Static Analysis of Numerical Algorithms

Eric Goubault and Sylvie Putot

CEA Saclay
MeASI

Taylor Methods Workshop, Boca-Raton, 12/2006
Goal of the talk: characterize the loss of precision in programs, due to floating-point arithmetic, at compile time
Goal of the talk: characterize the loss of precision in programs, due to floating-point arithmetic, at compile time

- A very brief introduction to static analysis by abstract interpretation
Goal of the talk: characterize the loss of precision in programs, due to floating-point arithmetic, at compile time

- A very brief introduction to static analysis by abstract interpretation
- Implicitly relational domain for real-number value analysis by abstract interpretation, relying on affine arithmetic
  - Join and meet operations, order
Goal of the talk: characterize the loss of precision in programs, due to floating-point arithmetic, at compile time

- A very brief introduction to static analysis by abstract interpretation
- Implicitely relational domain for real-number value analysis by abstract interpretation, relying on affine arithmetic
  - Join and meet operations, order
- Relational domain for values and errors, main ideas
Goal of the talk: characterize the loss of precision in programs, due to floating-point arithmetic, at compile time

- A very brief introduction to static analysis by abstract interpretation
- Implicitly relational domain for real-number value analysis by abstract interpretation, relying on affine arithmetic
  - Join and meet operations, order
- Relational domain for values and errors, main ideas
- Example based on an extract from instrumentation software
A program is considered as a dynamical system (discrete in general)

We can be interested in two main types of properties:

- **safety**, through invariant *true on all trajectories* - for all inputs or parameters. Application: give bounds for variables, prove absence of RTEs etc.
- **liveness** properties which become true at a certain time, on one or all of the trajectories. Application: reachability of a state, termination etc.

Similarity with certain concepts (and methods) of numerical mathematics and control theory.

Theory and tools for *automatic* analysis of such properties, given a program
But automatic (or algorithmic) means...

...undecidability (ex. Turing halting problem). So we use abstractions to find over-approximations of these sets of values (sometimes under-approximations too).

→ abstract interpretation
Example

```c
void main() { [0]
    int x=[-100,50]; [1]
    while [2] (x<100) { [3]
        x=x+1; [4]
    } [5]
}
```

\[
\begin{align*}
x_0 & = \top \\
x_1 & = [-100,50] \\
x_2 & = x_1 \cup x_4 \\
x_3 & = ]-\infty,99[ \cap x_2 \\
x_4 & = x_3 + [1,1] \\
x_5 & = [100, +\infty[ \cap x_2 
\end{align*}
\]
Resolution of semantic equations

- (Tarsky) \((\mathcal{P}(\mathbb{Z}), \subseteq)\) (similarly, intervals) is a complete lattice and the functional is monotonic \(\Rightarrow\) there is a least fixed point
Resolution of semantic equations

- (Tarsky) \( \mathcal{O}(\mathbb{Z}), \subseteq \) (similarly, intervals) is a complete lattice and the functional is monotonic \( \Rightarrow \) there is a least fixed point
- We compute the Kleene iteration (\( f \) is actually order-theoretically continuous here)

\[
\text{lfp}(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\perp)
\]

for the functional:

\[
F = \begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
\end{pmatrix} = \begin{pmatrix}
\top \\
[{-100, 50}] \\
x_1 \cup x_4 \\
] - \infty, 99] \cap x_2 \\
x_3 + [1, 1] \\
[100, +\infty[ \cap x_2 \\
\end{pmatrix}
\]
void main() { [0]  
int x=[-100,50]; [1]  
while [2] (x<100) { [3]  
x=x+1; [4]  
} [5]  
} 

x_0 = ⊤ 
x_1 = [-100, 50] 
x_2 = x_1 ∪ x_4 
x_3 = ] − ∞, 99] ∩ x_2 
x_4 = x_3 + [1, 1] 
x_5 = [100, +∞][∩x_2 

(choatic iteration here/Gauss-Seidel like)

\[
\begin{align*}
x_0^{100} &= \top \\
x_1^{100} &= [-100, 50] \\
x_2^{100} &= x_1 \cup x_4 \\
x_3^{100} &= ] - \infty, 99] \cap x_2 \\
x_4^{100} &= x_3 + [1, 1] \\
x_5^{100} &= [100, +\infty[ \cap x_2 
\end{align*}
\]

Of course this is naive: acceleration of convergence, relational domains etc.
Context of the present work

- Static analysis by abstract interpretation for inaccuracy errors in floating-point computations (FLUCTUAT tool)
  - Follows the floating-point control flow (given an evaluation order!)
  - Guaranteed bounds on errors between real number computation (what is expected) and the implementation in floating-point numbers
  - Identify operations responsible for the accuracy losses

- Applications
  - Safety-critical instrumentation software
  - Towards numerically more intensive programs

- Need for a very accurate real number value analysis
The set of floating-point values that a variable \( x \) can take is expressed as:

\[
f^x = r^x + e_1^x + e_{ho}^x \\
= r^x + \bigoplus_{i \in I} \alpha_i^x + e_{ho}^x
\]

where:
- \( r^x \) is the real-number value that should have been computed if we had exact arithmetic available
- the \( \alpha_i^x \) are coefficients expressing the propagation in \( x \) of the initial first-order error introduced by the arithmetic operation labelled \( i \) in the program
- \( e_{ho}^x \) is the higher-order error
float x = 0.1; // [1]
float y = 0.5; // [2]
float z = x+y; // [3]
float t = x*y; // [4]

x = 0.1 + 1.49011612e^{-9} [1]
y = 0.5
z = 0.6 + 1.49011612e^{-9} [1] +
    2.23517418e^{-8} [3]
t = 0.06 + 1.04308132e^{-9} [1] + 
    2.23517422e^{-9} [3] 
    -8.94069707e^{-10} [4] 
    -3.55271366e^{-17} [ho]
First natural idea: use interval arithmetic for coefficients $r^x$, $\alpha_i^x$ and $e_{ho}^x$

Rounding errors ($\alpha_i^x$) given by the IEEE 754 standard:
- in general, an interval of width $\|\mathbf{1}_p(x)\|$ when $x$ is not just a singleton

But of course, we run into dependency problems, wrapping effect
Each variable of a program has values given as a function (at some control point)

\[ g(r^{x_1}, \ldots, r^{x_k}, e^{x_1}, \ldots, e^{x_k}) \]

where \( r^{x_i} \) and \( e^{x_i} \) are respectively the enclosure of the real number values, and of the inaccuracy error, of variables \( x_i \)
Specificities

Each variable of a program has values given as a function (at some control point)

\[ g(r^{x_1}, \ldots, r^{x_k}, e^{x_1}, \ldots, e^{x_k}) \]

where \( r^{x_i} \) and \( e^{x_i} \) are respectively the enclosure of the real number values, and of the inaccuracy error, of variables \( x_i \)

- non-continuity of \( g \) in general (if statements) - “unstable” tests
- \( g \) can be \( >100 \)KLoC, with \( >10 \)K variables
- \( g \) is constructed on the fly (part of the analysis is actually to find \( g \!\)!)
  - interprocedural calls, depending on context
  - aliases between variables, to be discovered
- we are looking for \textit{invariant sets} of \( g \) in a large space of values, if possible, or else the result of an iteration of \( g \) over a long period of time
- hence computations in an algebra with union and intersection operations as well
...there are in fact two kinds of uncertainties to propagate:

- Uncertainties on the initial values of the variables (which represent inputs to the program) or uncertainties on the parameters of the program (the implemented model)
  - a priori large intervals [given through user-defined assertions]
- Rounding errors, deterministic but only known in general as belonging to some interval
  - a priori much smaller intervals
Recall that:

\[ f^x = r^x + e_1^x + e_{ho}^x \]
\[ = r^x + \bigoplus_{i \in I} \alpha_i^x + e_{ho}^x \]

- We use some form of affine arithmetic for \( r^x \) (and for the errors too as we shall see)
- We can refine further the floating-point enclosure, using *error on bounds*
To compute the floating-point enclosure, we take advantage of the fact that bounds are floating-point numbers.
To compute the floating-point enclosure, we take advantage of the fact that bounds are floating-point numbers

Consider:

\[ x = [0,1] * [0,1]; \]

\[ \text{Error is in } [-\frac{\text{ulp}(1)}{2}, \frac{\text{ulp}(1)}{2}] \text{ for any value of } x \text{ (this is accounted for by terms } e_1^x \text{ and } e_{ho}^x) \]

But the error is null on \( x=0 \) and \( x=1 \)
To compute the floating-point enclosure, we take advantage of the fact that bounds are floating-point numbers

Consider:
\[ x = [0,1]^* [0,1]; \]

- Error is in \(\left[-\frac{\text{ulp}(1)}{2}, \frac{\text{ulp}(1)}{2}\right]\) for any value of \(x\) (this is accounted for by terms \(e_1^x\) and \(e_{ho}^x\))
- But the error is null on \(x=0\) and \(x=1\)

Hence we maintain a correction on bounds \((\delta_-^x, \delta_+^x)\) which controls a potential drift of the bounds

- we compute \(r^x\), then the real number enclosure of \(r^x + e_1^x + e_{ho}^x\)
- then we round these bounds and deduce \((\delta_-^x, \delta_+^x)\) and the new first-order error

The enclosure is then of the form is \([\inf r^x + \delta_-^x, \sup r^x + \delta_+^x]\)
Affine Arithmetic for real number computation ($r^x$)

Proposed in 93 by Comba, De Figueiredo and Stolfi as a more accurate extension of Interval Arithmetic

- **Assignment** of a of a variable $x$ whose value is given in a range $[a, b]$ at label $i$, introduces a noise symbol $\varepsilon_i$:

$$\hat{x} = \frac{(a + b)}{2} + \frac{(b - a)}{2}\varepsilon_i.$$

- **Addition** of affine forms is computed componentwise:

$$\hat{x} + \hat{y} = (\alpha_0^x + \alpha_0^y) + (\alpha_1^x + \alpha_1^y)\varepsilon_1 + \ldots + (\alpha_n^x + \alpha_n^y)\varepsilon_n$$

- **Multiplication** : we select an approximate linear form, the approximation error creates a new noise term:

$$\hat{x} \times \hat{y} = \alpha_0^x\alpha_0^y + \sum_{i=1}^{n}(\alpha_i^x\alpha_0^y + \alpha_i^y\alpha_0^x)\varepsilon_i + \left(\sum_{i=1}^{n}|\alpha_i^x|\cdot|\sum_{i=1}^{n}|\alpha_i^y|\right)\varepsilon_{n+1}.$$ 

(can be improved, in particular with SDP)
The analyzer represents the real coefficients $\alpha_i^x$ by small intervals with MPFR bounds.

When the width of such intervals gets larger, we use new noise symbols.

Extended abstract domain $\mathbb{A}_i^x \hat{\times} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n$ with $\alpha_0^x \in \mathbb{I} \mathbb{R}$ and $\alpha_i^x \in \mathbb{I} \mathbb{R}$ ($i > 0$).
Join (and meet) operations on affine forms

- A natural join between $\hat{r}^x$ and $\hat{r}^y$ is

\[
\hat{r}^{x \cup y} = \alpha_0^x \cup \alpha_0^y + \sum_{i \in L} (\alpha_i^x \cup \alpha_i^y) \varepsilon_i
\]  

(1)

Result might be greater than the union of enclosing intervals (partly corrected by the $(\delta_-^x, \delta_+^x)$).

- But with interval coefficients $\hat{r}^{x \cup y} - \hat{r}^{x \cup y} \neq 0!$
Join (and meet) operations on affine forms

For an interval \( i \), we note

\[
\text{mid}(i) = \frac{i + \bar{i}}{2}, \quad \text{dev}(i) = \bar{i} - \text{mid}(i)
\]

the center and deviation of the interval.

- A better join is

\[
\hat{r}^{x \cup y} = \text{mid}([\alpha_0^x, \alpha_0^y]) + \sum_{i \in L} \text{mid}([\alpha_i^x, \alpha_i^y]) \varepsilon_i + \sum_{i \geq 0} \text{dev}([\alpha_i^x, \alpha_i^y]) \varepsilon_k^u
\]

(2)

- Then we have affine forms with real coefficients again

- Order on affine forms considers noise symbols due to join operations differently than noise symbols due to arithmetic operations
Example (join)

Let \( \hat{r}^x = 1 + 2\varepsilon_1 + \varepsilon_2 \) and \( \hat{r}^y = 2 - \varepsilon_1 \).

- Join on intervals \( r^x \cup r^y \in [-2, 4] \)
- First join on affine forms

\[
\hat{r}^{x \cup y} = [1, 2] + [-1, 2]\varepsilon_1 + [0, 1]\varepsilon_2 \subset [-2, 5]
\]

(larger enclosure than on intervals but still interesting for further computations to keep relations, over-approximation compensated by \((\delta^x_-, \delta^x_+)\))

- Second join on affine forms

\[
\hat{r}^{x \cup y} = 1.5 + 0.5\varepsilon_1 + 0.5\varepsilon_2 + 2.5\varepsilon_3^u \subset [-2, 5]
\]

Same enclosure in this case, but above all \( \hat{r}^{x \cup y} - \hat{r}^{x \cup y} = 0 \)

(Ongoing work on good join and meet operators, order on affine forms, widening and fixpoint computations)
First-order errors

Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigoplus_{l \in L_2} t'_l^{x} \eta_l \]

- \( t'_l^{x} \eta_l \): “uncertain” first-order error terms associated to the operation /
First-order errors

Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigoplus_{l \in L_2} t'^x_l \eta_l + \bigoplus_{l \in L_1} t^x_l \]

- \( t'^x_l \eta_l \): “uncertain” first-order error terms associated to the operation \( l \)
- \( t^x_l \): “exact” first-order error terms associated to the operation \( l \)
Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigodot_{l \in L_2} t'^x_{li} \eta_l + \bigodot_{l \in L_1} t^x_l \]

- \( t'^x_{li} \eta_l \): “uncertain” first-order error terms associated to the operation \( l \)
- \( t^x_l \): “exact” first-order error terms associated to the operation \( l \)
- the other terms are useful for modelling the propagation of the first-order error terms after non-linear operations
First-order errors

Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigoplus_{l \in L_2} t'_{l}^x \eta_l + \bigoplus_{l \in L_1} t_{l}^x + \bigoplus_{i \in I} t''_{i}^x \varepsilon_i \]

- \( t'_{l}^x \eta_l \): “uncertain” first-order error terms associated to the operation \( l \)
- \( t_{l}^x \): “exact” first-order error terms associated to the operation \( l \)
- the other terms are useful for modelling the propagation of the first-order error terms after non-linear operations
  - For instance, the term \( t''_{i}^x \alpha_{i} \varepsilon_i \) comes from the multiplication of \( t_{i}^x \) by \( \alpha_{i} \varepsilon_i \), and represents the uncertainty on the first-order error due to the uncertainty on the value, at label \( i \)
First-order errors

Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigoplus_{l \in L_2} t'^x_l \eta_l + \bigoplus_{l \in L_1} t'^x_l + \bigoplus_{i \in I} t''_i^x \varepsilon_i + \beta_0 + \bigoplus_{p \in P} \beta_p^x \vartheta_p \]

- \( t'^x_l \eta_l \): “uncertain” first-order error terms associated to the operation \( l \)
- \( t'^x_l \): “exact” first-order error terms associated to the operation \( l \)

The other terms are useful for modelling the propagation of the first-order error terms after non-linear operations

- For instance, the term \( t''_i^x \times y \varepsilon_i \) comes from the multiplication of \( t_i^x \) by \( \alpha_i^y \varepsilon_i \), and represents the uncertainty on the first-order error due to the uncertainty on the value, at label \( i \)
- The multiplications \( \varepsilon_i \eta_l \) cannot be represented in our linear forms: we use a new noise symbol \( \vartheta_p \)
First-order errors

Also represented in affine arithmetic (with other noise symbols):

\[ e_1^x = \bigoplus_{l \in L_2} t'^x_l \eta_l + \bigoplus_{l \in L_1} t^x_l + \bigoplus_{i \in I} t''^x_i \epsilon_i + \beta_0^x + \bigoplus_{p \in P} \beta_p^x \vartheta_p \]

- \( t'^x_l \eta_l \): “uncertain” first-order error terms associated to the operation \( l \)
- \( t^x_l \): “exact” first-order error terms associated to the operation \( l \)
- The other terms are useful for modelling the propagation of the first-order error terms after non-linear operations
  - For instance, the term \( t''^x_i \times y \epsilon_i \) comes from the multiplication of \( t^x_i \) by \( \alpha_i^y \epsilon_i \), and represents the uncertainty on the first-order error due to the uncertainty on the value, at label \( i \)
  - The multiplications \( \epsilon_i \eta_l \) cannot be represented in our linear forms: we use a new noise symbol \( \vartheta_p \)

(Notice: values [large intervals] are considered to be of order 0)
Higher-order error terms

- The multiplication of errors introduce higher-order error terms, which are modelled in the following manner:

\[ e_{ho}^x = (t_h^x + \bigoplus_{l \in L_2} t'_{h,l} \eta_l + \bigoplus_{i \in I} t''_{h,i} \varepsilon_i + \bigoplus_{p \in P} \beta_{h,p}^x \vartheta_p). \]
double xi, xsi, A, temp;
signed int *PtrA, *Ptrxi, cond, exp, i;
A = __BUILTIN_DAED_DBETWEEN(20.0,30.0);
/* inverse power of 2 closest to A */
PtrA = (signed int *) (&A);
Ptrxi = (signed int *) (&xi);
exp = (signed int) ((PtrA[0] & 0x7FF00000) >> 20) - 1023;
xi = 1; Ptrxi[0] = ((1023-exp) << 20);
cond = 1; i = 0;
while (abs(temp)>e-8) {
   xsi = 2*xi-A*xi*xi;
   temp = xsi-xi;
   xi = xsi;
   i++;  }
Symbolic execution:
- Input = 20.0 : i = 5, xi = 5.000000e-2 + [-2.81893e-18, -2.76471e-18]
- Output = 30.0 : i = 9, xi = 3.333333e-2 + [-5.28429e-18, 6.21309e-18]

With intervals
- does not converge, even when subdividing

With the relational model, finds \( i \in [5, 9] \) for input \( A \in [20, 30] \) (with subdivisions)
A closest look at results (relational)

Input plus initial error $[20,20.001] + [-1e-05,1e-05]$:

- $(0.03 \text{ sec}, 4.1M)$:
  - $x_i$ in $[4.999750e-2,5.000000e-2] + [-2.68644e-08,2.68644e-08]$
  - $\text{temp}=x_{si}-x_i$ in $[-5.06890974e-9,5.06891107e-9] + [-1.89053e-09,1.89053e-09]$ (the precise estimate of the error allows for a precise computation of the floating-point value)

For larger value domains: subdivision.
A new independent input \( E \) at each iteration of the filter:

```java
double S, S0, S1, E, E0, E1;
int i;

S = 0.0; S0 = 0.0;
E = __BUILTIN_DAED_DBETWEEN(0, 1.0);
E0 = __BUILTIN_DAED_DBETWEEN(0, 1.0);

for (i = 1; i <= 170; i++) {
    E1 = E0;
    E0 = E;
    E = __BUILTIN_DAED_DBETWEEN(0, 1.0);
    S1 = S0;
    S0 = S;
    S = 0.7 * E - E0 * 1.3 + E1 * 1.1 + S0 * 1.4 - S1 * 0.7;
}
```
Second-order filter

- Relational analysis on values and errors
  - with the default precision of the analysis (60 bits):
  - with 200 bits:
    $S$ in $[-1.09, 2.76]$, error $[-1.1e-14, 1.1e-14]$ in 5.2 sec, 27M
Relational analysis on values and errors

- with the default precision of the analysis (60 bits):
  S in \([-4.26, 4.26]\), error \([-5.11, 5.11]\) in 5.1 sec, 25M
- with 200 bits:
  S in \([-1.09, 2.76]\), error \([-1.1e-14, 1.1e-14]\) in 5.2 sec, 27M

(Notice the importance of using MPFR for representing the coefficients in the relational model)
Values and errors stabilized with MPFR\text{bits}=200

Values in $[-1.09, 2.76]$  
Error in $[-1.1e-14, 1.1e-14]$
Propagation of an error on the input:

- Each input has now an error in $[0, 0.001]$
- Relational on errors: $S$ in $[-1.09, 2.76]$, with a stabilized error in $[-0.00109, 0.00276]$
For embedded systems:

- the integrators (and everything built on that, i.e. PID controllers): probabilistic methods, CVFs?
- More generally, analysis of hybrid systems, i.e. systems combining the discrete semantics of the program with a system of PDEs/ODEs for the continuous physical environment (see O. Bouissou’s talk) - see ERTS’06, SCAN’06
- Analysis of code/specification in MatLab/Simulink [fragment]

Scientific codes: analysis of the methods to solve the linear equations (i.e. conjugate gradient etc.) used for instance when solving PDEs by a finite element method

General improvements:

- Computation of under-approximations as well → show the quality of the results
- Improvement of the resolution of the semantic equations by policy iteration; faster and better precision, incremental analysis etc. See CAV’05, ESOP’07