithmetic (IA).

```c
struct { double lo; double up; } ival_t ;

ival_t henon_map_ia(ival_t x, ival_t y, int n) {
    ival_t xi, yi;
    ival_t a = {1.05, 1.05};
    ival_t b = {0.3, 0.3};
    for (int i = 0; i < n ; i ++) {
        xi = x;  yi = y;
        ival_t t1 = ia_mul(xi, xi);
        ival_t t2 = ia_mul(a, t1);
        ...
    }
    return x;
}
```

5x slower
How to make the code sound?

Analysis at runtime:
Rewrite code to account for ranges, e.g., using *interval arithmetic* (IA).

```c
struct { double lo; double up; } ival_t;

ival_t henon_map_ia(ival_t x, ival_t y, int n) {
    ival_t xi, yi;
    ival_t a = {1.05, 1.05};
    ival_t b = {0.3, 0.3};
    for (int i = 0; i < n; i++) {
        xi = x; yi = y;
        ival_t t1 = ia_mul(xi, xi);
        ival_t t2 = ia_mul(a, t1);
        ...
    }
    return x;
}
```

The result is *sound* but it’s *not* accurate.

Overapproximation due to the dependency problem.

5x slower
Dependency problem: IA vs AA

*Example:*

*Interval arithmetic (IA)*

\[
\bar{a} = [0, 1] \\
\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1]
\]
Dependency problem: IA vs AA

*Example:*

*Interval arithmetic (IA)*

\[ \overline{a} = [0, 1] \]

\[ \overline{b} = \overline{a} - \overline{a} = [0, 1] - [0, 1] = [-1, 1] \]

**Correlation is lost**
Dependency problem: IA vs AA

**Example:**

*Interval arithmetic (IA)*

\[
\bar{a} = [0, 1]
\]

\[
\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1] = [-1, 1]
\]

Correlation is lost

*Affine arithmetic (AA)*

\[
\hat{a} = 0.5 + 0.5\epsilon_1, \quad \text{where} \quad \epsilon_i = [-1, 1]
\]
Dependency problem: IA vs AA

Example:

Interval arithmetic (IA)

\[ \bar{a} = [0, 1] \]
\[ \bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1] = [-1,1] \]

Correlation is lost

Affine arithmetic (AA)

\[ \hat{a} = 0.5 + 0.5\varepsilon_1, \quad \text{where} \quad \varepsilon_i = [-1, 1] \]
\[ \hat{b} = \hat{a} - \hat{a} = (0.5 + 0.5\varepsilon_1) - (0.5 + 0.5\varepsilon_1) \]
\[ \hat{b} = \hat{a} - \hat{a} = 0 \]

Correlation is kept using error symbols
Using affine arithmetic

Using AA library

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m; i ++) {
        xi = x;  yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```
Using affine arithmetic

Using AA library

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m; i++) {
        xi = x;  yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

Very accurate!...
Using affine arithmetic

Using AA library

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m ; i ++) {
        xi = x; yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

Very accurate!... but 10K slower
Using affine arithmetic

Using AA library

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m; i++) {
        xi = x;  yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

Very accurate!... but 10K slower

![Graph of Henon Map]

Every operation introduces a new error symbol.

After $m$ iterations:

$$x = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + x_3 \epsilon_3 + x_4 \epsilon_4 + \cdots + x_n \epsilon_n$$
Using affine arithmetic

Using AA library

```cpp
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m; i++) {
        xi = x;  yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

Very accurate!... but 10K slower

Every operation introduces a new error symbol.

After $m$ iterations:

\[ x = x_0 + x_1 \varepsilon_1 + x_2 \varepsilon_2 + x_3 \varepsilon_3 + x_4 \varepsilon_4 + \cdots + x_n \varepsilon_n \]
Using affine arithmetic

Using AA library

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m ; i ++) {
        xi = x;  yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

Very accurate!... but 10K slower

Every operation introduces a new error symbol.

After $m$ iterations:

\[ x = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + x_{n+1} \epsilon_{n+1} \]

Accuracy vs performance tradeoff
Challenges to achieve soundness

- Get the AA or IA code with little effort
- Decide what symbols to keep in AA
- How to achieve high performance
- What if input code has SIMD intrinsics

```c
affine_t henon_map_aa(affine_t x, affine_t y, int m) {
    affine_t xi, yi;
    affine_t a = aa_set(1.05, 0.);
    affine_t b = aa_set(0.3, 0.);
    for (int i = 0; i < m; i++) {
        xi = x; yi = y;
        affine_t t1 = aa_mul(xi, xi);
        affine_t t2 = aa_mul(a, t1);
        ...
    }
    return x;
}
```

\[
x = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + x_3 \epsilon_3 + \cdots + x_n \epsilon_n
\]
A compiler for sound floating-point computations

**Input:** Numerical C function (with SIMD)

```
double henon(double x, ...) {
  ...
  y = b * xi;
  ...
}
```

**Target precision**
Single, Double, Double-double

**Output:** Equivalent sound function using AA or IA (with SIMD)

```
f64a henon(f64a x, ...) {
  ...
  y = aa_mul_f64(b, xi);
  ...
}
```

**Analysis to decide on symbols to keep**
Decl (type a, foo, ..)
Stmt (for, if, ..)
Expr (a + b, sqrt, ..)

**Code transformation**

- Result is sound.
- Fast implementation.
- Optionally using SIMD.
A compiler for sound floating-point computations

**Input:** Numerical C function (with SIMD)

double henon(double x, ...) {  
  ...  
  y = b * xi;  
  ...  
}

**Target precision**  
Single, Double, Double-double

**Accuracy improvement**  
Analysis to decide on symbols to keep

**Code transformation**  
Decl (type a, foo, ..)  
Stmt (for, if, ..)  
Expr (a + b, sqrt, ..)

**Output:** Equivalent sound function using AA or IA (with SIMD)

f64a henon(f64a x, ...) {  
  ...  
  y = aa_mul_f64(b, xi);  
  ...  
}

- Result is sound.  
- Fast implementation.  
- Optionally using SIMD.

---

Transformations

Example

Original

double foo(double a, double b) {
    double c;
    c = a * b;
    c = c + 0.1;
    if (c > a) {
        ...
    }
    return c;
}

Generated

f64a foo(f64a a, f64a b) {
    f64a c;
    c = aa_mul_f64(a, b);
    f64a t1 = aa_set_f64(0.1, 1.38...e-17);
    c = aa_add_f64(c, t1);
    if (aa_cmpgt_f64(c, a)) {
        ...
    }
    return c;
}

Decl node
FP type ➔ Affine type

Expr node
Sound operations
Transformations

**Example**

<table>
<thead>
<tr>
<th>Original</th>
<th>Generated</th>
<th>Decl node</th>
<th>Expr node</th>
</tr>
</thead>
<tbody>
<tr>
<td>double foo(double a, double b) {</td>
<td>f64a foo(f64a a, f64a b) {</td>
<td>FP type</td>
<td>Sound operations</td>
</tr>
<tr>
<td>double c;</td>
<td>f64a c;</td>
<td>Affine</td>
<td>Sound constants</td>
</tr>
<tr>
<td>c = a * b;</td>
<td>c = aa_mul_f64(a, b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c = c + 0.1;</td>
<td>f64a t1 = aa_set_f64(0.1, 1.38...e-17);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (c &gt; a) {</td>
<td>c = aa_add_f64(c, t1);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>if (aa_cmpgt_f64(c, a)) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return c;</td>
<td>return c;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Limiting the number of symbols

Decide what symbols to keep to preserve accuracy
• at runtime based on a policy.
• at static time using DAG analysis.

\[ x = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + x_3 \epsilon_3 + \cdots + x_n \epsilon_n \]

\[ x = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + x_{n+1} \epsilon_{n+1} \]
Limiting the number of symbols

Decide what symbols to keep to preserve accuracy
- at runtime based on a policy.
- at static time using DAG analysis.

Decide at runtime using fusion policy:
- Random (RP)
- Oldest symbol (OP)
- Symbols with smallest absolute values (SP)
- Below mean (MP)
- ...

\[ x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 + \cdots + x_n\epsilon_n \]

Example with \( k = 3 \)

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4(\epsilon_1)</td>
<td>3(\epsilon_4)</td>
</tr>
<tr>
<td>1(\epsilon_3)</td>
<td>3(\epsilon_4)</td>
</tr>
<tr>
<td>3(\epsilon_4)</td>
<td>-2(\epsilon_5)</td>
</tr>
</tbody>
</table>
Preserving accuracy at static time: Using DAG analysis

\[ x \cdot z - y \cdot z \]

Example

\[ x = 1 + \epsilon_x \]
\[ y = 1 + \epsilon_y \]
\[ z = 1 + \epsilon_z \]
Preserving accuracy at static time: Using DAG analysis

\[ x \cdot z - y \cdot z \]

**Example**

\[ x = 1 + \epsilon_x \]
\[ y = 1 + \epsilon_y \]
\[ z = 1 + \epsilon_z \]

\[ t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1} \]
\[ t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2} \]
Preserving accuracy at static time: Using DAG analysis

\[x \cdot z - y \cdot z\]

**Example**

\[x = 1 + \epsilon_x\]
\[y = 1 + \epsilon_y\]
\[z = 1 + \epsilon_z\]
\[t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1}\]
\[t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2}\]
\[t_3 = t_1 - t_2 = \epsilon_x + \epsilon_{t1} - \epsilon_y - \epsilon_{t2}\]

\(\epsilon_z\) cancels out
Preserving accuracy at static time: Using DAG analysis

\[ x \cdot z - y \cdot z \]

**Example**

\[ x = 1 + \epsilon_x \]
\[ y = 1 + \epsilon_y \]
\[ z = 1 + \epsilon_z \]

\[ t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1} \]
\[ t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2} \]

\[ t_3 = t_1 - t_2 = \epsilon_x + \epsilon_{t1} - \epsilon_y - \epsilon_{t2} \]

\( \epsilon_z \) cancels out

**Reuse:**
When a symbol arrives to a node from two different paths. Symbols that are **reused** are likely to cancel out.

\( \epsilon_z \) should be kept to allow reuse.
Preserving accuracy at static time: Using DAG analysis

\[ x \cdot z - y \cdot z \]

**Example with \( k = 2 \)**

\[ x = 1 + \epsilon_x \]
\[ y = 1 + \epsilon_y \]
\[ z = 1 + \epsilon_z \]

\[ t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1} \]
\[ t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2} \]

Keep \( \epsilon_z \), i.e., fuse \( \epsilon_x \) and \( \epsilon_{t1} \)

\( \epsilon_z \) should be kept to allow reuse.
Preserving accuracy at static time: Using DAG analysis

\[ x \cdot z - y \cdot z \]

\( \varepsilon_x \), \( \varepsilon_z \), \( \varepsilon_y \)

\( \varepsilon_t1 \), \( \varepsilon_t2 \), \( \varepsilon_t3 \)

\( \varepsilon_z \) should be kept to allow reuse.

**Reuse**
When a symbol arrives to a node from two different paths.

**Problem**
Find the symbols to keep that maximizes reuse given that each node can keep at most \( k \) symbols.

**Modeled as an ILP program**

maximize \[ \sum_{s \in V} [\rho(s), \rho(s), \ldots, \rho(s)]q_s \]

subject to \[ \sum_{s \in V} p_s \leq [k - 1, k - 1, \ldots, k - 1]^T, \]
\[ p_s = R_s \circ q_s \quad \text{for all } s \in V, \]
Evaluation
**Setup**
Intel Xeon E-2176M @2.7GHz
Ubuntu 18.04, gcc 7.5

**f64a_dspv:**
smallest policy, with DAG analysis and vectorized.

**DAG analysis:**
4-8 bits improvement with 25% overhead.
Setup
Intel Xeon E-2176M @2.7GHz
Ubuntu 18.04, gcc 7.5

**Slowdown**

(b) *sor*

(a) *henon*
Comparison with IA and AA libraries

- 30x-70x faster than ceres-affine.
- 10x-36x slower than IGen (IA).

f64a_dspv:

30x-70x faster than ceres-affine.
10x-36x slower than IGen (IA).
Compiler for Sound Floating-Point using AA

file.c ─── DAG Analysis ─── Transform node
        Decl, Stmt, Expr

 safegen_file.c (optionally with SIMD)

f64a foo(f64a a, ...) {
  ...
  c = aa_add_f64(a, b);
  ...
}

Fusion Policies

DAG Analysis

Evaluation

30x-70x faster libraries for AA
More accurate than IA
Prioritization improves accuracy by 4-8 bits