Sardana: an Automatic Tool for Numerical Accuracy Optimization [1]

Arnault Ioualalen and Matthieu Martel
University of Perpignan Via Domitia
LIRMM - UMR5506
{arnault.ioualalen, matthieu.martel}@univ-perp.fr

SCAN'12 - 24/09/12

Presentation by Laurent Thévenoux
laurent.thevenoux@univ-perp.fr

[1] Work realized in the SARDANES project (UPVD, ENS and UBO), granted by the FNRAE

Floating-point arithmetics

Supposed to match the arithmetics of the reals

But it introduces a lot of errors from both:

- rounding errors of values (ex: 0.1)
- rounding errors of calculation (ex: $X + a$ with $X \gg a$)

Example

$$(x - 0.1) \times (x - 0.1) \neq_{\mathbb{F}} x^2 - 0.2x + 0.01$$

But which one is more precise ? and for what value of x ?

How to implement a "good" formula ?

- heuristics for simple cases (ex: sorting terms)
- proved algorithms (too specific and costly)

Improving the accuracy ?

Static analyzers available

ASTRÉE, Fluctuat both rely on abstract interpretation of programs

Fluctuat calculates a safe approximation of the rounding errors

- with interval arithmetics by considering `ulp` of numbers
- now with a more precise domain: zonotopes

+ very precise analysis
+ input values are described by intervals
− doesn't help to correct a program, it only raises alarms

Improving the accuracy automatically: Sardana

We want an automatic tool to improve the accuracy of programs

1. program size is growing ($\approx 100k$ *loc.*)
2. floating-point arithmetics is not intuitive
3. we can't test any inputs of the program

Sardana is compiler for Lustre language (synchronous programming)

- synchronous programs runs for hours, days or more
- often embedded with critical equipments (ex: planes, power-plants)
- code is written as repeated cycles of instructions

We want to synthesize a new and more accurate program

- for all the inputs of the initial program (intervals)
- for any duration of execution

How to synthesize a more accurate program ?

We need to transform it into a semantically equivalent one

Use numerical transformations producing equivalent formulas

- associativity
- commutativity
- distributivity, factorization
- propagation of minus operator

But we can't look exhaustively for a better equivalent formula

- $(2n - 3)!!$ way to calculate a sum of n terms
- exponential number of way to evaluate a polynomial function

$\Rightarrow$ We need abstraction to narrow down this research space

Abstraction of equivalent programs

APEG: Abstract Program Expression Graph

Features:

- handle intervals to abstract sets of traces of execution
- built from the syntactic tree
- constructed by polynomial algorithms
- stays polynomial in size of the initial program
- represents an exponential number of equivalent programs

APEGs represent an exponential number of expressions with

1. abstraction boxes
2. equivalence classes

Abstraction boxes

An abstraction box $\boxed{*, (p_1, \ldots, p_n)}$ is defined by

- a symmetric associative operator $*$ like $+$ or $\times$
- a list of constants, variables, expressions or abstraction boxes

An abstraction box represents by definition all the expressions constructed with the operator $*$ over $p_1, \ldots, p_n \Rightarrow$ at least $(2n-3)!!$ expressions

$(2n-3)!! \neq ((2n-3)!)!$
$(2n-3)!! = 1 \times 3 \times 5 \times \cdots \times (2n-3)$

Embedded abstraction box allows to represent more expressions

$$\boxed{*, (p_1, \ldots, p_n, \boxed{*', (p'_1, \ldots, p'_k)}\ )} \Rightarrow (2n-3)!! \times (2k-3)!!$$

$$(2n - 3)!! = 1 \times 3 \times 5 \times \cdots \times (2n - 3)$$

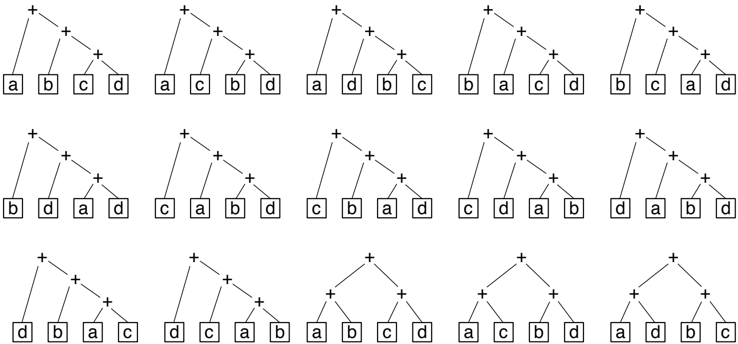


Figure: Every possible sums of 4 terms ⇒ 15 distinct expressions, distinct means that the accuracy could be different!

Equivalence classes

Equivalence classes merge equivalent expressions

equivalent expressions obtained by associativity, commutativity, distributivity. . .

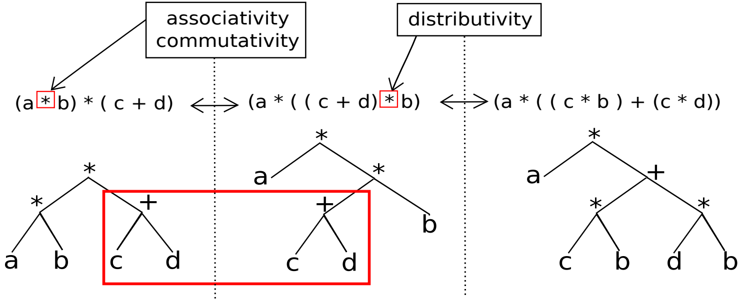


Figure: Example of transformations of expressions

The equivalence class concept

An equivalence class is a set of nodes which are the roots of expressions equivalent one to each other

Combining equivalence classes allow to represent an exponential number of expressions without exploding in size

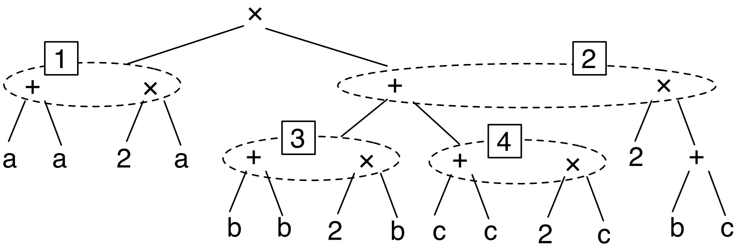


Figure: An APEG representing 10 equivalent expressions

Abstract Program Expression Graphs definition

We define the APEG set $\Pi_{\triangleright}$ inductively as the small set such as

Definition

1. $a \in \Pi_{\triangleright}$ where a is a leaf (constant, variable, interval)
2. $*(p_1, p_2) \in \Pi_{\triangleright}$ where $*$ is an operator apply on $p_1 \in \Pi_{\triangleright}$ and $p_2 \in \Pi_{\triangleright}$
3. $\boxed{*, (p_1, \ldots, p_n)} \in \Pi_{\triangleright}$ is an abstraction box, p_i are APEGs
4. $\langle p_1, \ldots, p_n \rangle \in \Pi_{\triangleright}$ is an equivalence class of equivalent expressions

To construct APEGs we use two kind of polynomial algorithms

- homogenization algorithms
- expansion algorithms

APEG construction: homogenization algorithms

Polynomial homogenization algorithms:

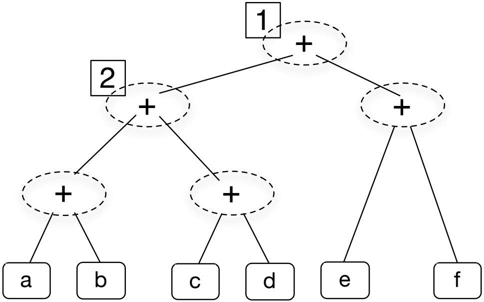
- distribute multiplications
- factorize common factors
- propagate subtractions through additions and multiplications

Homogenization algorithms introduce homogeneous parts

Homogeneous part: where a symmetric associative operator repeated itself

Homogeneous parts are crucial to introduce large abstraction boxes

APEG construction: horizontal expansion algorithm



We perform one walk through the APEG
$\Rightarrow O(n)$

We add at most:
$3(n-1)$ boxes

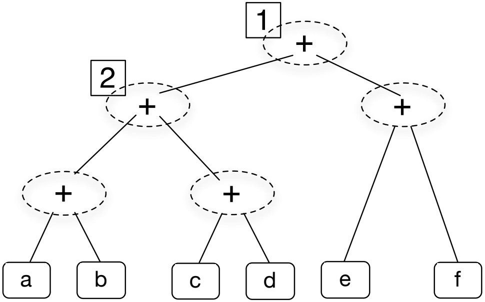
we add in equivalence class 1,

- $\boxed{+,\ (a,b,c,d)} + (e+f)$
- $((a+b)+(c+d)) + \boxed{+,\ (e,f)}$
- $\boxed{+,\ (a,b,c,d,e,f)}$

and in equivalence class 2

- $(a+b) + \boxed{+,\ (c,d)}$
- $\boxed{+,\ (a,b)} + (c+d)$
- $\boxed{+,\ (a,b,c,d)}$

APEG construction: vertical expansion algorithm



We perform one walk through the APEG $\Rightarrow O(n)$

We add at most: $(n-1)+n$ boxes

We add in equivalence class 1:

- $(a+b)+$ [+,(c,d,e,f)]
- $(c+d)+$ [+,(a,b,e,f)]
- $(e+f)+$ [+,(a,b,c,d)]
- $a+$ [+,(b,c,d,e,f)]
- $b+$ [+,(a,c,d,e,f)]
- $c+$ [+,(a,b,d,e,f)]
- $d+$ [+,(a,b,c,e,f)]
- $e+$ [+,(a,b,c,d,f)]
- $f+$ [+,(a,b,c,d,e)]

Synthesizing a more accurate program

APEG represent an exponential number of equivalent programs

To evaluate one program we use an over-approximation of the roundoff errors (using intervals).
ex: real error of $a + b \leq ulp(a + b) +$ errors on a and b.

We have still to synthesize more accurate programs from

- abstraction boxes
- equivalence classes

Synthesizing an accurate formula from an abstraction box

Heuristic is a greedy pairing algorithm

At each step we search for the pair (p_i, p_j) of terms where the error of $p_i * p_j$ is minimal

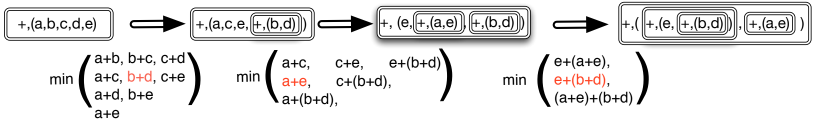


Figure: Expression synthesis from abstraction box $+, (a, b, c, d, e)$

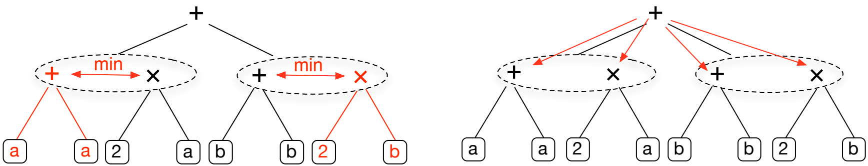
Complexity

$n - 2$ steps of pairing, at most n new pairs to consider each time $\Rightarrow O(n^2)$ complexity

Synthesizing an accurate formula from an APEG

Heuristic is a limited depth search (depth is set by the user)

We select the way an expression is evaluated by considering only the best way to evaluate its sub-expressions to a specific depth



Naive local search (depth=0) $\Rightarrow O(n)$

Local search (depth=1) $\Rightarrow O(n^2)$

Complexity is exponential

if *depth* is large enough we synthesize the optimal solution
But then it is in exponential time!

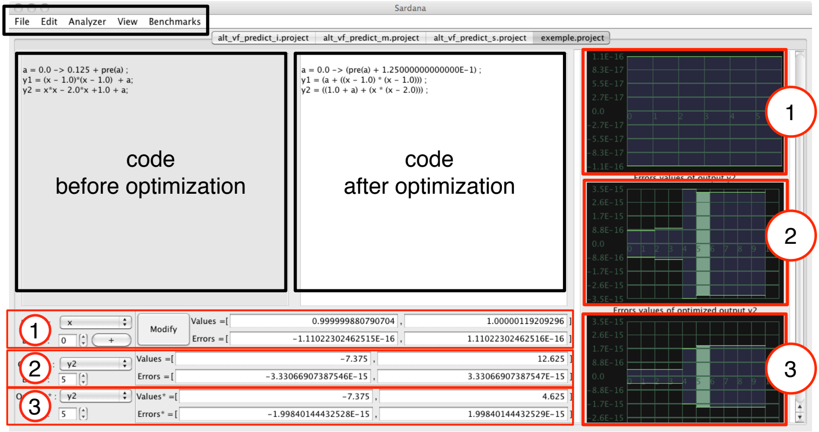
The Sardana tool

A static analyzer

- written in OCAML
- use GMP, MPFR libraries to represent number of any format
- works on floating-point number as well as fixed point numbers
- needs the range of the input values from the user
- takes a Lustre code and returns an optimized Lustre code

A graphical interface (optional)

- written in JAVA
- allow to represent codes accuracy in a user-friendly way
- allow to parametrize more easily the analyzer



Charts 1, 2, 3 show the evolution along the execution of the:

- 1: floating point values and errors of an input
- 2: floating point values and errors of an initial output
- 3: floating point values and errors of an optimized output

Case study: optimization of summations

Optimization of summations evaluation

Sums are used in various algorithms
There are many ways to write them $\rightarrow (2n-3)!!$

ill conditioned sums[1]

1. positive values, large values among little values
2. positive values, large values among little and medium values
3. both signs, large values among little values
4. both signs, large values among little and medium values

These configuration causes

- absorption issues
- catastrophic cancelations issues

[1] Large value $\approx 10^{16}$, medium values ≈ 1, small values $\approx 10^{-16}$

Case study: optimization of summations

Optimization of all possible summations of 7, 8 and 9 terms

$\approx$ 50% accuracy average improvement for any ill conditioned sum

#Terms	#Expressions	Configuration	%Avg Gain
7	10.395	1	36%
		2	63%
		3	51%
		4	47%
8	135.135	1	38%
		2	65%
		3	54%
		4	48%
9	2.027.025	1	40%
		2	68%
		3	53%
		4	48%

Case study: optimization of Taylor expansions

Taylor expansions of usual functions are widely used in programs

But the evaluation of a polynomial is not accurate near a root

We optimized exhaustively all the evaluation schemes, near a root, for several order or expansion of the Taylor expansion of a function

Function	Order	#Expressions	%Avg Gain
$\cos(x)$	4	62	12%
	6	15.924	17%
$\sin(x)$	5	412	22%
	7	235.270	28%
$\ln(x+2)$	3	43	14%
	4	2.128	17%
	5	323.810	23%

Conclusion

- APEGs allow to represent efficiently many equivalent expressions
- Sardana manipulates whole programs not only expressions
- Sardana presents some convincing results

APEG improvements

- new expansion algorithms to add more abstraction boxes
- new synthesis algorithms to improve the accuracy of programs

More experimental results

- On real case examples (complex avionic code written in Lustre)
- With fixed-point arithmetics

Questions ?

Thank you for you attention !

arnault.ioualalen@univ-perp.fr