Precisely Tracking Floating-Point Error with a Step-Function Abstract Domain

a Directed Research Project

Anthony Dario

March 11, 2024

Abstract

We present a new abstract domain for analyzing the floating-point error accrued during a computation. We model the floating-point error as a step function over an interval of possible values a variable can take. This model accurately mimics the discrete nature of floating-point error by taking into account the specific properties of floating-point numbers for closer approximations of the error. We discuss the application of this domain to some specific properties and how it allows for tighter fixpoint approximation. We implement a tool utilizing the step function domain that can analyze functions written in a subset of C. We show promising experimental results that improve on state-of-the art tools for certain output values.

1 Introduction

Floating point numbers are a ubiquitous, imperfect representation of the real numbers. This imperfection causes the results of a floating-point computation to diverge from the infinite-precision, exact result. The deviation of a floating-point program from an exact result can cause internal issues such as branch instability or incorrect calculations which can lead to significant issues. Understanding the potential round-off error a program can incur allows programmers to foresee and fix these issues.

An example of real-world impact caused by floating-point error is the patriot missile system bug that failed to intercept a scud missile [24]. The system's clock counted in tenths of seconds stored as an integer. During calculations the system converted the clock time to floating-point by multiplying by the unrepresentable 0.1. This caused an accumulation of rounding errors making the system unable to track an incoming missile and caused the deaths of 28 people.

Dynamic approaches to tracking floating-point error involve calculating the error of a variable alongside an execution [3, 10]. These techniques are effective at detecting error but they incur a significant runtime overhead; slowing execution by a factor of 30 or worse. Dynamic techniques are also unable to state an

upper bound on the error for *all* executions leaving the potential for unknown cases to incur significantly more error. For applications where the cost of a bug is large this is unacceptable.

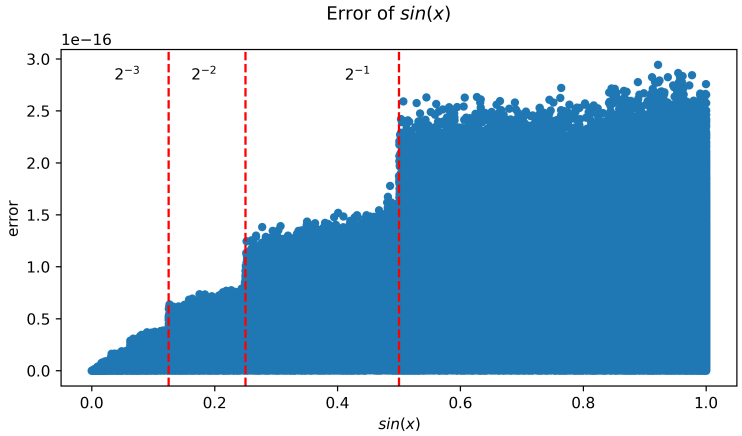
Many static approaches to analyzing floating-point error accumulation have been developed. Techniques that symbolically transform the program and then optimize for the minimum sound error produce tight error bounds but do not work with control flow [16, 25]. Libraries built with proof assistants can provide very tight error bounds but require manual steps to link a functional model to the program [9, 15]. The above techniques provide precise analysis but all require manually specifying the program in either a domain-specific language or a proof assistant. This manual step requires specific training and can introduce discrepancies between the program and the specification. Abstract Interpretation-based techniques can handle control flow and are more automated, however, their analyses are shown to be less precise than the previous approaches [6, 12].

Embedded systems are a particularly suitable domain for static analysis. Embedded systems are often used in safety-critical computers such medical devices and avionics where software errors can have dangerous consequences. Embedded software must conform to the capabilities of the hardware it is running on. Often, embedded software cannot use standard libraries which have been the focus of intensive previous analysis. Functionality is reimplemented requiring a new analysis. Performing the analysis manually adds a significant overhead to the development time for embedded systems. Automatic analyses allow non-specialists to run the analysis, providing a quicker feedback loop and less effort during development.

We base our approach on abstract interpretation as the analysis does not require translating programs into a specification language and can handle control flow. We ameliorate the precision issue with a novel abstract domain that mimics the discrete nature of floating-point numbers. We model the floating point error as a piece-wise step function over possible values a number can take during runtime. This step-function model follows the behavior of floating point error such as how the worst-case rounding error is proportional to the magnitude of the number.

Figure 1 shows the discrete nature of floating-point rounding error for a sine approximation. The error (y axis) increases as the output of the function (x axis) increases. Significant increases in error are seen whenever the output crosses a power of 2. We use the sine approximation from the FPBench community benchmarks [7].

This approach allows us to take into account particular properties of floating-point numbers such as the well-known Sterbenz's theorem, and accurately give rounding errors for values that can cross a binade. The domain also allows for tight fixpoint analysis of loops through the definition of widening and narrowing operators.




```
double sin_dbl(double x) {
    return (x - (x*x*x) / 6.0 +
            (x*x*x*x*x) / 120.0 -
            (x*x*x*x*x*x*x) / 5040.0;
}
```

Figure 1: Floating-Point error for an approximation of $\sin(x)$ from FPBench [7]

The contributions of this paper are:

- The definition of a new step-function abstract domain.
- Analysis of fixpoint convergence for the abstract domain.
- Incorporation of specific floating-point properties into the analysis.
- An implementation of a static analyzer based on the abstract domain.
- Experimental evaluations comparing the accuracy this domain to other abstract-analysis-based tools.

Section 2 gives some necessary background on floating-point numbers. Section 3 defines the language under analysis and the concrete reachability semantics. Section 4 defines abstract domain and abstract semantics. Section 5 incorporates special properties of floating-point numbers into the analysis. Section 6 defines the lattice over the abstract domain and shows the Galois connection demonstrating the soundness of the analysis. Sections 7 and 8 describe an implementation and empirical comparisons to other tools. Section 9 discusses other static analysis technique for analyzing floating-point error. Finally, Sections 10 and 11 conclude with future research directions.

2 Background

Floating-point error arises from the inability of the format to represent all reals in its domain. Floating-point numbers are represented by: $m\beta^e$ where e is the integer exponent, and m is the significand represented in radix β (typically 2 in computers) [20]. As there can be multiple representations of a floating-point number, the *canonical representation* is the representation with the smallest exponent. The *precision* of a format refers to the number of significant digits of the significand. A format is defined by its maximum and minimum exponents, radix, and precision. When a value cannot be represented in a format it must be rounded to one that can be represented. The IEEE-754 standard [1] defines a few different rounding schemes. In practice, the most common is round-to-nearest but a programmer can specify if they wish to always round up, down, or towards zero. This analysis focuses on the round-to-nearest rounding mode but it can easily be extended to work with other rounding modes.

Rounding error can be bounded by the "unit in the last place" function, $ulp(r)$. The ulp function represents the distance between the two floating point numbers nearest to r [19]. It is the difference when the mantissa is incremented or decremented. Regardless of the rounding mode, the introduced error can always be bounded using the ulp function: $|r - R(F(r))| \leq ulp(r)$ where $R : \mathbb{R} \to \mathbb{F}$ and $F : \mathbb{R} \to \mathbb{F}$ convert between the real and floating-point numbers.

The ulp function's output depends on the input's exponent value. As the exponent increases each increment of the mantissa will produce a larger change in value. The set of values that are canonically representable with the same exponent is called a *binade*. In a radix 2 format, binade boundaries lie on powers of 2. The effect of the binade's exponent on the error are what produce the step function behavior seen in Figure 1. Each significant increase in error occurs on the boundary of a binade.

3 Language and Concrete Semantics

Abstract interpretation [4, 5, 23] is a framework for static analysis that relies on defining an abstract domain to overapproximate all possible states of the program. In this section we define a model of the computation that contains the properties we are interested in, referred to as the *concrete reachability semantics*. In Section 4 we will define the abstract domain and semantics which overapproximate the reachability semantics. From the abstract semantics an abstract interpreter can be built that provides bounds on the desired properties.

The concrete reachability semantics define the precise set of states a program can reach. These semantics take in a set of possible program states and return the exact set of possible states after running the program. The semantics defined below are not automatically computable but serve as a model for the abstract domain to overapproximate. The syntax an concrete semantics are defined in Figure 2 and will be described in detail below.

We are analyzing a small imperative language based on the C language shown

$$A ::= x \mid f \mid i \mid A + A \mid A - A \mid A * A \mid A/A$$
$$B ::= \mathit{true} \mid \mathit{false} \mid !B \mid A \leq A \mid A < A \mid A == A \mid A > A \mid A \geq A$$
$$S ::= x = A \mid S; S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{for}(S; B; S)\{S\}$$

(a) Syntax

$$\begin{aligned}
&[\![A]\!] : \mathbb{M} \to \mathbb{V} \\
&[\![x]\!]m && = m\ x \\
&[\![f]\!]m && = (f, 0) \\
&[\![i]\!]m && = i \\
&[\![A_1 \odot A_2]\!]m && = [\![A_1]\!]m \odot [\![A_2]\!]m
\end{aligned}$$

$$\begin{aligned}
&[\![B]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M}) \\
&[\![\mathit{true}]\!]M && = M \\
&[\![\mathit{false}]\!]M && = \emptyset \\
&[\![!B]\!]M && = \{m \mid m \in M \wedge m \notin [\![B]\!]M\} \\
&[\![A_1 \lessgtr A_2]\!]M && = \{m \mid m \in M \wedge [\![A_1]\!]m \lessgtr [\![A_2]\!]m\}
\end{aligned}$$

(b) Concrete reachability semantics of expressions

$$\begin{aligned}
&[\![S]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M}) \\
&[\![x = A]\!]M && = \{m[x \mapsto [\![A]\!]m] \mid m \in M\} \\
&[\![S_1; S_2]\!]M && = [\![S_2]\!]([\![S_1]\!]M) \\
&[\![\text{if } B \text{ then } S_1 \text{ else } S_2]\!]M && = [\![S_1]\!]([\![B]\!]M) \cup [\![S_2]\!]([\![!B]\!]M) \\
&[\![\text{for } (S_1; B; S_2)\{S_3\}]\!]M && = [\![!B]\!] \left[\bigcup_{i \to \infty} ([\![S_2]\!] \circ [\![S_3]\!] \circ [\![B]\!])^i ([\![S_1]\!]M) \right]
\end{aligned}$$

(c) Concrete reachability semantics of statements

Figure 2: Syntax and Reachability Semantics

in Figure 2a. The language has variables x, double-precision floating-point values f, integer values, i, and arithmetic expressions A. We include integer values as they are commonly used in loop conditions. Boolean expressions B consist of *true*, *false*, numerical comparisons, and negation $!B$. Statements S allow for assignment, composition using semicolons, and control flow with for-loops and if-statements. For-loops work as in C with the first S being the initialization statement, B being the loop condition, the second S being the increment statement, and the final S being the loop body.

The concrete reachability semantics of expressions are given in Figure 2b. The reachability semantics model the exact set of all possible states a program can be in. Program states are modeled as memory $\mathbb{M} : \mathbb{X} \to \mathbb{V}$ which maps variable names, $\mathbb{X}$, to values $\mathbb{V}$. Values may be integer and double-precision floating-point values. Floating-point values are modeled as a tuple (f, ϵ) where

f is the floating-point value and ϵ is the rounding error of that value as a positive real number. The semantics of arithmetic expressions $[\![A]\!]$ returns the integer or floating-point value (with error incurred) $\mathbb{V} = (\mathbb{F} \times \mathbb{R}) \cup \mathbb{Z}$ of the expression given a specific program state. For brevity the set of arithmetic operations are written as $\odot \in \{+, -, *, /\}$ and boolean operations are written as $\lessgtr \in \{\leq, <, ==, >, \geq\}$. The language only considers double-precision floating-point numbers but the analysis can easily be extended to include single-precision floating-point numbers as well.

The semantics of boolean expressions $[\![B]\!]$ takes as input a set of possible program states and returns the subset of those states that satisfy the condition in the expression. In this way the semantics of boolean expressions acts as a filter that can be used during control flow to ensure only the appropriate states are evaluated in a particular branch. For *true* this is all input states and for *false* this is the empty set. Negation ($!B$) includes all states in the input that do not satisfy the condition. Comparisons ($A_1 \lessgtr A_2$) filter the input set by states that can evaluate the comparison to true.

Figure 2c shows the concrete reachability semantics of statements $[\![S]\!]$. Assignment ($x = A$) updates the variable x in every state. Composition ($S; S$) gives the possible states after executing the first statement then the second statement. For if-statements the input set of states may contain states that satisfy the condition and others that do not. The semantics filters the states using the semantics of boolean expressions on the condition and the condition's negation. Then, the appropriate branch is explored and both sets are combined with a union. For loops we first calculate the possible states after the initialization statement S_1, then filter by the loop condition and evaluate the loop body. The loop body will iterate a different number of times depending on the input state. We calculate the output states by taking the union of the loop body , filtered by the loop condition, for an infinite number of iterations. Finally the states are filtered by the negation of the loop condition to model exiting the loop.

The concrete reachability semantics define the exact set of states that a program can reach after executing. They capture the floating-point values and error of variables. The floating-point error propagation for the concrete semantics is left undefined as, by Rice's theorem [4], these semantics cannot be computed automatically. To provide automatic analysis we must approximate these concrete semantics through a more amenable abstraction.

4 Abstract Domain and Semantics

To automatically compute the possible output states of a program, we abstract the program states and provide a semantics for the abstraction that are computable. The abstraction overapproximates the set of all reachable states in real-world executions. It is important that the overapproximation is sound, that is, it conservatively captures *all* reachable states, while possibly including some unreachable ones as well. Without soundness the analysis could underapprox-

imate the error and becomes unreliable. The abstraction of program memory relies on *abstract memory* $\mathbb{M}^\sharp : \mathbb{X} \to \mathbb{V}^\sharp$ which maps variables to abstract values in an abstract domain. These abstract values describe the value of the variable for sets of possible states in the concrete semantics. Section 4.1 describes the abstract domain and Section 4.2 describes the abstract semantics.

4.1 Abstract Domain

We track floating-point error alongside a variable's possible values in a tuple called a *segment*:

$$\mathbb{S} = (\mathbb{I} \times \mathbb{E}).$$

The first element of the tuple, $\mathbb{I}$ is an interval that contains all possible values the variable can take. The second element, $\mathbb{E}$ is an overapproximation of the floating-point error represented as a positive float. Including the variable's possible values allows our error tracking to take into account the magnitude of the variable as well as certain properties of floating-point numbers as it will be explained in Section 5.

The analysis is improved by splitting the interval of values into multiple segments each with its associated error. This is because the maximum rounding error increases by a factor of two between binades. The multiple segments can be seen as a piecewise-step function of the error, with each segment being a separate "step" of the function. We define the step function domain as:

$$\mathbb{P} : \mathcal{P}(\mathbb{S})$$

Before computing the abstract semantics of a program, first the program's values must be converted to appropriate values in the abstract domain. We define an abstraction function $\alpha_\mathbb{F} : (\mathbb{F} \times \mathbb{R}) \to \mathbb{P}$ that maps floating-point constants (and associated error) to a single-segment step function. We form the interval by rounding the number down and up to the nearest representable numbers. The rounding error is overapproximated with the *ulp* function.

$$\alpha_\mathbb{F}(f, \epsilon) = ([F_\downarrow(f); F_\uparrow(f)], \frac{1}{2} ulp(f)) \tag{1}$$

where $F_\downarrow : \mathbb{F} \to \mathbb{F}$ and $F_\uparrow : \mathbb{F} \to \mathbb{F}$ convert a real value to a float rounding down and up respectively. Parameters have unknown values so their range of values is specified by the user.

Integers are also included as values in our language so we must abstract those as well. We define another abstraction function $\alpha_\mathbb{Z} : \mathbb{Z} \to \mathbb{I}$ that maps integers to the domain of intervals $\mathbb{I}$.

$$\alpha_\mathbb{Z}(i) = [i; i] \tag{2}$$

The abstract domain is then:

$$\mathbb{V}^\sharp : \mathbb{P} \cup \mathbb{I}$$

With the program values converted to the abstract domain we proceed to define the abstract semantics.

$$
\begin{array}{ll}
[\![A]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{V}^\sharp & \\
[\![x]\!]^\sharp M^\sharp & = M^\sharp\ x \\
[\![f]\!]^\sharp M^\sharp & = \alpha_{\mathbb{F}}(f) \\
[\![i]\!]^\sharp M^\sharp & = \alpha_{\mathbb{Z}}(i) \\
[\![A_1 \odot A_2]\!]^\sharp m^\sharp & = [\![A_1]\!]^\sharp M^\sharp \odot_{\mathbb{P}} [\![A_2]\!]^\sharp M^\sharp
\end{array}
$$

$$
\begin{array}{ll}
[\![B]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp & \\
[\![true]\!]^\sharp M^\sharp & = M^\sharp \\
[\![false]\!]^\sharp M^\sharp & = \bot_{\mathbb{M}^\sharp} \\
[\![!B]\!]^\sharp M^\sharp & = M^\sharp[x \mapsto \mathcal{F}_{\neg(A_1 \lessgtr\ A_2)}(x) \mid x \in \mathbb{M}^\sharp] \\
[\![A_1 \lessgtr A_2]\!]^\sharp M^\sharp & = M^\sharp[x \mapsto \mathcal{F}_{(A_1 \lessgtr\ A_2)}(x) \mid x \in \mathbb{M}^\sharp]
\end{array}
$$

(a) Abstract semantics of expressions

$$
\begin{array}{ll}
[\![S]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp & \\
[\![x = A]\!]^\sharp M^\sharp & = M^\sharp[x \mapsto [\![A]\!]^\sharp M^\sharp] \\
[\![S_1; S_2]\!]^\sharp M^\sharp & = [\![S_2]\!]^\sharp([\![S_1]\!]^\sharp M^\sharp) \\
[\![\text{if } B \text{ then } S_1 \text{ else } S_2]\!]^\sharp M^\sharp & = [\![S_1]\!]^\sharp([\![B]\!]^\sharp M^\sharp) \cup_{\mathbb{M}^\sharp} [\![S_2]\!]^\sharp([\![!B]\!]^\sharp M^\sharp) \\
[\![\text{for } (S_1; B; S_2)\{S_3\}]\!]^\sharp M^\sharp & = [\![!B]\!]^\sharp \left[\bigcup_{i \to \infty} ([\![S_2]\!]^\sharp \circ [\![S_3]\!]^\sharp \circ [\![B]\!]^\sharp)^i([\![S_1]\!]^\sharp M^\sharp) \right]
\end{array}
$$

(b) Abstract semantics of statements

Figure 3: Abstract Semantics

4.2 Abstract Semantics

Figure 3 defines the abstract semantics $[\![\cdot]\!]^\sharp$ over the same syntax as in Figure 2a. The abstract semantics are similar to the concrete with the main difference being the change in types. The semantics of arithmetic operations now produces an abstract value from an abstract memory. The semantics of boolean expressions filters abstract memory rather than a set of states. The semantics of statements produces a new abstract memory that approximates the effect of the statement on the input abstract memory. We will describe the semantics of each of these syntactic forms in turn.

4.2.1 Arithmetic Expressions

The main difference between the concrete and abstract semantics for arithmetic expressions is the change in values. This requires us to abstract constants and define new operators for our abstract domain $\odot_{\mathbb{P}}$. For arithmetic operators over the interval abstraction of integers, interval arithmetic [13, 17] suffices. When mixing intervals and step functions the step function is "cast" to an interval by taking the union of all of its segment's intervals. After the cast we apply

$$
\begin{aligned}
err_+(i_1, e_1, i_2, e_2) &= e_1 + e_2 2 + \frac{1}{2} ulp(\uparrow i_1 + \uparrow i_2) \\
err_-(i_1, e_1, i_2, e_2) &= |e_1 + e_2| + \frac{1}{2} ulp(\uparrow i_1 + \uparrow i_2) \\
err_*(i_1, e_2, i_2, e_2) &= \uparrow i_1 e_2 + \uparrow i_2 e_1 + e_1 e_2 + \frac{1}{2} ulp((\uparrow i_1) * (\uparrow i_2)) \\
err_/(i_1, e_2, i_2, e_1) &= \frac{(\uparrow i_1) e_2 + (\downarrow i_2) e_1}{(\downarrow i_2)^2 - (\downarrow i_2) e_2} + \frac{1}{2} ulp(\uparrow i_1 / \downarrow i_2)
\end{aligned}
$$

Figure 4: Operator Error Propagation

interval arithmetic. The rest of this section will focus on arithmetic operators over step functions.

The general idea for abstract arithmetic operators is to apply the operator to each pair of segments from both values and then *merge* any overlapping segments.

$$X \odot_{\mathbb{P}} Y = merge_{\mathbb{P}}(\{x \odot_{\mathbb{S}} y \mid \forall x \in X, y \in Y\}) \tag{3}$$

Where $\odot_{\mathbb{P}}$ is an arithmetic operator for step-functions and $\odot_{\mathbb{S}}$ is an arithmetic operator for segments.

We define segment operators by using interval arithmetic on the underlying intervals while propagating error using an error function.

$$(i_1, e_1) \odot_{\mathbb{S}} (i_2, e_2) = (i_1 \odot_{\mathbb{I}} i_2, err_{\odot}(i_1, e_1, i_2, e_2))$$

Where $\odot_{\mathbb{I}}$ is the arithmetic operator for intervals.

To define the functions that propagate existing errors we modify the analysis in [27] to work on segments. The error of the result comes from two sources: the propagation of the error from the two operands, and the rounding error from an unrepresentable result. When overapproximating these sources of error it is important to pick the values in the interval that produce the largest error. To this end, it is convenient to define functions $\uparrow\colon \mathbb{I} \to \mathbb{F}$ and $\downarrow\colon \mathbb{I} \to \mathbb{F}$ which select the value in the interval with the largest and smallest magnitude respectively.

$$
\begin{aligned}
\uparrow [l; u] &= \max(|l|, |u|) \\
\downarrow [l; u] &= \begin{cases} 0 & \text{if } l < 0 < u \\ \min(|l|, |u|) & \text{otherwise} \end{cases}
\end{aligned}
$$

Where l and u are the lower and upper bound of the input interval.

Rounding error is be bounded by the *ulp* function. As we are focused on the rounding-to-nearest rounding mode we need to find $\frac{1}{2} ulp(x)$ where x is the resulting value with the largest magnitude. If we were concerned with other rounding modes we could bound the error with $ulp(x)$ instead. x is acquired by selectively using the bound on the operand intervals with the largest (or

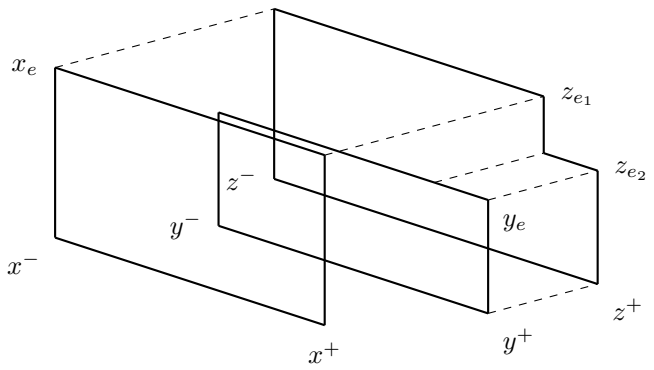


Figure 5: A Merge Operation

smallest) magnitude. As an example, for multiplication we select the largest magnitude boundary of both variables to multiply together giving us a rounding error of $\frac{1}{2}ulp(\uparrow i_1* \uparrow i_2)$ where i_1 and i_2 are the intervals being multiplied. Note that a preexisting error term does not need to be involved as the interval analysis is already sound, so the value cannot lie outside the interval. In the case of division if the interval contains 0 then the error is unbounded and the `NaN` value is assigned to the expression.

To propagate existing errors through an operation we notice that two values, with associated errors, will produce a third value with a new error. By selecting values with the maximum magnitude we bound the expression and solve for the error of the result.

In the example of addition, we know that $(i_1, e_1) + (i_2, e_2) = (i_3, e_3)$. We select any two arbitrary values $x_1 \in i_1$ and $x_2 \in i_2$ and they will produce $x_1 + x_2 = x_3 \in i_3$. By adding the errors to these values we get $(x_1 + e_1) + (x_2 + e_2) = (x_3 + e_3)$. We cancel out the xs to end up with $e_1 + e_2 = e_3$. Similar reasoning can be used for the rest of the operations. The error functions are listed in Figure 4.

When performing the arithmetic operations on step functions defined in Equation 3, output segments may overlap. As an example, if

$$X = \{([2;4], e_{x1}), ([4;8], e_{x2})\} \qquad Y = \{([1;3], e_y)\}$$

then

$$X -_{\mathbb{P}} Y = \{([-1;3], err_-(x_1, y_1)), ([1;7], err_-(x_2, y_1))\}$$

where x_i and y_i is the ith element of X or Y respectively. In this example,the output has two segments with intervals that overlap between 1 and 3.

To remove this redundancy we perform a $merge_{\mathbb{S}} : \mathbb{S} \times \mathbb{S} \to \mathbb{P}$ operation by creating a set consisting of the segment with larger error and any non-

overlapping portions of the segment with lower error.

$$merge_{\mathbb{S}}(s_1, s_2) = \begin{cases} \{s_1\} \cup s_2/s_1 & \text{if } err(s_1) > err(s_2) \\ \{s_2\} \cup s_1/s_2 & \text{otherwise} \end{cases}$$

Here, $err(s)$ projects the error from the segment and $s_2/s_1 : \mathbb{S} \to \mathcal{P}(\mathbb{S})$ returns all (possibly discontinuous) portions of s_2 that don't overlap with s_1.

Figure 5 illustrates the merge operation with two segments x and y being merged into a step function z.

The merge operation is naturally be lifted to operating on a step function, $merge_{\mathbb{P}} : \mathbb{P} \to \mathbb{P}$, by merging all overlapping segments.

$$merge_{\mathbb{P}}(sf) = \{merge_{\mathbb{S}}(s_1, s_2) \mid \forall s_1, s_2 \in sf \wedge overlap(s_1, s_2)\}$$

The *overlap* predicate is true if the intervals of s_1 and s_2 overlap.

With the arithmetic operators of step functions and the abstraction function defined the abstract semantics are in place for arithmetic expressions.

4.2.2 Boolean Expressions

The abstract semantics of boolean expressions operate on a single abstract memory instead of a set of memory states as in the concrete semantics. The semantics restrict the values in the abstract memory to values that satisfy the condition. For *true* we return the input memory and for *false* we return the empty memory $\bot_{\mathbb{M}^\sharp}$ which maps every variable to the empty set.

Comparisons involve the $\mathcal{F}_B : \mathbb{P} \to \mathbb{P}$ function which removes any section of the domain of its input that cannot satisfy the comparison B. As an example, for $X \leq Y$, X is limited to the set of segments that are less than Y's upper bound, written Y^+, and any segments that cross Y's upper bound are now bounded by Y^+:

$$\begin{aligned} \mathcal{F}_{X \leq Y}(X) = &\{s \mid s \in X \wedge s^+ \leq Y^+\} \cup \\ &\{([s^-; Y^+], err(s)) \mid s \in X \wedge s^- \leq Y^+ \wedge s^+ > Y^+\} \end{aligned}$$

When using strict inequalities the upper bound of the domain becomes $Y^+ - ulp(Y^+)$:

$$\begin{aligned} \mathcal{F}_{X < Y}(X) = &\{s \mid s \in X \wedge s^+ < Y^+\} \cup \\ &\{([s^-; Y^+ - ulp(Y^+)], err(s)) \mid s \in X \wedge s^- < Y^+ \wedge s^+ \geq Y^+\} \end{aligned}$$

For negation we negate the condition we are filtering on. Comparing step functions to constants is similar.

4.2.3 Statements

The main difference between the abstract semantics of statements and the concrete semantics is the union operator now must be defined over abstract memory.

We also must handle the problem of calculating the infinite union found in the for loop semantics. Assignment is slightly different and now only updates the single abstract memory instead of all input states.

To calculate the abstract semantics of if-statements we must take the union of both branches. The union of abstract memory is the union of all their values.

$$M_1^\sharp \cup_{\mathbb{M}^\sharp} M_2^\sharp = [x \mapsto M_1^\sharp x \cup_{\mathbb{V}^\sharp} M_2^\sharp x \mid \forall x]$$

For the union of abstract values we define the union of two step functions is the union of all their segments merged. The union of intervals is defined in the standard way as the interval that contains both operands:

$$X \cup_{\mathbb{P}} Y = merge_{\mathbb{P}}(X \cup Y)$$
$$[a^-; a^+] \cup_{\mathbb{I}} [b^-; b^+] = [\min(a^-, b^-); \max(a^+; b^+)]$$

Calculating the infinite union in the for-loop body relies on finding a fixpoint of the semantic function for the body $\overline{f}(i) = (\llbracket S_2 \rrbracket^\sharp \circ \llbracket S_3 \rrbracket^\sharp \circ \llbracket B \rrbracket^\sharp)(i) \cup\ i$. To do this we iterate the semantic function of the loop body $\overline{f}_0(i) = i, \overline{f}_{n+1}(i) = \overline{f}(\overline{f}_n(i))$. If the iteration stabilizes, $\overline{f}_n(i) = \overline{f}_{n+1}(i)$, then we have found the least fixed point. If the function $\overline{f}$ is monotone over a lattice then a least fixed point exists by Kleene's Fixed-Point Theorem. In our case the lattice is over abstract memory as expanded on in Section 6.

In practice, the step function lattice is large, and finding the fixpoint through iteration is slow. A common strategy in abstract interpretation for quickly converging on a fixpoint is to extrapolate the iterates to a coarse approximation, and then refine through interpolation. Extrapolation is achieved through *widening*. Widening identifies any constraints that are unstable and approximates them with their extreme case. Interpolation is achieved through *narrowing*. Narrowing attempts to replace the approximation of unstable constraints through downward iteration. Both widening and narrowing operators are defined on the step function domain.

4.2.4 Widening

To speed up the fixpoint convergence a widening operator is defined. Rather than taking the union of subsequent iterates of the transfer function we apply a widening operator to extrapolate any unstable constraints on the approximation.

As an example, for the error domain the widening operator ($\nabla_{\mathbb{E}}$) is defined as:

$$e_1 \nabla_{\mathbb{E}} e_2 = (\!|e_1 \leq e_2\ ?\ e_1 : \infty|\!)$$

The expression $(\!|b\ ?\ x : y|\!)$ acts as a ternary operator and takes value x if b is true and value y if b is false. e_1 is the current iterate and e_2 is the next iterate. If the bound on the error is increasing then the operator extrapolates the error bound to infinity.

For step functions the unstable constraints can be the upper or lower bounds of the domain or the error of any segment. If the bounds on the domain are expanding then we add a new segment to the bottom or top of the interval that extrapolates this instability. The new segment pushes the bounds to $-\infty$ or ∞ depending on the bound and has infinite error. If any of the existing segments has an unstable error bound then the error bound is pushed to infinity as well.

Definition 1 *Step Function Widening*
The widening operator for step functions $\nabla_{\mathbb{P}} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ *is defined as:*

$$\begin{aligned} sf_1 \nabla_{\mathbb{P}} sf_2 = {} & (\!|(\bigcup_{\mathbb{I}} sf_1)^- \geq (\bigcup_{\mathbb{I}} sf_1)^- \ ? \ \emptyset : ([-\infty; (\bigcup_{\mathbb{I}} sf_2)^-], \infty)|\!) \cup \\ & (\!|(\bigcup_{\mathbb{I}} sf_1)^+ \leq (\bigcup_{\mathbb{I}} sf_1)^+ \ ? \ \emptyset : ([-\infty; (\bigcup_{\mathbb{I}} sf_2)^-], \infty)|\!) \cup \\ & \{(i_1, e1 \nabla_{\mathbb{E}} e_2) \mid \forall (i_1, e_1) \in sf_1, (i_2, e_2) \in sf_2 . \, overlap(i_1, i_2)\} \end{aligned}$$

We write $\bigcup_{\mathbb{I}} : \mathbb{P} \to \mathbb{I}$ for the union of all intervals in the step function, representing the domain of the step function, with $(\bigcup_{\mathbb{I}} sf)^-$ and $(\bigcup_{\mathbb{I}} sf)^+$ are the lower and upper bounds of the resulting interval respectively.

The first two terms in the union determine if the lower or upper bound on the domain is unstable and add a new segment pushing the bound to its extreme. The last term checks each segment to see if its error bound is unstable and pushes unstable bounds to infinite error.

4.2.5 Narrowing

Widening provides improved performance but the loss of precision is substantial. To recover precision, we can interpolate by narrowing. The narrowing operator attempts to improve bounds that have been pushed to infinity during widening by comparing the bounds to subsequent iterations of the loop body semantics. Continuing to iterate the loop body semantics has desirable effects because the loop condition filters portions of the domain.

For error, the narrowing operator $\Delta_{\mathbb{E}}$ will only modify error bounds that have been pushed to infinity:

$$e_1 \Delta_{\mathbb{E}} e_2 = (\!|e_1 = \infty \ ? \ e_2 : e_1|\!)$$

The narrowing operator for step functions attempts to replace any lower and upper bound segments widening produced as well as any unstable error bounds.

Definition 2 *Step Function Narrowing*
The narrowing operator for step functions $\Delta_{\mathbb{P}} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ *is defined as:*

$$
\begin{aligned}
&sf_1 \Delta_{\mathbb{P}} sf_2 = \\
&\quad (\!|(\underset{\mathbb{I}}{\cup} sf_1)^- = -\infty \ ? \ \downarrow_{\mathbb{P}} sf_2 : \emptyset|\!) \cup \\
&\quad (\!|(\underset{\mathbb{I}}{\cup} sf_1)^+ = \infty \ ? \ \uparrow_{\mathbb{P}} sf_2 : \emptyset|\!) \cup \\
&\quad \{(\!|err(s_1) = \infty \ ? \ (i_1, e_1 \Delta_{\mathbb{E}} e_2) : s_1 \mid \ \forall (i_1, e_1) \in sf_1.\ i_1^- \neq -\infty \wedge \ i_2^+ \neq \infty \wedge \\
&\quad (i_2, e_2) = \max(\{s \mid s \in sf_2, overlap(s, s1)\})\}
\end{aligned}
$$

Where $\uparrow_{\mathbb{P}}$ and $\downarrow_{\mathbb{P}}$ select the segments with the lowest and highest bounds on their domain respectively.

The first term examines the lower bound of the step functions domain for a widened bound $(\underset{\mathbb{I}}{\cup} sf_1)^- = -\infty$ and replaces it with the value of the next iteration. The second term does the same for the upper bound. The third term examines each segment in sf_1 for an extrapolated error bound and attempts to improve it with the subsequent iteration.

Widening and narrowing allow for the analysis to converge on a fixpoint for loops in a reasonable amount of time at the loss of precision.

With the abstract semantics in place for arithmetic expressions, boolean expressions, and statements we can provide overapproximations for the domain and error of floating-point values. Next, we will discuss how the analysis can be improved by taking advantage of specific properties of floating-point numbers.

5 Splitting Properties

Floating-point operations do not produce error uniformly. The rounding error is heavily dependent on the binade the result lands in. There are also special cases of operations that produce no error at all. The step function domain can handle these cases by creating new segments when it is possible to increase the accuracy of the error analysis. This section discusses techniques for improving the error analysis by taking into account these scenarios.

5.1 Binade Splitting

As discussed before, a floating-point numbers exponent has a significant impact on its rounding error. We can improve our analysis by splitting an operations result along binade boundaries into segments. As each binade has a different maximum rounding error, smaller binades can be given a more precise error bound.

For the analysis to take this into account we update the arithmetic operators on segments. The idea is to calculate the output interval, then split the interval along all binade boundaries. The resulting subintervals can then be mapped to output segments with error functions slightly modified from the operators

defined in Figure 4. First we define the $split : \mathbb{I} \to \mathcal{P}(\mathbb{I})$ function.

$$
\begin{aligned}
&split([a^-; a^+]) = \\
&\quad \{[2^n; 2^{n+1} - ulp(2^{n+1})] \mid n \in \mathbb{Z} \ \wedge \ a^- < 2^n \ \wedge \ a^+ > 2^n + 1\} \cup \\
&\quad \{[a^-; 2^n - ulp(2^n)] \mid n = exp(a^-)\} \cup \\
&\quad \{[2^n; a^+] \mid n = exp(a^+)\}
\end{aligned}
$$

Here $exp(f) : \mathbb{F} \to \mathbb{Z}$ gives the exponent of the floating point value f. The first term finds all subintervals that span an entire binade. The second and third terms are intervals formed from the lower and upper bounds of input interval and their nearest binade boundary.

To propagate error to these split intervals we slightly modify the error operators in Figure 4. The propagation component stays the same but now we can improve the bound on the rounding error of the operation by taking the upper bound of the output interval, i_o.

$$
\begin{aligned}
err_+(i_1, e_1, i_2, e_2, i_o) &= e_1 + e_2 2 + \frac{1}{2} ulp(\uparrow i_o) \\
err_-(i_1, e_1, i_2, e_2, i_o) &= |e_1 + e_2| + \frac{1}{2} ulp(\uparrow i_o) \\
err_*(i_1, e_2, i_2, e_2, i_o) &= \uparrow i_1 e_2 + \uparrow i_2 e_1 + e_1 e_2 + \frac{1}{2} ulp(\uparrow i_o) \\
err_/(i_1, e_2, i_2, e_1, i_o) &= \frac{(\uparrow i_1) e_2 + (\downarrow i_2) e_1}{(\downarrow i_2)^2 - (\downarrow i_2) e_2} + \frac{1}{2} ulp(\uparrow i_o)
\end{aligned}
$$

Now we update the arithmetic operators on segments to take into account this splitting.

$$(i_1, e_1) \odot_{\mathbb{S}} (i_2, e_2) = \{(i, err_\odot(i_1, e_1, i_2, e_2, i)) \mid i \in split(i_1 \odot_{\mathbb{I}} i_2)\}$$

The updated segment arithmetic operators are then used in Equation 3 to provide an improved bound on the error. While taking advantage of the rounding error of binades was the main motivation for splitting the values domain, splitting the domain is also used to take advantage of other properties of floating-point numbers.

5.2 Sterbenz's Lemma

An example of a special case is Sterbenz's Lemma:

Theorem 1 (Sterbenz [26]) $\forall x, y \in \mathbb{F}$ *with* $x, y \geq 0$

$$\frac{x}{2} \leq y \leq 2x \implies x - y \text{ is exact.}$$

This theorem states that floating-point subtraction of two positive floats x and y incurs no rounding error when y is within a certain interval of x. This theorem

can be extended [16] to account for negative numbers by changing the condition to:

$$\frac{x}{2} \leq y \leq 2x \text{ or } 2x \leq y \leq \frac{x}{2} \tag{4}$$

To take advantage of this property, any section of the domain where the condition holds can be treated as a separate segment. When performing the subtraction $x - y$ first the intervals in which the condition in Equation 4 holds are calculated using interval arithmetic on x. We define the $sbenz : \mathbb{I} \to \mathbb{I}$ function which produces the interval in which Sterbenz's lemma would hold.

$$\begin{aligned} sbenz(i) = [&\min\ (\uparrow i/[2;2], \downarrow i *_{\mathbb{I}} [2;2])\ ; \\ &\max\ (\uparrow i/[2;2], \downarrow i *_{\mathbb{I}} [2;2])] \end{aligned} \tag{5}$$

In the interval where Sterbenz's lemma holds we only consider error propagation, and not rounding error.

$$err_{sbenz}(e_1, e_2) = e_1 + e_2$$

We can now perform subtraction taking into account Sterbenz's lemma by splitting the right hand operand using Equation 5 then applying err_{-} and err_{sbenz} to the appropriate segments.

$$\begin{aligned} (i_1, e_1) -_{\mathbb{S}} (i_2, e_2) = &\{(i_1 -_{\mathbb{I}} i_s, err_{sbenz(e_1,e_2)}) | i_s = i_2 \cap_{\mathbb{I}} sbenz(i_1)\} \cup \\ &\{(i_1, e_1) -_{\mathbb{S}} (i, e_2) \mid \forall i \in (i_2 / sbenz(i_1))\} \end{aligned}$$

We write $i_1 \cap_{\mathbb{I}} i_2$ for the interval formed from the overlap of i_1 and i_2. The first term in the union performs the exact subtraction on the portion of the interval where Sterbenz's lemma holds. The second term performs a standard segment subtraction for all other portions.

6 Correctness

To ensure the abstract domain soundly approximates the concrete reachability semantics we show that the domain forms a lattice and that the abstraction function forms a Galois connection between the abstract and concrete domain.

The tuple $(\mathbb{P}, \sqsubseteq_{\mathbb{P}}, \sqcup_{\mathbb{P}}, \sqcap_{\mathbb{P}}, \{([-\infty; +\infty], +\infty)\}, \emptyset)$ forms a complete lattice over the domain of step functions. We define the order relation on the lattice as:

$$\begin{aligned} sf_1 \sqsubseteq_{\mathbb{P}} sf_2 \triangleq &\bigcup_{\mathbb{I}} sf_1 \sqsubseteq_{\mathbb{I}} \bigcup_{\mathbb{I}} sf_2 \wedge \\ &\forall s_1 \in sf_1, \forall s_2 \in sf_2.\ overlap(s_1, s_2) \implies err(s_1) < err(s_2) \end{aligned}$$

Where $\sqsubseteq_{\mathbb{I}}$ is the order on intervals based on inclusion. The first conjunct ensures that the domain of the lesser step function is contained in the domain of the greater. The second conjunct ensures that the error of the lesser step function is always lower than the error of the greater. The least upper bound $\sqcup_{\mathbb{P}} =$

$merge_{\mathbb{P}} \circ \cup$ is the merge operation after a union of the step functions. The greatest lower bound $\sqcap_{\mathbb{P}} = antimerge_{\mathbb{P}} \circ \cap$ is an "antimerge" operation after taking the intersection of the step functions. The antimerge operation is the same, as a merge except it prioritizes the segment with the lesser error.

$$antimerge_{\mathbb{S}}(s_1, s_2) = \begin{cases} \{s_1\} \cup s_2/s_1 & \text{if } err(s_1) < err(s_2) \\ \{s_2\} \cup s_1/s_2 & \text{otherwise} \end{cases}$$

$$antimerge_{\mathbb{P}}(sf) = \{antimerge_{\mathbb{S}}(s_1, s_2) \mid \forall s_1, s_2 \in sf \wedge overlap(s_1, s_2)\}$$

The top element is the single segment set containing the top interval and error $\{([-\infty; +\infty], +\infty)\}$ and the bottom element is the empty set.

To show that the abstract domain soundly overapproximates the concrete reachability semantics we describe the Galois connection formed by the abstraction function $\alpha_{\mathbb{P}}$ defined in Equation 1 and the *concretization* function $\gamma_{\mathbb{P}} : \mathbb{P} \to (\mathbb{F} \times \mathbb{R})$ converts a step function to the set of concrete values it describes.

The concretization function for step functions $\gamma_{\mathbb{P}} : \mathbb{P} \to \mathcal{P}(\mathbb{F} \times \mathbb{R})$ produces the set of all floating-point values and errors described by the step function. All segments in the step function are concretized to produce this set. The concretization of a segment $\gamma_{\mathbb{S}} : \mathbb{S} \to \mathcal{P}(\mathbb{F} \times \mathbb{R})$ is all tuples where the first element is a floating-point number contained in the interval and the error is less than the error of the segment. The concretization functions for intervals, segments, and step functions are defined below.

$$\begin{aligned} \gamma_{\mathbb{I}}([a^-; a^+]) &= \{f \mid a^- \leq f \leq a^\}\} \\ \gamma_{\mathbb{S}}((i, e)) &= \{(f, \epsilon) \mid f \in \gamma_{\mathbb{I}}(i), \epsilon < e\} \\ \gamma_{\mathbb{P}}(sf) &= \{(f, \epsilon) \mid \forall s \in sf.(f, \epsilon) \in \gamma_{\mathbb{S}}(s)\} \end{aligned}$$

As the concrete and abstract semantics are defined over memory and abstract memory, we show a connection between the different memories. We can abstract concrete memory by creating a function that abstracts the set of values a variable can take into a step function.

$$\alpha_{\mathbb{M}}(M) = \forall x \in \mathbb{X}\ x \mapsto \bigsqcup_{\mathbb{P}} \{\alpha_{\mathbb{P}}(mx) \mid \forall m \in M\}$$

Where $\alpha_{\mathbb{P}}$ selects the appropriate abstraction function for the type of the variable and the $\bigsqcup_{\mathbb{P}}$ is the least upper bound of the set of abstract values. The concretization of abstract memory forms the set of all program states that have variables approximated by the abstract memories abstract values:

$$\gamma_{\mathbb{M}^\sharp}(M^\sharp)) = \{m \mid \forall x \in \mathbb{X}, m \in M^\sharp.\ m\ x \in \gamma_{\mathbb{P}}(M^\sharp x)\}$$

The pair $(\alpha_{\mathbb{M}}(M), \gamma_{\mathbb{M}^\sharp})$ form a Galois connection $(\mathbb{M}, \subseteq) \underset{\alpha_{\mathbb{M}}}{\overset{\gamma_{\mathbb{M}^\sharp}}{\leftrightarrows}} (\mathbb{M}^\sharp, \sqsubseteq_{\mathbb{M}^\sharp})$. In the Galois connection concrete memory is ordered by the subset relation and abstract memory is ordered pointwise by each variable. That is, $M_1^\sharp \sqsubseteq M_2^\sharp$ if each variable in $M_2^\sharp$ is larger than each variable in $M_1^\sharp$.

```
x = {[0; 1.57079632679489656], 0.0}
```

(a) The specification file - `sine.spec`

```
double sine(double x) {
    return x - (x*x*x) / 6.0 +
           (x*x*x*x*x) / 120.0 -
           (x*x*x*x*x*x*x) / 5040.0;
}
```

(b) The code file - `sine.c`

```
var,type,low,high,err
y,flt,1.7192e-02,2.6568e-02,4.8992e-16
y,flt,-1.9073e-06,3.9062e-03,4.9078e-16
...
y,flt,1.1465e+00,1.1473e+00,1.14174e-15
y,flt,1.0619e+00,1.1465e+00,1.14217e-15
```

(c) The output file - `sine.csv`.
Numbers have been truncated to four significant digits

Figure 6: The input and output files when analyzing $\sin(x)$

The Galois connection between the abstract and concrete domain shows that the abstraction soundly overapproximates the concrete reachability semantics.

7 Implementation

We implemented the step function abstract domain in OCaml. The analyzer accepts a C source file, the name of a function to analyze, and a specification file containing preconditions for each parameter of the function. The analyzer will determine the range of values and the error for each variable declared in the specification file and the function. Figure 6 shows the input and output files from an analysis of a sine approximation.

The specification file is written by the user to provide bounds on the function parameters value and error. The file contains variable names and associated segments that provide the bounds. Variables are separated by a newline. The segment is written as `([lb ; ub], err)` where `lb` and `ub` are the lower and upper bound on the variables value and `err` is the upper bound on the error for the variable. Multiple segments can be included by enclosing them in curly braces and separating with commas. An example specification file is shown in Figure 6a. The input `x` ranges from 0 to $\pi/2$ with no error.

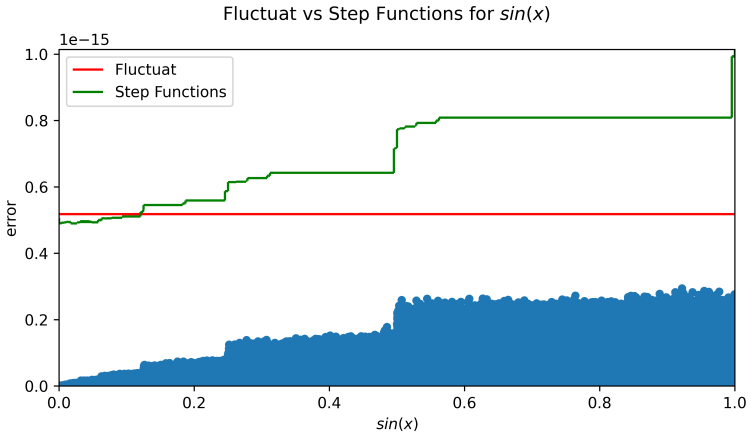


Figure 7: Step-functions vs Fluctuat on $sin(x)$

The C code is ingested using the CIL project [21], a tool to parse C into an intermediate language representation in OCaml. CIL's intermediate representation is then transformed to the internal representation described in Section 4. We chose CIL as it is a mature project that has been used as the basis for other static verification tools such as Frama-C. While C uses zero and nonzero values for true and false, these are converted to standard OCaml booleans during this transformation. The programs values are then abstracted to step functions using the abstraction functions 1 and 2 . OCaml does not natively support floating-point rounding modes so we defined C functions that allow for the specification of the rounding-mode and linked them using OCaml's foreign function interface.

Figure 6c shows the output file of an analysis. The output is a csv file where each row is a segment containing with the lower bound (`low`), upper bound (`high`), and error (`err`). The `var` column specifies which variable in memory the segment belongs to. The `type` column specifies if the variable is a floating-point value or an integer.

8 Evaluation

Preliminary experiments have been run to compare the analyzer to other comparable tools. We focus on automatic tools that can handle control flow. Currently we only compare against the abstract interpretation based static analyzer Fluctuat.

Comparison benchmarks are taken from FPBench [7] a suite of floating-point benchmarks contributed by the community. Note that these functions differ from the typical library functions one would find in a `math.h` implementation. In particular the sine function is a Taylor series approximation of sine [8]. Absolute error is calculated as the difference in result when using the `double` C type, with

a precision of 53 bits, and a 200 bit precision representation. Higher precision calculations are done using the multiple precision library MPFR [11]. 400,000 samples are taken to produce the absolute error.

Figure 7 compares Fluctuat with the step-function analyzer on FPBench's implementation of `sin(x)`. The y axis is the absolute floating-point error and the x axis is the output of the function. Each blue dot near the bottom of the graph represents the observed absolute error for some sample input of `x`. The green line shows the error of the segments of step-function analyzer and the red horizontal line shows the error produced by Fluctuat. The step function domain performs worse on larger outputs but nears Fluctuat's approximation as the output gets smaller eventually providing a more precise bound on the error.

The step function approach shows promise. While less precise for larger values the bounds are closer to reality for smaller values. The lack of precision for larger values is somewhat expected. The step-function abstract domain uses interval arithmetic to calculate the range of values a variable can be in. Intervals are less precise than Fluctuat's model of zonotopes. As a consequence the error must be overapproximated for a larger set of possible values. Possibilities for replacing interval arithmetic with a more precise value domain are discussed as a future research direction in Section 10.

9 Related Work

Many models for floating-point numbers have been proposed and implemented. Precisa [27], Astree [6], and Fluctuat [12] are tools that implement their models as abstract domains in the abstract interpretation framework. FPTaylor [25] and an approach by Lee et al. [16] symbolically change the floating-point numbers in a function to forms that include explicit error terms, then determine the maximum error by optimizing over the function parameters. PRECiSA, VCFloat2 [15], and Gappa [9] require the user to define the model and then provide tight error bounds automatically.

These models utilize constraints that do not necessarily hold for floating point numbers. Fluctuat uses an abstract domain based on affine arithmetic that depends on linear forms and provides a poor abstraction for non-linear operators. Astree utilizes several abstract domains such as octagons and interval linear forms that both rely on linear relationships between variables, another poor fit for non-linear operators. [18, 17]. The Gappa proof assistant supports automatic error tracking using interval arithmetic [9]. Since interval arithmetic can only provide a course approximation, Gappa contains a database of rewriting rules that tighten the approximation. However, Gappa's analysis can get stuck and require manual hints from the user.

Proof assistants allow users to automatically generate machine-checkable proofs of bounds on their floating-point error using bespoke functional models of their code. Defining these functional models is a manual process that can prove difficult and requires skilled technicians. PRECiSA [27] is an abstract

interpretation-based tool that focuses on the detection of branch instability using the abstract domain of *conditional error bounds*. This abstract domain uses an infinite-precision real number representation to compare the floating-point control flow. Users must write a model of their desired program in the PVS proof assistant before running the analysis. VCFloat [15] is a Coq extension that automatically computes round-off error bounds on floating point expressions. Users must write a functional model of their code in Coq, and then prove the model accurately represents their program using a tool such as the Verified Software Toolchain [2].

Some techniques symbolically change the floating-point values in a program to a model that explicitly contains error terms then the error bound is determined by optimizing for the maximum error over the possible values of the parameters. FPTaylor [25] performs the symbolic transformation and then approximates and simplifies the function using Taylor series while approximating the error terms with interval arithmetic. FPTaylor's, symbolic approach applies a general transformation of floating point terms that does not take exactness properties of floating point numbers such as Sterbenz's lemma. Lee et al. [16] utilize the exactness properties of floating point numbers by defining inference rules that take advantage of the tighter error bounds provided by particular cases. Their model uses a linear form for the error terms which does not align with quadratic error terms. This difficulty is handled by approximating the quadratic terms with a linearization function. This work is implemented in Mathematica and does not provide a tool for analyzing new codebases. Both of these techniques focus on straight-line code and cannot handle control flow.

Finding an accurate model is essential to producing a tight analysis. Requiring users to define a model can significantly slow the pace of software development due to the difficulty involved. Abstract interpretation-based approaches easily allow for the reuse of a well-defined model on new code bases. The step functions abstract domain accurately represents the discrete nature of floating-point error and can take advantage of particular properties of floating-point numbers to improve the precision of the analysis.

10 Future Work

There are some clear directions to continue developing the step functions abstract domain. The underlying value approximation (interval arithmetic) is known to be imprecise. Function calls are noticeably absent from the language being analyzed. The analysis could give the expected error for inputs in addition to the outputs.

The domain relies on the non-relational interval arithmetic as an overapproximation of a variables potential values. There are more precise abstract domains that take into the account the relationship between two variables. Examples of these are the zonotopic abstract domain used in Fluctuat [12] and the octogon abstract domain used in Astreé [6]. Incorporating these domains would require splitting the error function over a more complicated value space. An improved

approximation of variable values would improve the error analysis as it depends on the maximum magnitude of numbers to approximate the round-off error.

Backward error analysis is a technique that allows the user to specify an error threshold and then determines the inputs that will meet that threshold [4]. Backward error analysis provides information to the developer on how they need to constrain their program's inputs to meet the desired error threshold. Adding backwards error analysis requires defining a backwards reachability semantics that takes in a set of output values and computes the set of states that can produce that output. These semantics would then need to be abstracted with the step-function domain.

Function calls have a presence in every programming language. Adding support for function calls would allow the analysis to handle a much broader set of program. Function calls introduce new challenges as they complicate the control flow of the program. Typical approaches to handling function calls in abstract interpretation include abstracting the call stack [14] and computing the effect a function has on inputs [22]. The first technique does not require special consideration of the step-function abstract domain while the second technique requires finding relationships between inputs and outputs that can be determined from the specifics of step-functions.

11 Conclusion

In this paper we presented a promising new abstract domain for analyzing floating-point error, step functions. The domain provides a more granular error analysis for floating-point code by mimicking the behavior of floating-point error. Evaluations show that the granular analysis improves upon existing static analysis tools for certain outputs.

Acknowledgment

This work was funded in part by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Sangeeta Chowdhary and Santosh Nagarakatte. Fast shadow execution for debugging numerical errors using error free transformations. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1845–1872, 2022.

[4] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[6] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*, pages 21–30. Springer, 2005.

[7] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. 2016.

[8] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 235–248, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2010.

[10] Nestor Demeure. *Compromise between precision and performance in high performance computing.* Theses, École Normale supérieure Paris-Saclay, January 2021.

[11] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, jun 2007.

[12] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 232–247. Springer, 2011.

[13] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, sep 2001.

[14] Daniel Kästner and Christian Ferdinand. Proving the absence of stack overflows. In *Computer Safety, Reliability, and Security: 33rd International Conference, SAFECOMP 2014, Florence, Italy, September 10-12, 2014. Proceedings 33*, pages 202–213. Springer, 2014.

[15] Ariel E. Kellison and Andrew W. Appel. Vcfloat2: Floating-point error analysis in coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)*, January 2024. To appear.

[16] Wonyeol Lee, Rahul Sharma, and Alex Aiken. On automatically proving the correctness of math.h implementations. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[17] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*, pages 3–17. Springer, 2004.

[18] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19:31–100, 2006.

[19] Jean-Michel Muller. On the definition of ulp (x). 2005.

[20] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[21] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[22] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Annual Asian Computing Science Conference*, pages 331–345. Springer, 2006.

[23] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[24] R. Skeel. Roundoff error and the patriot missile. *SIAM News*, 1992.

[25] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1), dec 2018.

[26] Pat H Sterbenz. Floating-point computation. *(No Title)*, 1974.

[27] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings 19*, pages 516–537. Springer, 2018.