Automatic Generator of Floating-Point Inputs by Constraint Solving

Gene Der Su and Jian (Kevin) Xu

University of California, Davis
{gdsu, kevxu}@ucdavis.edu

ABSTRACT

Program constraint with floating-point value is very common for numerical programs. A huge loss can be caused by floating-point exceptions, such as overflow, underflow, inexact number and invalid number. Floating-point inputs generation tools are needed to detect exceptions. However, symbolic execution tools like KLEE does not perform well in generating floating-point inputs. This is because its inability to solve multivariate and nonlinear constraints. In this paper, we implement a small automatic generator in Matlab to solve floating-point constraints. For simplicity, we focus on polynomial constraint with at most 5 variables and exponent indices ranging from 1 to 5. As a result, we got 100% coverage for inputs to satisfy most of the constraints. However, when the largest index is even (like 4), the coverage can go down to 96.24%. Algorithm to check the upper and lower bound of the constraint is developed to improve the coverage (from 40% to 98%). For a better coverage and for multi-constraint cases, a better concretization method is needed.

1. INTRODUCTION

Floating-point arithmetic is used by numerical software, which has a wide application in control system of industrial manufacture, transportation and health care. A floating-point exception can cause huge loss. US $370 million loss is caused by the self-destruction of European Space Agency's Ariane 5 rocket, due to a floating-point overflow (Barr et al)[1]. There are also risk of floating-point exception in car brake system and remote surgery system, which will endanger human life. Thus, developing an automatic testing method to generate inputs to detect the floating-point method will be promising.

Floating-point numbers is a finite representation of real numbers that we use every day. It contains two components: exponent and mantissa both with sign occupying one bit. In IEEE 754 standard, a single precision floating point has a length of 32 bits, where 24 bits are used as mantissa and 8 bits are used as exponent, constructing a minimum representable number of 1.18×10^{-38} and a maximum representable number of 3.40×10^{38}. A double precision floating point has a length of 64 bits, where 53 bits are used as mantissa and 11 bits are used as exponent, constructing a minimum representable number of 2.23×10^{-308} and a maximum representable number of 1.79×10^{308}. We can keep increasing the length to reduce the risk of falling out of the bound as well as reduce the inaccuracy of the floating-point representation. However, real number spans an infinite set, no matter how long the floating point, we cannot completely remove the chance of getting a floating-point exception. Therefore, we need a way to determine these exceptions and take special care of such cases.

There are four types of floating-point exceptions: Underflow, Overflow, Inexact, and Divide-by-Zero/ Invalid. Underflow and overflow are the most common floating-point exceptions out of all. They contained over 90% of the total floating-point exceptions (Barr et al)[1]. An underflow happens when a value is smaller than the smallest representable number, so the system can only truncate it to zero. An overflow is completely opposite to underflow. It happens when a value is larger than the largest representable number, so the system can only assign it +/- infinity. An inexact value is a value that is in between the smallest and largest representable but still cannot be represented by a finite-length floating point. Examples such as π, e, or $\sqrt{2}$ are irrational numbers or numbers that simply contain more significant digits than the capability of the floating point. Such number will be truncated to the nearest representable floating-point value. Finally the last floating-point exception is divide-by-zero and invalid (e.g. 1/0 and $\sqrt{-1}$. Such numbers are undefined in the original mathematical theorem, therefore floating-point will assign such number as +/- infinity and NaN (not a number) and not giving it a real value.

constraint	source
$(1.5 - x1 * (1 - x2)) == 0$	Beale
$(-13 + x1 + ((5 - x2) * x2 - 2) * x2) + (-29 + x1 + ((x2 + 1) * x2 - 14) * x2) == 0$	Freudenstein and Roth
$pow((1 - x1), 2) + 100 * (pow((x2 - x1 * x1), 2)) == 0$	Rosenbrock
$((pow(((x * (sin((((y * 0.017) - (z * 0.017)) + (((((((pow(w, 2.0))/((sin((t* 0.017)))/(cos((t*0.017)))))/68443.0)*0.0)/w)*-1.0)*x)/(((pow(x, 2.0))/((sin((t* 0.017)))/(cos((t * 0.017)))))/68443.0)))))) - (w * 0.0)), 2.0)) + (pow(((x* (cos((((y * 0.017) - (z * 0.017)) + (((((((pow(w, 2.0))/((sin((t * 0.017)))/(cos((t* 0.017)))))/68443.0)*0.0)/w)*-1.0)*x)/(((pow(x, 2.0))/((sin((t*0.017)))/(cos((t* 0.017)))))/68443.0)))))) - (w * 1.0)), 2.0))) == 0.0$	TSAFE
$((exp(x) - exp((x * -1.0)))/(exp(x) + exp((x * -1.0)))) > (((exp(x) + exp((x * -1.0))) * 0.5)/((exp(x) - exp((x * -1.0))) * 0.5))$	PISCES
$x^{tan(y)} + z < x * atan(z) \wedge sin(y) + cos(y) + tan(y) >= x - z \wedge atan(x) + atan(y) > y$	manual

Table 1. Examples of constraints.

In some numerical programs, there can be a lot of constraints containing floating-point values. Our work focuses on solving the floating-point constraint of a program. Table 1 (Souza et al.)[4] shows some examples of representative constraints. The constraints can be very complicated like in TSAFE and PISCES, which are subjects from NASA. These constraints contain multivariants with nonlinear mathematical functions, which make them very difficult to solve. In our case, we only consider the polynomial constraints (like the first 2 cases in the table). Instead of getting the constraint

from real programs (by using tools like KLEE), we generate the coefficient and exponent index of the constraints and try to solve them.

2. MOTIVATING EXAMPLE

A famous example of a number that can be represented in base 10 accurately but cannot be represented in floating-point binary is 0.1. In base 10 it is 0.1, but in binary it will be 0.0001100 (in 8 digit). In a single precision floating-point value, 0.1 will be forced to round to the nearest representable floating-point value and since there are 24 bit of mantissa, the rounding error will be around 2^{-24}. Even though this rounding error looks small, if we did not take special care of it, it will accumulated a large error in our floating-point program. A piece of code that adds 0.1 twenty thousand times will accumulate a notable difference. The output of the function is expected to be 2000 in real numbers, but instead it outputted as 1999.658813 solely due to the accumulation of error (Benz)[9].

```
float t = 0.0f;
int i;
for (i = 0; i < 20000; i++)
     t += 0.1f;
printf("%f\n",t);
```

Another case to take special care is the roundoff error when two nearly equal numbers are subtracted from each other. One example is for a function $f(x) = \frac{1-cos(x)}{x^2}$with $x = 1.2 \times 10^{-5}$. Assuming using a 10-digit floating-point arithmetic, $cos(x)$ will be 0.9999999999 so $1 - cos(x) = 0.0000000001$. Therefore $\frac{10^{-10}}{1.44\times18^{-10}} = 0.6944$. However, we know the identity $cos(x) = 1-2sin^2(x/2)$, substituting it into the function we will get $f(x) = \frac{1}{2}(\frac{sin(x/2)}{x/2})^2 \approx 0.5$ for a small angle approximation of sin function. As we see because of 1 and $cos(x)$ are really close, the roundoff error becomes really large and gives us an answer that is completely different from the expected answer with a relative error of 0.39.

Finally, another example we need to be careful about is that when adding up numbers that are really different in their magnitudes. Often time in a numeric program, we will do an infinite sum. Sum like $\sum_{k=1}^{\infty} \frac{1}{k^2}$ is expected to be $\frac{\pi^2}{6} = 1.644934066848$. Normally we will just write a function that sums up each $\frac{1}{k^2}$ from $k = 1$ to a large number like 10^9. However, if we use floating point to represent these numbers and sum them up as $\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + ...$it will return 1.64472532, where it only has four correct significant digits in the program. The problem is due to at $k = 4096$, $\frac{1}{k^2} = \frac{1}{4096^2} = 2^{-24}$, which is right on the smallest value a single precision floating point can hold. Therefore all terms for $k > 4096$ will be dropped off in the final sum and leaving we with an incorrect result. One solution to fix this problem is to sum from the smaller numbers to the larger ones. In that case the floating point can use denormalized numbers to represent sum of small number and keep enough significant digits at the same time. However, that does not solve the problem when we are doing a dot product of two vectors because we do not always know which value is small and which is large. It only concludes that computing a sum in floating point will always cause the loss of significant digits.

3. TECHNICAL APPROACH

KLEE has the capability of automatically generating inputs for testing a program. However, the floating-point programs has multivariate and nonlinear constraints, KLEE cannot provide the inputs for them automatically. The biggest challenge KLEE faces is the SMT solver it uses. KLEE's algorithm surely works on both integer and floating-point programs, but because of the constraints in floating-point programs, the SMT solver cannot solve the constraints and provide inputs. Our approach will be focused on finding the solutions about solving the constraints in a floating-point program.

In Barr et al's paper *Automatic Detection of Floating-Point Exception*[1], they discussed about using Microsoft Z3 as the SMT solver in KLEE instead of the original one. What they found was that it can only handle less than 1% of the constraints while rejecting all else because of the multivariate and nonlinear constraint issues. They also tried to directly feed the constraints into newer version of Z3 which just added features of solving nonlinear multivariate constraints, but the result is that Z3 can only handle a small percentage. It becomes necessary to concretize multivariate constraints to univariate ones and linearize the constraints before applying to Z3.

The first step to avoid the floating-point exception is to transform the program into floating-point exception free one. Knowing which condition will give floating-point exceptions we can construct a transformations table as following:

$$T(x \odot y) = \begin{cases} Overflow & if\ |x \odot y| > \Omega \\ Underflow & if\ 0 < |x \odot y| < \lambda \\ x \odot y & otherwise \end{cases} \quad (1)$$

$$T(x/y) = \begin{cases} Invalid & if\ x = 0 \wedge y = 0 \\ Divide - by - Zero & if\ x \neq 0 \wedge y = 0 \\ Overflow & if\ |x| > |y|\Omega \\ Underflow & if\ 0 < |x| < |y|\lambda \\ x/y & otherwise \end{cases} \quad (2)$$

$$T(sqrt(x)) = \begin{cases} Invalid & if\ x < 0 \\ d, where\ d \times d = x & otherwise \end{cases} \quad (3)$$

$$T(exp(x)) = \begin{cases} Overflow & if\ x > log(\Omega) \\ Underflow & if\ x < log(\lambda) \\ d & otherwise \end{cases} \quad (4)$$

$$T(cos(x)) = d \quad where\ -1 \leq d \leq 1 \quad (5)$$

$$T(sin(x)) = d \quad where\ -1 \leq d \leq 1 \quad (6)$$

$$T(pow(x,y)) = \begin{cases} Invalid & if\ x = 0 \wedge y \leq 0 \\ d & otherwise \end{cases} \quad (7)$$

Note $\odot \in (+, -, *)$ in the first table. In this transformation step, floating-point exceptions are found and their positions are located. These rules are also developed to deal with other mathematical function, such as $tan(), sinh()$, cosh(), etc. This step is simple and efficient to find floating-point exceptions in the program. In Barr et al's work[1], 41% of the floating-point exceptions are reported by using this kind of transformation method.

The second step is to implement the algorithm to concretize multivariate constraints to univariant constraints. All the symbolic variables will be used to construct a DAG (directed acyclic graph), and they will be concretized in topological sort order.

The final step is to implement the algorithm to solve univariant constraints. If the polynomial constraint is nonlinear, it will be rewritten into equivalent disjunction of linear interval predicates. This can be done using polynomial solver like GSL 1.14. The algorithm will then take these roots found in the polynomial solver and construct the regions that can satisfies the constraints. In the end, it will either output a set of floating-point numbers that solves the constraints or terminate and answer there are no such solution.

The two main challenges are multivariate, and nonlinear constraints. Basically, there are 3 steps to find a floating-point solution: (1) concretization, (2) linearizing univariate nonlinear constraints and (3) looping over intervals to find the solution. In the first step, concretization is done to convert multivariate to univariate polynomial constraints, and the symbolic variables that appear in multivariate numeric constraints in DAG (directed acyclic graph) are concretized in topological order. Then, a quadratic algorithm is applied to replace nonlinear polynomial with an equivalent linear disjunction, thus linearizing the formula of reals. After that, solutions are found after looping over the intervals that the linear constraint defines. By looping, it means picking a small interval first and trying the nearest floating number geq (greater or equal) or leq (less or equal) for a solution. If it does not solve the constraint, increase the interval and try another floating. The intervals are generated by the Linearizing Algorithm in the previous step.

An small example of a kind of constraint we are trying to solve is ${x_1}^2 + {x_2}^3 + {x_3}^5 < 10$. The approach is to first concretize two variables out of the three variables with a concrete value 1. We can choose to make $x_2 = 1$ and $x_3 = 1$, so we just need to solve ${x_1}^2 < 8$. In this case we are trying to solve for a variable less than a constant, therefore we will be attempting to solve ${x_1}^2 = 7$ and that will give us $x_1 = 2.646$.

Using this algorithm, we are able to cover the case of odd polynomials 100% of the time because odd polynomials are guaranteed to have at least one real root. However, if we have a constraint like ${x_1}^2 = -1$, it will not be possible to solve and generate floating-point values. Moreover, in another case like ${x_1}^2 + {x_2}^2 = 0.5$, we know there are solutions that can satisfy the constraint, but we have to choose the concrete value wisely. If we just blindly choose $x_2 = 1$, then we can not solve for x_1 as a real number. One solution is to impose a bound on all variables, and figure out the concrete values that can make the variables satisfy the constraint. We will solve for the maximum or minimum of a given variable by ignoring all other variables. For this example we will solve the maximum or minimum of $f_1 = {x_1}^2$ and $f_2 = {x_2}^2$ and get the values of minimum $f_1 = 0$ at $x_1 = 0$ and $f_2 = 0$ at $x_2 = 0$. This will form a global minimum of 0 and telling us k can be anything greater or equal to 0 and that this constraint is satisfiable. After we eliminate the unsatisfiable constraint, it becomes much easier to find the concrete values. In this example, we know f_1 alone can cover the value of k, so we concretize $x_2 = 0$ leaving us solving the equation ${x_1}^2 = 0.5$ and give the solution $x_1 = 0.7071$.

The algorithm will take in the inputs: number of variables, n, maximum exponent, m, coefficients from each variables and exponents, c, constant on the right hand side of the equation, k, and the sign of the equality of the constraint, signf. It will output: a constant whether the constraint is satisfiable, r (1 as satisfiable and 0 as unsatisfiable) and a vector of the floating point generated by the algorithm, x. The algorithm goes as the following:

Algorithm for floating-point constraint solver

if *ODD exponent constraint* **then**
- (1) Concretize all variables to 1 except the first variable;
- (2) Solve for the roots of the equation with one variable;
- (3) Use the first real root as the generated value;

else if *EVEN exponent constraint* **then**
- (1) Determine the minimum/maximum of each individual variables;
- (2) Find the global minimum and/or maximum of the function;
- (3) Determine whether the constraint is satisfiable;
- **if** *Unsatisfiable* **then**
  - Output "Unsatisfiable"
- **else**
  - (1) Use the minimum/maximum information to determine which variables to concretize to 0;
  - (2) Solve the root of last variable as the generated value;

4. EVALUATION METHODOLOGY

4.1 Running time and coverage

Our tool will be a small program that takes input of path constraints and output whether the constraints are satisfiable, and if so, output floating-point values that can help to reach the inputted constraints. The most direct way to measure the performance of such a tool is by measuring the time it take to find the solution. In general, our testing path constraint has a form of $\sum_{j=1}^{m} \sum_{i=1}^{n} c_{i,j} x_i^j \oplus k$, where $c_{i,j}$ is a constant coefficient in front of the i^{th} variable with j exponent, x_i is the i^{th} variable, j is the exponent of the i^{th} variable, $\oplus \in (<, =, >, \leq, \geq)$, and k is an arbitrary constant. First plan is to randomly generate a path constraint with different coefficients, the equality, and k. We can set different number of variables (i.e. n) and the maximum degree of the variable (i.e. m) constant in each case and run the tool one hundred time to find the average of each different values of n and m. Since the larger the n and m are, the more terms there will be, we are planing to do evaluations on $n, m \in [1, 5]$ to generate a total of 25 cases to show the performance.

The second plan if we still have time in the end we can start trying out solving multiple constraints at once. There will then contain multiple equations of the single constraint. Since it introduced another order of complexity, we will probably do $n, m, o \in [1, 3]$ or fix the number of variables and/or exponent to one in order to test the performance on solely solving multiply constraints.

The following are the average time in seconds for each run, average out from doing 10,000 runs.

	m=1	m=2	m=3	m=4	m=5
n=1	1.776e-4	4.306e-4	1.504e-4	2.758e-4	1.832e-4
n=2	1.623e-4	3.283e-4	1.741e-4	4.002e-4	1.752e-4
n=3	1.301e-4	4.428e-4	1.803e-4	5.726e-4	1.749e-4
n=4	1.510e-4	5.073e-4	1.472e-4	6.385e-4	1.620e-4
n=5	1.660e-4	6.140e-4	2.097e-4	7.786e-4	1.876e-4

Table 2. Average running time in seconds

The following are the percentage of the coverage from doing 10,000 runs, in which

Coverage =

$$\frac{Number\ of\ satisfied\ cases\ by\ our\ tool}{Number\ of\ total\ satisfiable\ cases}$$

$\times 100\%$

	m=1	m=2	m=3	m=4	m=5
n=1	100%	100%	100%	100%	100%
n=2	100%	97.04%	100%	96.24%	100%
n=3	100%	97%	100%	97.12%	100%
n=4	100%	97.85%	100%	97.97%	100%
n=5	100%	98.9%	100%	99.43%	100%

Table 3. Percentage of coverage

From the result table, we can observe a few things:

- Number of variables in the constraint will not affect the result in terms of time and coverage.
- Maximum exponent will not affect the results.
- Odd and even exponents do affect the result in both time and coverage. Specifically, even exponents are $2-4$ times slower than odd exponents and when $m = 4$ the coverage is not always 100%.

The reason that the number of variables not affecting the performance is because we always concertize all the variables but one. That means we are alway attempting to solve an equation with one variable regardless what the setting is. Also, it is obvious to see why solving the even exponent constraints are much slower than solving the odd ones. For the even exponent constraints, we are not guaranteed to have at least a real root, so we need to calculate the extra information about the bounding of each variables. The calculation in the algorithm took majority of the time to solve the maximum and minimum. Moreover, it is not always easy to determine which variable to solve and which variables to concertize. That is why it took more than twice of time to find a solution for even exponents than odd exponents. In the future, we can probably work on optimizing the algorithm for solving an even exponent constraints as well as improving the coverage of even exponents.

A question raised as whether there may be false positive in our floating point generator saying that the constraint is unsatisfiable, but in reality it is satisfiable. It is clear to see that for odd number of exponent, any constraint is satisfiable, so the problem really lies in the even number of exponent. For even exponent, our algorithm will check for the global maximum and minimum and compare with the k value to see if it is possible to satisfy the equation. The algorithm pick out all the case that will lead to the constraint unsatisfiable. Therefore, if the tool is implemented correctly, we should not have any case that outputs unsatisfiable, but is really satisfiable.

However, for a constraint such as $(x_1 + 2)^2 + (x_2 + 2)^2 = 1$, our tool is not able to output the correct input that satisfies the constraint. This causes the coverage to be less than 100% in the exponent of 2 and 4. We find such case will need both variables to be specific values in order to satisfies the equation. Due to the algorithm concertize all but one variable to zero, the constraint is therefore can not be satisfied. In the future work, we can add a smarter choice of the concertized values to increase the coverage for these cases.

4.2 Examples of constraint solving

4.2.1 *Satisfied constraint*

When the maximum index of exponent is odd, the constraint can always be satisfied. Here is an example when n=5 and m=5.

$$\begin{aligned} &0.5447x_1 + 0.9129{x_1}^2 - 0.6191{x_1}^3 - 0.4655{x_1}^4 + 0.9946{x_1}^5 \\ &+ 0.2500x_2 + 0.0132{x_2}^2 + 0.6500{x_2}^3 + 0.9432{x_2}^4 - 0.0404{x_2}^5 \\ &+ 0.9473x_3 + 0.1055{x_3}^2 + 0.8696{x_3}^3 - 0.4645{x_3}^4 - 0.3716{x_3}^5 \\ &- 0.1780x_4 - 0.0363{x_4}^2 - 0.5614{x_4}^3 - 0.5838{x_4}^4 - 0.0558{x_4}^5 \\ &- 0.9398x_5 - 0.4217{x_5}^2 + 0.2957{x_5}^3 - 0.7398{x_5}^4 - 0.5765{x_5}^5 \\ &> 0.4450 \end{aligned}$$

Example 1: Constraint that can be satisfied

The generated input:
$x_1 = 1.5196, x_2 = 1.0000, x_3 = 1.0000, x_4 = 1.0000, x_5 = 1.0000$ and $f(1.5196, 1, 1, 1, 1) = 6.2783 > 0.4450$

4.2.2 *Unsatisfied constraint*

When the maximum index of exponent is even, there are about 2% of such cases that the generated constraint cannot be satisfied.

$$\begin{aligned} &-0.4847x_1 + 0.8676{x_1}^2 - 0.2709{x_1}^3 + 0.7819{x_1}^4 \\ &+ 0.4646x_2 + 0.9364{x_2}^2 - 0.6033{x_2}^3 + 0.6978{x_2}^4 \\ &+ 0.0857x_3 - 0.3179{x_3}^2 - 0.1652{x_3}^3 + 0.8966{x_3}^4 \\ &- 0.3374x_4 - 0.3033{x_4}^2 + 0.4198{x_4}^3 + 0.2463{x_4}^4 \\ &- 0.4986x_5 + 0.4031{x_5}^2 - 0.0648{x_5}^3 + 0.7948{x_5}^4 \\ &\leq -0.6933 \end{aligned}$$

Example 2: Constraint that cannot be satisfied

No generated input.

Analysis: Using our algorithm, $f(x_1) \in [-0.0689, \infty)$, $f(x_2) \in [-0.0495, \infty)$, $f(x_3) \in [-0.0359, \infty)$, $f(x_4) \in [-0.3477, \infty)$, $f(x_5) \in [-0.1188, \infty)$, $f(x_1, x_2, x_3, x_4, x_5) \in [-0.6207, \infty)$. Since $-0.6207 > -0.6933$, the constraint cannot be satisfied.

5. RELATED WORK

5.1 General introduction of SMT solvers

Basically, SMT solvers are used for solving the path constraints. SMT is short for "satisfiability modulo theories", which is design for solving SAT problem (decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables). SMT solver is widely used in dynamic symbolic execution, because the semantics of most program statements can be easily modeled by using the theories of the solvers. SMT solver played a central role in tools like CUTE, DART, KLEE, SAGE and Pex (De Moura and Bjørner)[3] .

5.2 Algorithms to convert multivariate to univariate

The conversion from a multivariate constraint to univariate one can be reduced to CSP (constraint satisfaction problem) (Scott and Sorkin)[7]. Barr et al[1] uses DAG (directed acyclic graph) as the concretization method, which is an efficient way of solving the CSP. The most commonly used strategy of solving CSP is branch and prone, which is to divide the simplified initial problem into subproblems, whose solutions union is equivalent to the initial one. One of the techniques used is domain reduction, which reduces the domain of variables without discarding any solution (Vu, Schichl, and Sam-Haroud)[8]. Before the introduction of DAG, tree is a data structure used to represent the domain reduction. Forward evaluations and backward projections are performed recursively on the whole tree representing the constraint. DAG improves the performance of the tree structure, by overcoming the limitation that each constraint is propagated individually. Also, DAG accurately handles the influence of subexpressions shared by several constraints. The algorithm of DAG is of some complexity, and we will try to implement it if time is permitted.

5.3 Solve nonlinear constraints

One of the challenges is to solve nonlinear constraints. The default solver used in KLEE is STP, which is designed using interval-based arithmetic constraint solving technique. Also, Barcelogic is another solver for nonlinear constraints. It linearizes the constraints by assigning a finite domain to variables occurring in nonlinear constraints (Borralleras et al.)[2]. The other way is using a quantifier elimination method such as Fourier-Motzkin procedure for linear arithmetic, which treats non-linear terms as if they were linear (Monniaux)[6] . The solvers based on quantifier elimination are: Mathematica, QEPCAD, Redlog-CAD and Redlog-VTS. There are other state-of-art non-linear SMT solvers, such as CVC3 and MiniSMT.

Most SMT solvers do not have a good support of solving the nonlinear constraints, since solving nonlinear constraints over real domain can be expensive. In Barr et al's work[1], they feed the constraints from GNU Scientific Library into iSAT, but it only handles less 1%. Most of the constraints are rejected because they contains large constant that exceeds iSAT's bounds. Z3 also does not perform well and it only handles a small percentage. For the rejection cases, it returns unknown, time out, or parse errors.

5.4 Other works about program transformation

The idea of doing a program transformation is similar to the pre-substitution introduced by Goldberg (Goldberg)[5] . Pre-substitution is a straightforward hardware implementation to avoid exceptions. Once the type of exception is determined, it can be used to index a table that contains the desired result of the operation. For example, for the computation of $sin(x/x)$, $x = 0$ will result in an exception. To avoid the error caused by the exception, a handler is written to return the value 1 for x/x in $sin(x/x)$, even when an invalid operation occurs. This explains how to implement the pre-substitution and it is usually done in hardware. However, this hardware implementation is not widely accepted by manufactures so doing program transformation is a good way of making up for it.

5.5 Floating-point exceptions

Most of the exceptions are overflow and underflow (91.9%), and many overflow and underflow are avoidable, which means those overflows are intermediate and two overflow numbers will be canceled in the end. A simple example is $y = x + (-x)$. If $x = \Omega$, it is a overflow. However, y can be calculated to 0 without referring to x as a floating-point value (Barr et al.)[1]. Rewriting the floating-point operations into rational operations will solve the problem.

5.6 Floating-point accuracy problem

Besides floating-point exceptions, another floating-point problem is floating-point accuracy, which happens when the program does not crash but the result is inaccurate. Even though it may not be significant at the first place, it can lead to big error after further accumulation. The basic floating-point accuracy problems are insufficient precision, rounding errors and catastrophic cancellation.

Static analysis can be done for analyzing floating-point accuracy. Two basic approaches are interval arithmetic and affine arithmetic. Tools like Gappa (interval arithmetic) and SmartFloat (affine arithmetic) are designed for this purpose. The advantage of using static analysis is that the properties can be proven. However, the disadvantage is large error bounds and it only works well in scalar analysis (Benz, Hildebrandt, and Hack)[9]. Also, Both interval arithmetic and affine arithmetic seems return overly pessimistic result in the implementation. In contrast, SMT solvers are more precise in measuring the floating-point error (Chiang et al)[10]. However, it also states that the SMT method of detecting floating-point exception (Barr et al)[1] it is not efficient in error estimation. This may be caused by the limited scalability of SMT method.

6. CONCLUSIONS

For polynomial constraint, the parity of maximum exponent determines whether the constraint has solution or not. When the maximum exponent is odd, the value of the constraint spans the entire real number space and is always

satisfiable. However, when the maximum exponent is even, special treatments are needed to deal with satisfiability and more than one free variables may be needed to solve the constraint. For a better coverage of satisfiability, a better concretization algorithm is needed. Also, future work should be done for solving more complex constraints.

7. CONTRIBUTIONS

Gene Der Su: Main programmer in Matlab. Testing the code. Main writer for motivating example, technical approach and evaluation part. Jian (Kevin) Xu: Testing the code and help algorithm design. Main writer for related work part. Report Formatting.

8. REFERENCES

[1] Barr, Earl T. et al. *Automatic Detection of Floating-Point Exceptions*, ACM SIGPLAN Notices. Vol. 48. ACM, 2013. 549-560. Google Scholar. Web. 20 Apr. 2015.

[2] Borralleras, Cristina et al. *Solving Non-Linear Polynomial Arithmetic via SAT modulo Linear Arithmetic*, Automated Deduction-CADE-22. Springer, 2009. 294-305. Google Scholar. Web. 10 May 2015.

[3] De Moura, Leonardo, and Nikolaj Bjørner. *Satisfiability modulo Theories: Introduction and Applications.*, Communications of the ACM 54.9 (2011): 69. CrossRef. Web. 7 May 2015.

[4] De Moura, Leonardo, and Nikolaj Bjørner. *CORAL: Solving Complex Constraints for Symbolic Pathfinder.*, NASA Formal Methods. Springer, 2011. 359-374. Google Scholar. Web. 26 May 2015.

[5] Goldberg, David. *What Every Computer Scientist Should Know about Floating-Point Arithmetic*, ACM Computing Surveys (CSUR) 23.1 (1991): 5-48. Print.

[6] Monniaux, David. *A Quantifier Elimination Algorithm for Linear Real Arithmetic.*, Logic for Programming, Artificial Intelligence, and Reasoning. Springer, 2008. 243-257. Google Scholar. Web. 10 May 2015.

[7] Scott, Alexander D., and Gregory B. Sorkin. *Polynomial Constraint Satisfaction Problems, Graph Bisection, and the Ising Partition Function.*, ACM Transactions on Algorithms (TALG) 5.4 (2009): 45. Print.

[8] Vu, Xuan-Ha, Hermann Schichl, and Djamila Sam-Haroud. *Interval Propagation and Search on Directed Acyclic Graphs for Numerical Constraint Solving.*, Journal of Global Optimization 45.4 (2009): 499?531. Print.

[9] Benz, Florian, Andreas Hildebrandt, and Sebastian Hack. *A Dynamic Program Analysis to Find Floating-Point Accuracy Problems.*, ACM SIGPLAN Notices. Vol. 47. ACM, 2012. 453?462. Google Scholar. Web. 20 Apr. 2015.

[10] Chiang, Wei-Fan et al. *Efficient Search for Inputs Causing High Floating-Point Errors.*, ACM Press, 2014. 43-52. CrossRef. Web. 20 Apr. 2015.