Expression Simplification Using E-Graphs for Interval Evaluation

Simon Björklund
Abstract

Daisy is a framework for verifying and bounding the magnitudes of rounding errors introduced by floating-point arithmetic in numerical programs. As part of this, Daisy employs a rudimentary algorithm for simplifying expressions derived from the programs. We show that more advanced simplifications allow Daisy to prove tighter, and thereby more accurate error bounds. Furthermore, we design a simplification algorithm using e-graphs and equality saturation to provide these more advanced simplifications, and implement this in a tool named OMELETTE. Expressions can be extracted from OMELETTE using an arbitrary metric. To decide the metric optimal for use in Daisy, we define several and find that prioritizing expressions with a small range of possible values yields the lowest bound of rounding errors.

We also introduce an optimization where rounding errors are computed directly from equivalences proved, and not from a simplified expression. Compared to the current algorithm, we find that OMELETTE yields a 15% more accurate rounding error on average when using this optimization — with some benchmarks in particular showing significant improvement. This improvement comes at the cost of a longer running time as OMELETTE is on average 3.5x slower than the current algorithm.
## Contents

1 Introduction .................................................. 1

2 Background .................................................... 3
   2.1 Interval Arithmetic ........................................ 3
   2.2 Floating-Point Errors and Taylor Analysis ............. 5
   2.3 E-Graphs and Equality Saturation ....................... 6

3 Design .......................................................... 7
   3.1 E-Node and E-Class Intervals ......................... 8
   3.2 Constant Folding ......................................... 9
   3.3 Rewrite Rules ............................................. 10
   3.4 Cost Function ............................................. 12

4 Implementation ................................................. 14
   4.1 E-Class Interval Analysis ............................ 15
   4.2 Rewrite Rules ............................................. 16
   4.3 Short-Circuiting Interval Evaluation .................. 17
   4.4 Omelette Interface ..................................... 17
   4.5 Preprocessing Expressions ............................ 18

5 Evaluation ..................................................... 19
   5.1 Cost Function ............................................. 20
   5.2 Short-Circuiting Interval Evaluation .................. 21
   5.3 Preprocessing ............................................. 22

6 Conclusion ..................................................... 22

7 Future Work .................................................. 23

8 Related Work ................................................ 25

A Benchmark Names ............................................. 27

B Benchmarks for T-interval-magnitude and T-interval-width 28
1 Introduction

Unlike the mathematical concepts like real numbers or Turing machines that they model, computers must contend with limitations imposed by practicality. Most notably, the infinite tapes of Turing machines are modelled by finite lists of bits in which all pertinent information must fit — including real numbers.

Since there is no way to accurately model the infinite amount of possible real numbers within a computer’s finite memory, computers rely on finite precision approximations [1]. Typically, these approximations consist of 32 or 64 bits, and can in theory store a set of up to $2^{32}$ or $2^{64}$ possible real numbers, respectively. Any real number not contained within this set must therefore be approximated through rounding.

The rounding of real to finite precision numbers can be hard to predict and reason about [3]. Due to this, programs are often written with the assumption that the rounding errors introduced are negligible. However, for many applications — such as scientific computing — this proves insufficient.

This has led to the creation of a number of methods for bounding these errors using static analysis [3, 4]. Given a numeric program and bounding intervals of all input variables, these tools produce an upper-bound for the rounding error introduced. This can be used to verify that programs do not introduce errors beyond a certain accepted interval under expected conditions.

These methods all have their own pros and cons, with differing results and accuracy-to-efficiency trade-offs, and so deciding between them for any particular application can be challenging [4]. Solving this problem is Daisy’s mission statement. According to its tool paper, Daisy was developed “from the ground up to be modular, and thus easily extensible [· · ·] and currently already provides many of today’s state-of-the-art techniques — all in one tool.” [4]

One method implemented in Daisy is of particular interest to this paper: Taylor analysis, which uses Taylor series to approximate the error intervals [3]. One step of Taylor analysis requires simplifying arithmetic expressions derived from the input program. This is achieved using a simplification algorithm, which in some manner applies a set of rewrite rules to the expression that is to be simplified.

Here is an example of such a rewrite rule:

$$X + X \rightarrow 2 \cdot X$$

Here, $X$ represents an unknown value — a placeholder — that is substituted with an element of the expression upon applying the rule. Through placeholders like $X$, rules are generic over the expressions they apply to.

The current simplification algorithm implemented in Daisy works by greedily and recursively applying the set of rules until they no longer modify the ex-

\footnote{In practice, the amount of real numbers representable by the IEEE-754 standard [2] is smaller due to quirks such as NaN, $\pm$ infinity, and aliased bit-patterns.}
pression. A single application of a rule is performed according to the following steps:

1. Check whether the expression conforms to the shape required by the left-hand side. If so, proceed with the next step.
2. Extract concrete values from the expression for each placeholder in the left-hand side.
3. Substitute the placeholders in the right-hand side with these concrete values.
4. Finally, substitute the input expression with the result of the right-hand side from the previous step.

Since the rules are applied greedily — which is to say, they are applied as soon as an applicable rule is found and without any backtracking — this algorithm suffers the following flaws:

- Any particular rule may be beneficial for some expressions but detrimental to others. To solve this conflict, one would either need to accept the cases where the rule may be detrimental, or remove the rule altogether.
- The application of a rule may modify the expression in such a way that it precludes the application of another, potentially more beneficial rule.
- Numeric literals not immediately next to each other in the expression cannot be collated. Such is the case for expressions like $1 + (2 + a)$, which cannot be properly simplified as $3 + a$.
- Designing rules of any considerable complexity in such a system is notoriously difficult [5]. There are therefore a number of obvious rules that are currently missing.

All of these flaws compound and lead to Daisy overestimating the rounding error interval when using Taylor analysis, making it less accurate overall.

This paper details the replacement of this simplification algorithm with a more sophisticated one. The target of this replacement is an algorithm called equality saturation, which efficiently selects the best rewriting of a given expression according to some set of rewrite rules and a cost function, which codifies a metric of how “preferable” any given rewriting is [6].

In replacing the simplification algorithm, we find a more accurate rounding error yielded by Daisy overall. Furthermore, we introduce an optimization where expression intervals are computed using equivalences proved during equality saturation, which we find further improves both rounding errors and runtime.
Contributions:

• We design a set of rewrite rules, several cost functions, and other components needed for the new simplification algorithm.

• We implement them all with Rust in a tool named Omelette using the Egg library [6].

• We perform benchmarks to determine the best cost function and find that AST-size cost has the lowest analysis time average and that interval-width cost has the lowest rounding error average.

• We introduce and implement the short-circuiting optimization, whereby intervals are computed directly from the e-graph and not from an extracted expression.

• We empirically compare the current simplifier to Omelette and find that Omelette yields 15% more accurate rounding errors but has a 248% worse runtime performance, on average.

2 Background

2.1 Interval Arithmetic

Though intervals formally represent a continuous set of values, they are often reduced to the bounds of the set when represented in computer code. For an interval \([a, b]\), these bounds are \(a\) and \(b\), representing the minimum and maximum values contained within the set, respectively. When \(a = b\), the interval is called a point interval.

When variables in an arithmetic expression have unknown values in known intervals, interval arithmetic may be used to determine the interval of the expression as a whole.

In Theory

Since any real number \(r\) can be accurately represented and operated on through the point interval \([r, r]\), interval arithmetic is a superset of real arithmetic. Following this notion, if \(f\) is an \(n\)-ary real function with arguments spanning their respective intervals \(I = \{I_1, I_2, \cdots, I_n\}\), its counterpart under interval arithmetic \(\hat{f}\) is a function of \(I\).

Let \(\text{Im}_f(I)\) be the image of \(f\) when applied to all combinations of values in \(I\):

\[
\text{Im}_f(I) = \{f(i_1, i_2, \cdots, i_n) : [i_1, i_2, \cdots, i_n] \in (I_1 \times I_2 \times \cdots \times I_n)\}
\]  

(1)

Generally, Equation 1 can be used to define \(\hat{f}\) for any given function \(f\):

\footnote{Only closed intervals will be considered in this paper.}
\[ \hat{f}(I) = [\min\{\text{Im}_f(I)\}, \max\{\text{Im}_f(I)\}] \] (2)

Equation 2 may be read in natural language as “the highest and lowest values produced by \( f \) when applied to all combinations of values within the intervals of each argument.”

If \( f \) is monotonic, the bounds of \( \text{Im}_f(I) \) are computed entirely from the bounds of each interval in \( I \). E.g., if \( f \) is a unary monotonic function, \( \hat{f} \) is defined as:

\[ \hat{f}([x_1, x_2]) = [\min\{f(x_1), f(x_2)\}, \max\{f(x_1), f(x_2)\}] \] (3)

Similarly, if \( f \) is a binary monotonic function, \( \hat{f} \) is defined as:

\[ \hat{f}([x_1, x_2], [y_1, y_2]) = \left[ \min \{f(x_1, y_1), f(x_1, y_2), f(x_2, y_1), f(x_2, y_2)\}, \max \{f(x_1, y_1), f(x_1, y_2), f(x_2, y_1), f(x_2, y_2)\} \right] \] (4)

Most algebraic operators are monotonic, and since their behaviours are well understood, the formulas above can often be simplified. This simplification is predicated on knowing which bounds of the arguments yield the bounds of the function.

**Example:** The lowest value of \(-[x_1, x_2]\) is known to be calculated from \( x_2 \) since \( x_2 \geq x_1 \). Likewise, the highest value is known to be calculated from \( x_1 \). This yields \(-[x_1, x_2] = [-x_2, -x_1] \).

Note that when all arguments to a function under interval arithmetic are point intervals, the function emulates the corresponding function under real arithmetic by itself producing a point interval. More formally, let \( f \) be a function of a set of reals \( \{a, b, c, \cdots\} \). Then, its counterpart under interval arithmetic \( \hat{f} \) is a function of \( \{[a, a], [b, b], [c, c], \cdots\} \):

\[ \hat{f}([a, a], [b, b], [c, c], \cdots) = [f(a, b, c, \cdots), f(a, b, c, \cdots)] \] (5)

**In Practice**

The previous section outlines the theoretical ideal of interval arithmetic. However, a practical implementation will require certain approximations due to it being infeasible to consider the entire real domain and image of a function. Crucially, these approximations should be over-approximations, such that an approximated interval subsumes its theoretical ideal.

There are two kinds of approximations required.

The first approximation is caused by the inability to accurately model irrational numbers, and by extension, functions whose domains or images contain irrational numbers. Such is the case for e.g. square root, which yields an irrational when applied to a prime number, or the sine function, for which the only rational yielding another rational is zero. This causes issues for interval
arithmetic when the lower or upper bounds of intervals given to or yielded by functions are irrational. In these cases, the bounds must be approximated.

Depending on how the bounds are stored in memory, the approximated bounds can get nearly\(^3\) arbitrarily close to their irrational counterparts but can never equal them. Instead, the bounds of a real interval \([r_1, r_2]\) may be approximated as \([a_1, a_2]\) such that \(a_1\) and \(a_2\) are two numbers representable by the program that are “close” to \(r_1\) and \(r_2\), and for which \(a_1 \leq r_1\) and \(a_2 \geq r_2\) hold. This way, the approximated interval still contains the irrationals, without being explicitly bounded by them.

The second approximation is caused by the lack of variable correlations. In real arithmetic, two instances of the same variable in an expression signifies that the two operands have the same real value. By the same logic, in interval arithmetic, two instances of a variable signifies that they have the same interval. This leads to the counter-intuitive over-approximation of \(x - x\), where \(x \in [x_1, x_2]\), as 
\[
[x_1, x_2] - [x_1, x_2] = [x_1 - x_2, x_2 - x_1],
\]
and not [0, 0]; the correlation between the first and second operand in the subtraction are not taken into account. This can be remedied at the implementation level by keeping track of variable instances as an extension to interval arithmetic.

### 2.2 Floating-Point Errors and Taylor Analysis

Taylor analysis is a method of bounding the error introduced when approximating some real function using floating-point arithmetic.

Let \(f\) be a real-valued function and let \(\tilde{f}\) be its counterpart under floating-point arithmetic. Then, the magnitude of the absolute error introduced when modelling \(f\) with \(\tilde{f}\) is generally proportional to the values operated on in \(f\). If the values are small, the error is generally small; if the values are large, the error is generally large. This is due to the density of representable floating-point numbers being inversely proportional to magnitude. E.g., when using 64-bit floating-point numbers encoded with the IEEE-754 floating-point standard \([2\), the distance between the value of smallest magnitude and the one directly above it is about \(5e^{-324}\), whereas the distance between the value of greatest magnitude and the one directly below it is about \(2e292\). The error for functions operating on large real values is therefore greater than for those operating on small ones.

As a consequence, the error \(\text{error}_f\) introduced when modelling a real-valued function \(f\), which is over a set of real values \(\{a, b, c, \ldots\}\), with \(\tilde{f}\) is necessarily also a function of \(\{a, b, c, \ldots\}\):

\[
\text{error}_f(a, b, c, \ldots) = f(a, b, c, \ldots) - \tilde{f}(a, b, c, \ldots) \tag{6}
\]

If the values lie on known intervals \(\{A, B, C, \ldots\}\), Equation 6 can be used to compute an upper-bound for the error as:

\[
\max \{|\text{error}_f(a, b, c, \ldots)| : [a, b, c, \ldots] \in (A \times B \times C \times \cdots)\} \tag{7}
\]

\(^3\)Limited by the amount of total memory of the program, though this is rarely a factor in practice.
Which may be read in natural language as “the largest magnitude of error when $f$ is applied to any combination of values in their respective intervals.” In general, for any given $f$ and any given intervals, the maximum error is hard to compute due to how irregular and discontinuous $\tilde{f}$, and by extension, $\text{error}_f$, is [3]. Consequently, Daisy uses a Taylor series approximation of $\text{error}_f$ during Taylor analysis — hence the name “Taylor analysis.”

Once the Taylor series for the error of $f$ has been determined as a function of the arguments of $f$, several methods can be used to determine the maximum possible error given the intervals of all arguments. The most accurate but computationally demanding methods rely on global optimization [3]. Daisy instead opts for less accurate but simpler methods [4], among them being interval arithmetic — which makes the error a subject to the various over-approximations incurred by interval arithmetic. Once an interval $[e_1, e_2]$ of the error is determined, the largest error is computed as $\max\{|e_1|, |e_2|\}$.

Minimizing the over-approximations incurred by interval arithmetic is key to increasing the accuracy. For this, Daisy simplifies the Taylor series before evaluating it; the intuition being that fewer operations (a simpler expression) should yield fewer over-approximations. Although this may be true, not all over-approximations are equal — the best rewriting is therefore one which has the smallest sum of over-approximations, i.e., one for which $\max\{|e_1|, |e_2|\}$ is as small as possible. This is where the Daisy algorithm shows room for improvement.

2.3 E-Graphs and Equality Saturation

An e-graph can represent any set of equivalent expressions in finite space [6]. A vertex in the graph, called an equivalence class (e-class), contains a set of equivalent expression nodes, called e-nodes. Operator-operand relationships in the expression are represented as edges in the e-graph pointing not from the operator e-node to the operand e-node, but from the operator e-node to the operand e-class. This is significant since it encodes the idea that any e-node inside the operand e-class can be equivalently used in the context.

**Example:** Examine the e-graph depicted in Figure 1a. Provided that $a$ is an integer in base 2, it shows the equivalence between $2 + (2 \cdot a)$ and $2 + (a \ll 1)$, where $\ll$ is the bitwise left shift operator.

Another beneficial feature of e-graphs is that e-nodes are not duplicated when reused [6]. This can be seen in Figure 1a, where the e-node representing the number 2 is used in two separate operations, but instanced only once. This allows e-graphs to represent an infinite number of equivalent expressions with only a finite number of vertices and edges. A demonstration of this concept is shown in Figure 1b, which shows the equivalence of all expressions where $a$ multiplied by 1 an arbitrary number of times.

Though e-graphs fundamentally represent equivalent expressions, they can also be used to represent all possible ways to rewrite some baseline expression without changing its semantic meaning [6]. This usage is the basis for equality saturation, an algorithm for generating such an e-graph from the baseline expression and a set of rewrite rules.
The first step of equality saturation is to translate the baseline expression into an e-graph, with each e-node being assigned its own e-class [6]. Call the e-class containing the top-level e-node the root e-class. The only significant change to the expression at this point is the joining of repeated e-nodes to single instances.

The rewrite rules are then iteratively applied to the e-graph. This is achieved by searching for e-classes containing the left-hand side of the rules, and if a match is found, adding to them the respective right-hand sides [6]. Each such application may create new e-nodes, new e-classes, or join already existing e-classes. Note that the application of a rule cannot destroy or remove information present in the e-graph. This property negates the need to backtrack during the rule application process.

Once the application of the rewrite rules no longer modifies the e-graph, it can be declared to represent all equivalent rewritings; it is saturated [6].

A saturated e-graph can then be used to extract particular rewritings that may be of interest [6]. This is usually achieved with a cost function, which assigns a score to the individual rewritings. The one with the lowest cost, or highest score, can be declared “optimal” and extracted for further use.

For this paper, the cost function is implemented as a binary function choosing one of its two operands as optimal. Let choose be one such cost function, and let \{e_1, e_2, \ldots, e_{n-1}, e_n\} be the e-nodes inside an e-class c. Then, the optimal e-node is chosen from c as:

\[
\text{choose}(e_1, \text{choose}(e_2, \text{choose}(\cdots \text{choose}(e_{n-1}, e_n))))
\]  

(8)

The extraction algorithm runs from the bottom up in the e-graph, which is to say, the optimal operand is chosen before the operator is considered. The cost of the operator is then computed as a function of both the operator and its optimal operands. This leads to an issue if the optimal operand does not yield the optimal operator according to the cost function. If such is the case, we say that the cost function is non-monotonic, the usage of which can cause a suboptimal expression to be erroneously extracted.

For non-monotonic cost functions, even though the optimal operands may not necessarily yield the optimal operator, they may still be a useful predictor. How useful a predictor it is depends on the cost function and directly impacts the correctness of it. In some sense, cost functions can be more or less monotonic, and thereby more or less correct for its purpose.

3 Design

The current simplifier implemented in Daisy as part of the Taylor analysis method of interval evaluation is lacking in that there are several classes of simplifications it is unable to perform. This causes it to overestimate the interval of rounding errors for certain programs. The hypothesis posed in this paper states that these issues can be rectified with a more sophisticated simplification algorithm; one powered by e-graphs and equality saturation.

The replacement algorithm is outlined as follows:
1. Build an e-graph from the input expression. See Table 1 for a list of the expression nodes considered by the simplifier.

2. Run equality saturation over the e-graph with a set of rewrite rules, while ascribing a bounding interval to each e-class. The intervals will be used to inform which rules are applicable for a given e-node, and to implement constant folding.

3. Perform extraction according to a cost function. Several are considered in this section.

4. Yield the extracted expression as result.

The implementation of these steps will require the design of an algorithm to evaluate e-nodes and e-classes under interval arithmetic, an algorithm to perform constant folding, a set of rewrite rules, and a cost function. All of these are detailed in this section.

### 3.1 E-Node and E-Class Intervals

Each e-node and e-class is ascribed a bounding interval, containing all possible real values it can take. This is used to allow the definition of rules that are predicated on the intervals of operands in an expression, to implement the cost function, and to implement constant folding. The interval itself is represented as a pair of rationals, representing its lower and upper bound, respectively.

The interval of an e-class is computed based on the intervals of the e-nodes contained within the e-class. The intervals of the e-nodes, in turn, are computed according to the following conditions:

- If the e-node is a rational literal with value $r$, its interval is the point $[r, r]$.
- If the e-node is a variable, its interval must be provided along with the expression before simplification takes place.
- If the e-node is an operation, its interval is computed using interval arithmetic. Let $f$ be the real-valued function the operation represents, and let...
### Table 1: The nodes of the AST processed by Daisy. Arguments for functions are denoted as elements in \{a, b\} for the purposes of displaying the syntax; in practice, they contain nested expressions of the same AST, making the AST a recursive structure.

<table>
<thead>
<tr>
<th>Node</th>
<th>Denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic negation</td>
<td>(-a)</td>
</tr>
<tr>
<td>Square-root</td>
<td>(\sqrt{a})</td>
</tr>
<tr>
<td>Sine</td>
<td>(\sin a)</td>
</tr>
<tr>
<td>Cosine</td>
<td>(\cos a)</td>
</tr>
<tr>
<td>Tangent</td>
<td>(\tan a)</td>
</tr>
<tr>
<td>Natural exponential</td>
<td>(\exp a)</td>
</tr>
<tr>
<td>Natural logarithm</td>
<td>(\ln a)</td>
</tr>
<tr>
<td>Addition</td>
<td>(a + b)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>(a - b)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>(a \cdot b)</td>
</tr>
<tr>
<td>Division</td>
<td>(a / b)</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>(a ^ b)</td>
</tr>
<tr>
<td>Rational literal</td>
<td>Integer or fraction</td>
</tr>
<tr>
<td>Variable</td>
<td>(\in {a, b, c, \cdots, z})</td>
</tr>
</tbody>
</table>

\(\{I_1, I_2, I_3, \cdots\}\) be the respective intervals of its operand e-classes. Then, the interval of the e-node is computed with the counterpart to \(f\) under interval arithmetic: \(\hat{f}(I_1, I_2, I_3, \cdots)\).

Once the bounding intervals for all e-nodes in an e-class have been computed, the interval of the e-class as a whole is computed as the intersection between them. Coming from real arithmetic, one might expect the intervals of all e-nodes inside an e-class to be equivalent since the e-nodes themselves are equivalent, thereby rendering the intersection superfluous. This is not the case in practicality since the interval of each e-node is only an over-approximation. Taking the intersection between them therefore serves to cancel these over-approximations out, yielding the tightest bound of the e-class.

**Example:** Consider an e-class \(c\) containing two e-nodes: \(x - x\), where \(x \in [x_1, x_2]\), and \(0\).

1. The interval of the e-class containing \(x\) is evaluated as \([x_1, x_2]\), and the one containing \(0\) as \([0, 0]\).

2. The interval of \(x - x\) is evaluated using the counterpart to subtraction under interval arithmetic, with \([x_1, x_2]\) as the first and second operand: \([x_1 - x_2, x_2 - x_1]\).

3. The interval of \(c\) is computed as: \([0, 0] \cap [x_1 - x_2, x_2 - x_1]\) = \([0, 0]\).

### 3.2 Constant Folding

*Constant folding* is the means by which a literal expression, i.e., an expression involving only literals, can itself be evaluated as a literal \([7]\). This allows expressions like \(1 + 2 \cdot 3\) to be evaluated as \(7\).
Typically, this would be implemented with rational arithmetic for functions that can be evaluated without approximations. However, since interval arithmetic is a superset of rational arithmetic, and since interval evaluation of e-nodes are required regardless, it can be used to implement constant folding as well.

After an interval has been established for an e-node $e$, constant folding is implemented by checking whether the interval is a point $[r, r]$, in which case the single rational $r$ contained within the interval is the only possible value of $e$. As such, $r$ can itself be declared equivalent to $e$, and subsequently added to the e-class containing $e$ as a new e-node.

In cases where an e-class $c$ can be evaluated to a point $[r, r]$, $r$ can be declared the most efficient representation of $c$ since no further rewrites will improve upon it. As such, all other e-nodes inside $c$ are removed, which saves the performance penalty of having the equality saturation algorithm explore the possible rewritings of them. In a sense, the e-class is forcefully declared saturated.

Not all literal expressions can practically be evaluated to a single point, due to the bounds of the intervals being represented as rationals; irrational bounds cannot be expressed. E.g., even though $\sqrt{2} \in [2, 2]$ evaluates to a point — namely $[\sqrt{2}, \sqrt{2}]$ — this point cannot be represented in a system using rationals. An interval with irrational bounds is therefore approximated by rounding the lower bound to a rational toward $-\infty$, and the upper bound to a rational toward $+\infty$. After this rounding has taken place, the interval is no longer a point, and as such cannot be used for constant folding.

Example: Consider an e-class $c_1$ containing a single e-node $1 + 2 \cdot 3$.

1. The e-classes containing the literals $1$, $2$, and $3$ are evaluated as the point intervals $[1, 1]$, $[2, 2]$, and $[3, 3]$, respectively.
2. The interval of the e-class $c_2$ containing $2 \cdot 3$ is evaluated as $[2, 2] \cdot [3, 3] = [6, 6]$. Since this is a point interval, $6$ is added to $c_2$.
3. The interval of $c_1$ is evaluated as $[1, 1] + [6, 6] = [7, 7]$. Since this too is a point interval, $7$ is added to $c_1$.

3.3 Rewrite Rules

The rewrite rules provided to equality saturation describe how expressions may be transformed. Ultimately, some combination of the rules applied to any given expression should be able to yield the optimal equivalent rewriting of the expression when used in Taylor analysis. The definition of “optimal” is provided in Section 3.4 as the cost function is defined; the purpose of the rules is merely to ensure that the optimal rewriting, whatever it may be, can in some way be derived from the input expression.

How Rules Are Chosen

Applying a rewrite rule to an e-graph during equality saturation cannot remove information already present in the e-graph [6]; the set of all possible rewritings of an expression can never decrease in size as a rule is applied. Therefore, the
only reason to be conservative with the inclusion of any individual rule in the set of rewrite rules is to minimize the runtime of the algorithm.

According to a benchmark conducted by running a set of test-cases several times with different amounts of rules, the algorithm’s runtime scales linearly with the number of rules with a factor of roughly 0.04. Albeit crude, this benchmark shows how small of an impact the number of rules has on overall runtime performance. Therefore, rules are included provided they have at least some tangible use to the simplifier, and without much reasoning beyond that.

The rules are based on common algebraic and trigonometric identities involving the operations in the AST.

Rule Structure

Rules have a right-hand side and a left-hand side. Call these patterns. They have the same set of AST nodes as concrete expressions with one crucial difference: variables range not over rational numbers but instead over other expressions. Call these pattern variables placeholders. Through these, patterns are generic over the expressions they apply to.

The left-hand side pattern describes the general shape the expression must hold for the rule to be applicable. The right-hand side pattern describes the general shape of the expression that can be declared equivalent to the input expression according to the rule. Only placeholders declared in the left-hand side may be referenced in the right-hand side.

Upon applying a rule to an expression $e$, all placeholders in the left-hand side are paired with concrete expressions nested within $e$. All placeholders in the right-hand side are then substituted with these concrete expressions, thereby itself becoming a concrete expression, which is the result of the rule application.

Some properties, particularly those pertaining to the intervals of expressions, cannot easily be encoded in the left-hand side pattern and are therefore expressed as additional predicates over the placeholders declared in the rule. An example of such a predicate may be read in natural language as “the denominator of this division must be non-zero.”

Additionally, some rules may be declared symmetric, which means that both patterns in the rule can act as both left- and right-hand side patterns. Put differently: either side of the rule should be rewritable as the other. This is useful for certain identities where either side may be used in further rewrites. Not all rules may be declared symmetric due to the requirement that all placeholders referenced in the right-hand side must be declared in the left-hand side.

The Rules

For the purposes of this paper, placeholders will be denoted as elements of $\{X, Y, Z\}$. Asymmetric rules will follow the syntax $\text{lhs} \rightarrow \text{rhs}$; symmetric rules will follow the syntax $\text{lhs} \leftrightarrow \text{rhs}$.

---

4See Table 1.
Note that some rules may be derived from others, rendering their inclusion inconsequential. For the most part, these will still be listed for two reasons:

1. We have not formally derived a minimal set of rewrite rules that encode the desired mathematical properties.
2. Sometimes, the derivation of a rewrite rule from others is non-trivial. We therefore list these for pedagogical value.

Exceptions include if the derivation follows from fundamental properties of the operators, such as commutativity. E.g., we will not list both $X \cdot 1 \rightarrow X$ and $1 \cdot X \rightarrow X$.

All rewrite rules utilized either directly or indirectly by the simplifier are listed in Table 2. Note that we define $0 \land 0 = 1$ for parity with Daisy.

### 3.4 Cost Function

The cost function is used when extracting an “optimal” expression from a saturated e-graph [6]. Its purpose is to choose one among all e-nodes in each individual e-class, and is implemented as a binary function choosing one of its two operands as the better one.

Among the simplest conceivable cost functions is one to prioritize overall small expressions, which is implemented by choosing the rewriting with the smallest total node count. Call this the AST-size cost. Indeed, this would yield the most simplified expression. However, this is not necessarily the optimal expression for further processing by Daisy.

Recall that the Taylor analysis implemented in Daisy derives the Taylor series expansion of the floating-point function, simplifies it, and evaluates it using interval arithmetic. Afterward, the rounding error is computed as the largest magnitude of value contained within the interval. Call this the interval magnitude. For an interval $[x_1, x_2]$, its magnitude is computed as $\max \{|x_1|, |x_2|\}$.

The optimal cost function is therefore one which minimizes this value for the simplified expression.

Though using interval magnitude as a cost function directly would intuitively follow, interval magnitude is non-monotonic. This means that the lowest magnitude of the operator is not necessarily computed from the lowest magnitudes of the operands, which can cause a suboptimal expression to be erroneously extracted. However, monotonicity is non-binary, and as such, interval-magnitude cost may still behave correctly for some subset of all expressions.

To empirically determine the best cost function, we therefore define several of them, which are benchmarked and compared in Section 5.1. In addition to AST-size and interval-magnitude cost, we also define interval-width cost, the intuition being that interval width should be a good predictor of magnitude, but may potentially be more monotonic than interval-magnitude cost.

Using either interval-magnitude or interval-width cost in isolation presents an issue. Namely, that both will attribute the same cost to $1+2\cdot3$ as 7, even though the latter rewriting may be preferred. We therefore also introduce a variation
<table>
<thead>
<tr>
<th>Description</th>
<th>Rewrite Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternate form of unary −</td>
<td>−X ↔ −1 · X</td>
</tr>
<tr>
<td>Alternate form of binary −</td>
<td>X − Y ↔ X + −Y</td>
</tr>
<tr>
<td>Alternate form of tangent</td>
<td>tan X ↔ sin X / cos X</td>
</tr>
<tr>
<td>Alternate form of ^</td>
<td>X ^ 2 ↔ X · X</td>
</tr>
<tr>
<td>Commutativity for +</td>
<td>X + Y → Y + X</td>
</tr>
<tr>
<td>Commutativity for ·</td>
<td>X · Y → Y · X</td>
</tr>
<tr>
<td>Associativity for +</td>
<td>X + (Y + Z) ↔ (X + Y) + Z</td>
</tr>
<tr>
<td>Associativity for ·</td>
<td>X · (Y · Z) ↔ (X · Y) · Z</td>
</tr>
<tr>
<td>Identity for +</td>
<td>X + 0 → X</td>
</tr>
<tr>
<td>Identity for −</td>
<td>X − 0 → X</td>
</tr>
<tr>
<td>Identity for ·</td>
<td>X · 1 → X</td>
</tr>
<tr>
<td>Identity for /</td>
<td>X / 1 → X</td>
</tr>
<tr>
<td>Identity for ^</td>
<td>X ^ 1 → X</td>
</tr>
<tr>
<td>Identity for ln and exp</td>
<td>ln (exp X) → X</td>
</tr>
<tr>
<td>Identity for exp and ln</td>
<td>exp (ln X) → X</td>
</tr>
<tr>
<td>Inverse for +</td>
<td>X + −X → 0</td>
</tr>
<tr>
<td>Inverse for binary −</td>
<td>X − X → 0</td>
</tr>
<tr>
<td>Inverse for / (a)</td>
<td>X ^ 0 → 1</td>
</tr>
<tr>
<td>Inverse for ^ (b)</td>
<td>1 ^ X → 1</td>
</tr>
<tr>
<td>Zero for ·</td>
<td>X · 0 → 0</td>
</tr>
<tr>
<td>Zero for /</td>
<td>0 / X → 0 if X ≠ 0</td>
</tr>
<tr>
<td>Constant sin</td>
<td>sin 0 → 0</td>
</tr>
<tr>
<td>Constant cos</td>
<td>cos 0 → 1</td>
</tr>
<tr>
<td>Constant tan</td>
<td>tan 0 → 0</td>
</tr>
<tr>
<td>Constant exp</td>
<td>exp 0 → 1</td>
</tr>
<tr>
<td>Constant ln</td>
<td>ln 1 → 0</td>
</tr>
<tr>
<td>Distributivity of · over +</td>
<td>(X + Y) · Z ↔ (X · Z) + (Y · Z)</td>
</tr>
<tr>
<td>Distributivity of ^ over ·</td>
<td>(X ^ Y) ^ Z ↔ (X ^ Z) · (Y ^ Z)</td>
</tr>
<tr>
<td>Power product</td>
<td>(X ^ Y) · (X ^ Z) ↔ X ^ (Y + Z)</td>
</tr>
<tr>
<td>Power division (a)</td>
<td>(X ^ Y) / (X ^ Z) ↔ X ^ (Y − Z)</td>
</tr>
<tr>
<td>Power division (b)</td>
<td>(X ^ Y) / (Z ^ Y) ↔ (X / Z) ^ Y</td>
</tr>
<tr>
<td>Negative angle for sin</td>
<td>sin(−X) ↔ −sin X</td>
</tr>
<tr>
<td>Negative angle for cos</td>
<td>cos(−X) → cos X</td>
</tr>
<tr>
<td>Double angle for sin</td>
<td>sin(2 · X) ↔ 2 · sin X · cos X</td>
</tr>
<tr>
<td>Double angle for cos</td>
<td>cos(2 · X) ↔ 1 − 2 · (sin X) ^ 2</td>
</tr>
</tbody>
</table>

Table 2: The rewrite rules used by the simplifier.
on both, named tolerant interval-magnitude cost (t-interval-magnitude cost) and tolerant interval-width cost (t-interval-width cost), respectively.

The tolerant variations are defined by first deriving a relative difference between the properties of the two intervals being chosen among. These properties are either the interval magnitudes for t-interval-magnitude cost or interval widths for t-interval-width cost.

Relative difference is simply computed as the absolute difference between two values, scaled with their magnitudes. The intuition to use relative difference as opposed to absolute difference is that we hypothesize that differences between intervals of large magnitudes are of less importance than those with small ones, in terms of the rounding error yielded by them.

For two intervals with the respective properties \(x\) and \(y\), the relative difference is defined as:

\[
\text{relative difference}(x, y) = \begin{cases} 
\frac{|x-y|}{\max\{x,y\}} & \text{if } x \neq y \\
0 & \text{if } x = y
\end{cases}
\]  

(9)

The separate case for when \(x = y\) is required to avoid division by zero when both \(x\) and \(y\) are zero.\(^5\)

The relative difference lies on the interval \([0, 1]\), where a value of zero means that the two values are equal, and a value of one means that either the two values are infinitely far apart, or that one value is zero whereas the other is not.

To choose between two rewrites, we use AST-size cost if the relative difference between their respective interval properties is below some threshold tolerance \(T \in [0, 1]\). Otherwise, the regular interval-width or -magnitude cost is used instead.

When \(T = 0\), the AST-size cost is used only when the interval properties are equal; when \(T = 1\), the AST-size cost is always used. The exact value for \(T\) is here left undefined as it requires experimental data to derive, which is detailed in Section 5.1.

4 Implementation

The new simplification algorithm is implemented in a tool named OMELETTE with the Rust programming language. This tool is integrated into the existing Daisy codebase and replaces the previous simplifier defined therein. OMELETTE is invoked by Daisy via command-line and accepts as arguments the expression to be simplified and a table of the intervals of all variables contained within the expression. It then prints the resulting simplified expression to the terminal, along with its interval, which is read by Daisy for further processing.

Rust was chosen as a development language for OMELETTE primarily due to it having the library Egg, which serves as an interface to e-graphs and equality

\(^5\)Note that interval widths and magnitudes are always greater than or equal to zero; the only way for \(\max\{x, y\}\) to be zero is when \(x = y = 0\).
saturation. This vastly reduces the required complexity of Omelette in terms of the code written.

The Egg library provides the following utilities [6]:

- The e-graph data structure.
- The equality saturation algorithm.
- An interface to define the structure of the AST to be processed. For Omelette, this structure is one copied from Daisy, shown in Table 1.
- An interface to define a set of rewrite rules to transform the AST.
- A means to attach and update arbitrary data to each e-class as a function of the e-nodes contained within it through a mechanism they call “e-class analysis” [6]. For Omelette, this is used to compute the interval evaluation of e-classes.
- Extraction according to a user-defined cost function.

Additionally, in its implementation of equality saturation, Egg boasts an optimization where invariances of the e-graph data structure are not restored upon each rule application, but are rather amortized to certain phases of the equality saturation algorithm [6]. According to results presented by the authors of Egg, this optimization improves performance with a factor of about 21, as compared to the naive implementation [6].

The modules designed in Section 3 are implemented using API provided by Egg. For cost functions, this implementation is relatively straightforward, requiring only the implementation of the functions in question. Other modules, for which the implementation is less straightforward, are detailed further in this section.

4.1 E-Class Interval Analysis

To allow the definition of rules that are dependent on the values of the expressions they apply to, an interval is calculated for and attached to each e-class. Egg provides functionality for this attachment through a mechanism they call “e-class analysis” [6], which allows for attaching arbitrary data to an e-class.

To enable e-class analysis, three functions must be implemented:

- The \texttt{make(enode)} function specifies how analysis data is computed from a newly created e-node.
- The \texttt{join(eclass\_1, eclass\_2)} function specifies how analysis data is computed from two e-classes to be merged.
- The \texttt{modify(eclass)} function allows for arbitrarily modifying a newly merged e-class.

When a new e-node is added to the e-graph, Egg creates a new e-class containing only that e-node. At this point, \texttt{make} is used to compute the analysis data for this e-class, and is a function of the e-node. If this e-node is merged with an existing e-class, \texttt{join} is called to compute the new analysis data of the e-class, and is a function of the analysis data of both e-classes. Conceptually, \texttt{make} assigns the data to each e-node in an e-class, and \texttt{join} folds them to assign the
To implement interval evaluation using this mechanism, all three functions are defined:

- The `make(eNode)` function is defined to compute the interval of `eNode` using interval arithmetic.

- The `join(eClass_x, eClass_y)` function is defined to compute the intersection of the intervals of `eClass_x` and `eClass_y`. Let \([x_1, x_2]\) and \([y_1, y_2]\) denote the intervals associated with the two e-classes, respectively. Then, the intersection is computed as \([z_1, z_2] = [\max\{x_1, y_1\}, \min\{x_2, y_2\}]\). If \(z_2 > z_1\), no intersection between the two intervals was found, and an error is reported.\(^6\)

- If the interval of an e-class is a point-interval, which is to say it contains only a single value, the `modify(eClass)` function is defined to add this value as a literal to `eClass`. Since interval arithmetic is a superset of real arithmetic, this allows the interval analysis to implement constant folding, whereby e-classes containing expressions involving only literals can have the evaluation of that expression added to them. Furthermore, all non-leaf e-nodes are removed from `eClass`, which saves the performance penalty of having to traverse their rewritings.

### 4.2 Rewrite Rules

To declare rules for consumption by equality saturation, Egg provides a macro that parses their proprietary rule syntax to Rust code that instantiates and returns the rule.\(^7\) Notably, this macro is applied one rule at a time; to declare a list of rewrite rules of length \(n\), one would invoke the macro \(n\) times and put the results in a list.

Macro invocations declaring symmetric rules yield not one rule, but two rules in a list; \(lhs \rightarrow rhs\) and \(rhs \rightarrow lhs\). This is troublesome if symmetric and asymmetric rules are to be stored in a single list, as is the case for OMELETTE. In Rust, one list cannot contain objects of two or more different types, due to its strict type system. There are some ways to get around this with varying ergonomics.

One approach considered is to unify the symmetric rules and asymmetric rules to a single common type by placing each asymmetric rule in a list of length one, such that all rules are represented as lists. Afterward, the complete set of rewrite rules can be obtained by concatenating the elements. However, this approach introduces an overhead of a dynamic memory allocation per asymmetric rule, on top of the allocation already required for symmetric rules. Furthermore, it requires a lot of boilerplate logic to distinguish the two types of rules from each other, and to process them individually.

---

\(^6\)This can happen if an incorrect equivalence is derived from the rewrite rules, or if the implementation of interval arithmetic is faulty.

\(^7\)Rust macros are essentially compile-time functions over the Rust AST. Their result is a new Rust expression which the macro invocation is substituted with before further compilation takes place.
Due to this overhead, we instead introduce a new macro that parses the rules in their entirety instead of one at a time. The macro is defined recursively and maintains an accumulator list to which each parsed rule is appended. Once all rules have been parsed, this list is simply returned — all at compile-time. Notably, this allows us to append the two rules yielded by symmetric rule definitions in the accumulator at once, thereby circumventing the need for any dynamic memory allocations at runtime.

Though this optimization naturally reduces the runtime overhead of constructing the rules, it does not likely have a significant impact on performance since the rules only need to be constructed once per instance of the Omelette program. The main benefit is that of readability, since it does not require any boilerplate at the point of rule declaration.

4.3 Short-Circuiting Interval Evaluation

Since bounding intervals in OMELETTE are already computed during rewriting, we introduce an optimization where this interval is returned to Daisy directly, which saves Daisy having to recompute it as part of the Taylor analysis. Call this short-circuiting the interval evaluation. This can be approached in one of two ways.

The first is simply to return the interval of the optimal rewriting once it has been computed. However, this showed no significant improvement in runtime performance, and since the implementation of interval arithmetic should be equivalent between Daisy and OMELETTE, no difference in rounding error was observed for any benchmark.

The second approach, however, turned out to be a more powerful idea; return the interval of the root e-class back to Daisy. The idea being that since the interval ascribed to the root e-class is an intersection between the intervals of all rewritings it represents, it contains a tighter and more accurate interval than those computed for any individual rewriting.\(^8\) In some sense, it utilizes equivalences proved through equality saturation to derive a tighter interval for the input expression.

Furthermore, this approach circumvents the need for extraction since we are no longer interested in any one rewriting. Flaws with non-monotonic cost functions are thereby also circumvented. This is not to say that short-circuiting yields theoretically perfect results as it predicates on tighter intervals yielding smaller rounding errors, which may not be true for all cases.

4.4 Omelette Interface

Though several mechanisms for transferring the required data were considered — e.g., sockets — it was not deemed performance critical enough to warrant their complexity. Instead, a simple command-line interface was chosen for robustness. This also allows for invoking OMELETTE manually via the command-line.

\(^8\)Recall that interval evaluation for e-nodes is fraught with over-approximations, and that the intersection between intervals of equivalent e-nodes serves to cancel these over-approximations out.
The interface needs to transfer two pieces of data from Daisy to Omelette via command-line arguments: the expression to be simplified and the intervals of all variables referenced in the expression. After equality saturation has taken place, the result needs to be transferred back from Omelette to Daisy by printing it to the terminal.

By default, Egg provides a mechanism for converting expressions to and from strings using fully parenthesized prefix notation. Since this is a simple syntax to compose and parse on the Daisy side, it is used to communicate the expressions.

The intervals of all variables are communicated with a proprietary syntax, meant to be human-readable while also being easy to both compose and parse. For the interval \([x_1, x_2]\) of a variable \(x\), this syntax follows the pattern: \(x: [x_1, x_2]\). To communicate the intervals of several variables, this pattern is simply repeated for each variable, delimited with spaces.

All instances of rational values in string form are denoted either as integers, or as a ratio of two integers. In the latter case, the syntax for the rational \(a/b\) follows the pattern \(a/b\). No support for parsing decimals is currently included.

When invoking Omelette via the command-line, the first argument should be the expression to be simplified, followed by arbitrarily many arguments each declaring the interval of a variable. Due to how arguments are delimited on different operating systems, the expression should be surrounded with quotes to ensure it is not interpreted as two or more separate arguments. Similarly, spaces inside the intervals (e.g., after the comma) may cause the argument to be erroneously split in two.

**Example:** To invoke Omelette with the expression \(a \cdot b/a + 1.5\), where \(a \in [-5, -2]\) and \(b \in [-5.5, 7.75]\):

```
$ omelette '(* (/ (* a b) a) 3/2)' a:[-5,-2] b:[-11/2,31/4]
```

### 4.5 Preprocessing Expressions

Since equality saturation does not, by default, remove e-nodes from the e-graph during its execution, the entire space of rewritings of an expression is explored, even if the optimal rewriting could have been discovered with a single application of a rewrite rule.

**Example:** Even though the denominator in \(x \cdot (y \cdot y + 1) / (y \cdot y + 1)\) should get cancelled, leaving only \(x\), equality saturation will still explore its possible rewritings. For example, it would find \(y \cdot y + 1\) equivalent to \(y^2 + 1, 1 + y \cdot y\), and so on.

For particularly big expressions, this needless exploration may incur a runtime performance penalty. We therefore suggest an optimization where simplification is performed twice per expression: first with a subset of all rewrite rules, and then with the entire set of rewrite rules as normal. Call the first simplification preprocessing.
The idea behind preprocessing is that cancellations and other obvious rewrite rules can be applied in isolation before the entire space of rewritings is explored. The hypothesis being that the final simplification would then explore only rewritings that are more likely to be optimal. The subset of rewrite rules used for preprocessing should therefore reasonably be those most likely to yield a more optimal expression when applied.

5 Evaluation

To justify and derive certain decisions made in the implementation of Omelette, as well as to empirically evaluate its performance, we run Daisy’s Taylor analysis over a set of benchmarks. These are given by FPBench, a project that provides standards for tools and benchmarks relating to floating-point computation [8]. At the time of writing, FPBench defines 131 benchmarks, though not all are within the remit of Daisy. Furthermore, some benchmarks define computations that divide by zero or take the root of a negative number. After filtering these out, 96 benchmarks remain and are consequently used for this paper.

Each benchmark consists of one real-valued function along with the intervals of its arguments. One such benchmark, named bspline3, consists of the function \(-\left(u \cdot u \cdot u\right) / 6\), where \(u \in [0, 1]\). Daisy takes functions like it and bounds the rounding errors produced when modelling them using floating-point arithmetic. Generally, this upper-bound will be an over-approximation; our goal is therefore to see a lower, and thereby more accurate, overall upper-bound when using Omelette instead of the current simplifier.

Though the names of the benchmarks allows for comparing the results seen in this project with those of other projects, they are not of direct relevance. Therefore, the benchmarks will be referred to according to arbitrarily ordered identifiers in \{\(b_1, b_2, b_3, \ldots, b_{96}\}\}. See Appendix A for the benchmark names corresponding to each identifier.

From the benchmarks, we collect two data points: the rounding error calculated for each function, and how long it took Daisy to calculate it, referred to as analysis time. Due to the sometimes highly varying magnitudes of the data points between different benchmarks, we use the values yielded by the existing simplifier as baseline. The data is then shown as percentages of this baseline; 100% meaning no difference, and 50% meaning that the data point is half that of the existing simplifier. This makes the data much easier to interpret and reason about, and allows for collecting more useful averages.

When displaying rounding errors from multiple data sets, we filter out benchmarks for which all rounding errors are the same.\(^9\) This is done for readability since a majority of the benchmarks will often yield the same rounding error over several data sets.

Averages are computed as the average percentage a benchmark is to that of the baseline. Regardless of whether they differ from the baseline, all benchmarks are accounted for in averages shown.

\(^9\)We define “the same” in this case to mean within 0.2%, as we find that differences smaller than that are not likely to be seen on a graph.
5.1 Cost Function

In Section 3.4, we defined five cost functions: AST-size, interval-magnitude, interval-width, t-interval-magnitude, and t-interval-width cost, all of which are benchmarked and compared in this section.

Thresholds for Tolerant Cost Functions

Both the t-interval-magnitude and t-interval-width costs have the tunable parameter $T$; the threshold tolerance below which AST-size cost is used. The value for $T$ is hard to derive manually. Instead, we benchmark both costs for several values of $T$ to determine the best ones. Reasonably, they should be relatively small, but seeing the trend for larger values may be interesting as well. We therefore perform benchmarks for $T \in \{0, 0.05, 0.2, 0.4, 0.6\}$.

We find that $T = 0$ yields the best rounding error for both t-interval-magnitude and t-interval-width costs. Generally, the rounding error is observed to be proportional to $T$, though outliers can be seen for certain benchmarks. Most likely, these outliers are a product of the non-monotonicity of the cost functions, whereby a higher value for $T$ extracted a better rounding error simply by happenstance.

Interestingly, no real correlation is observed between values for $T$ and analysis time. As opposed to rounding error data, analysis time is highly susceptible to non-deterministic noise. Conclusions drawn around analysis time should therefore be considered with caution.

See Appendix B for data showing the differences in rounding errors and analysis times for the two tolerant cost functions between different values of $T$.

All Cost Functions

With $T = 0$ established as optimal for both t-interval-magnitude and t-interval-width, we compare all cost functions against each other.

AST-size cost, being the worst predictor of interval magnitude among the cost functions, performs the worst on all but two benchmarks in terms of rounding error. For these two benchmarks, it is counter-intuitively underperformed by interval-magnitude cost. Likewise, t-interval-magnitude cost yields remarkably poor rounding errors. These observations are likely attributable to the non-monotonicity of the cost functions based on interval magnitude. See Figure 2 for a graph showing the differences in rounding error compared between all cost functions.

On average, interval-width and t-interval-width costs yield the best rounding errors, 87.9% and 88.0% that of the baseline, respectively — but also the worst analysis times, 1110.3% and 1599.5%, respectively. The best analysis time is observed for AST-size cost, which can be understood by considering that all other cost functions require an interval evaluation during extraction, which AST-size cost does not. Notably, the costs based on interval magnitude yield much faster analysis times than those based on interval width, with interval-magnitude cost

\[ \text{The case for when } T = 1 \text{ is equivalent to only using AST-size cost, which is benchmarked separately.} \]
Table 3: Average rounding errors and analysis times when using different cost functions.

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>Average Rounding Error</th>
<th>Average Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST-size</td>
<td>90.4%</td>
<td>365.4%</td>
</tr>
<tr>
<td>Interval-magnitude</td>
<td>89.7%</td>
<td>606.0%</td>
</tr>
<tr>
<td>Interval-width</td>
<td>87.9%</td>
<td>1110.3%</td>
</tr>
<tr>
<td>T-interval-magnitude</td>
<td>89.2%</td>
<td>794.9%</td>
</tr>
<tr>
<td>T-interval-width</td>
<td>88.0%</td>
<td>1599.5%</td>
</tr>
</tbody>
</table>

having an analysis time 606.0% that of the baseline and t-interval-magnitude having 794.9%. Conceivably, this could be attributed to differences in the implementation of the numeric operations required.

In general, we observe that cost functions that yield better rounding errors generally also yield worse analysis times, and vice versa. Furthermore, no one cost function yields the best rounding errors over all benchmarks. We therefore conclude that there is no one optimal cost function among the ones here defined, with all cost functions except t-interval-width being viable depending on the desired accuracy-to-efficiency trade-off.

5.2 Short-Circuiting Interval Evaluation

We find that short-circuiting, as detailed in Section 4.3, significantly improves runtime performance, and yields rounding errors that are as good as or better than any cost function.

As observed in Section 5.1, there is no one optimal cost function in terms of rounding error, with some performing the best on some benchmarks but not others. Short-circuiting, by contrast, performs the best on all benchmarks, yielding the same rounding error as the best cost function for all benchmarks but three, for which it yields a better one: \( b_{10}, b_{30}, \) and \( b_{96} \).
On average, rounding errors with short-circuiting are 87.21% that of the baseline, compared to the combined best over all cost functions: 87.22%. The average analysis times with short-circuiting are 348.4% that of the baseline, compared to the 365.4% that was observed for the fastest cost function, AST-size cost.

Though this optimization yields the best results overall, the ability for OMELETTE to yield simplified expressions is retained for future work.

5.3 Preprocessing

In our benchmarking of this optimization, we simply used Daisy’s current simplifier for the preprocessing step. Running the benchmarks, we find that preprocessing yields worse analysis times on average; 117% that which is observed when using only short-circuiting. The reason for this is likely that most benchmarks simply do not yield expressions that benefit from preprocessing.

6 Conclusion

OMELETTE is shown to successfully improve the rounding errors computed by Daisy. Some benchmarks show significant improvements, some show slight improvements, and some — for which Daisy’s current simplifier likely already yields the optimal rewrite — show no improvements what-so-ever. Notably, however, no benchmarks perform worse.

The cost functions defined in Section 3.4 show varying improvements and analysis times. Generally, the improvement over the baseline is observed to be proportional to the analysis time — cost functions that yield more accurate errors take longer to compute. Furthermore, no single cost function yields the lowest rounding errors across all benchmarks, which is to say, there is no one optimal cost function. Likely, this is due to the non-monotonicity of the cost functions.

With the exception of t-interval-width — which is shown to be less effective than interval-width and with a substantial analysis time premium — all cost functions are viable, each having its own accuracy-to-efficiency trade-off.

The cost function with the best rounding errors, but slowest analysis times is t-interval-width. The cost function with the worst rounding errors, but fastest analysis times is AST-size.

Compared to all cost functions, the biggest improvements and best analysis times are observed when using the short-circuiting optimization, whereby equivalences proved through equality saturation are used to derive a tighter interval evaluation. This allows the algorithm to skip extraction, which in turn means that the cost function is rendered obsolete. The problems of non-monotonicity are therefore circumvented by this optimization. Short-circuiting yields the best rounding errors across all benchmarks — on average, 87.21% that of the baseline.

These improvements are not without penalty. OMELETTE is in its current state on average 3.5 times slower than just using Daisy’s simplifier. For many appli-
cations, this may be prohibitively slow. Extrapolating from the extreme differences in analysis time observed between different cost functions in Section 5.1, the interval arithmetic likely accounts for a large percentage of the time spent in OMELETTE. Efforts to optimize the implementation of this may therefore yield a more reasonable analysis time.

For a tangible example of the impact of OMELETTE, consider the carbonGas benchmark. One expression derived from the input is the following unwieldy product, where \( v \in [0.1, 0.5] \) and \( \delta \in [-1.1e{-16}, 1.1e{-16}] \):

\[
0.401 \frac{1000}{v(1 + 0)} \frac{1000}{v(1 + 0)} (v(1 + 0) - 0.0427(1 + 0)) \cdots \\
\cdots (1 + 0)(1 + 0)(1 + 0)(1 + 0)(1 + 0)(1 + 0) \delta
\]

The default Daisy simplifier rewrites the expression as:

\[
0.401 \frac{1000}{v} \frac{1000}{v} (v - 0.0427) \delta
\]

This roughly evaluates to the interval \([-2.1e{-9}, 2.1e{-9}]\).

Omelette, using t-interval-width cost \((T = 0)\),\(^{11}\) rewrites the expression as:

\[
\frac{401000 + \frac{-17122.7}{v} \delta}{v}
\]

This roughly evaluates to the interval \([-4.1e{-10}, 4.1e{-10}]\). This interval has a five times smaller magnitude, which contributes to Daisy computing a more accurate rounding error for the benchmark as a whole.

7 Future Work

This section details some ideas for future improvements of OMELETTE and its integration in Daisy that were never experimented with for a lack of time.

More Efficient Interval Evaluation

The interval arithmetic is implemented over a data type containing a rational lower- and upper-bound. Each rational consists of two dynamically sized integers — the nominator and denominator. This makes deep copies of the rational data type very expensive as it incurs two dynamic memory allocations.

The implementation of rational arithmetic is supplied by a third-party library. A very informal overview of its source code reveals a number of apparent inefficiencies. Notably, many operations perform deep copies of the rational; seemingly without need or reason. Switching to a different library — a more optimized library — may therefore improve the performance of our implementation of interval arithmetic, and thereby also improve the analysis time as a whole.

\(^{11}\)For this one example, t-interval-width and interval-width yield expressions with the same intervals. However, t-interval-width, due to also prioritizing overall small expressions, more intuitively shows the difference in the quality of the simplification.
More Accurate Analysis Time Benchmarks

As opposed to rounding error, analysis time is susceptible to non-deterministic noise and may be influenced by other programs running during the benchmarks. Though the data for analysis times shown in this report is averaged over the set of benchmarks, also averaging over multiple runs of each benchmark would provide more accurate data.

Iteration Limit

The space of possible rewritings of an expression can grow tremendously in size before it is completely saturated. Due to this, Egg provides a mechanism whereby an upper limit is placed on the number of times the rewrite rules may be applied [6]. By default, this is set to 30. A higher value should generally yield more accurate results, but take longer to run. However, the improvement to accuracy is likely not linear. Future benchmarks may therefore show a value for the upper limit that balances accuracy to time for our application.

Named Constants

In order to evaluate more literal expressions involving transcendental functions, named constants such as $\pi$ or Euler's number $e$ could be introduced. Currently, these are unrepresentable by the AST since they are irrational in value. Operations involving $\pi$ or $e$ will therefore have to be approximated, potentially affecting the final rounding error produced by them.

Instead, we propose that they are encoded as separate AST nodes. This would allow the introduction of rewrite rules such as $\sin \pi \to 0$ or $\ln e \to 1$. The interval evaluation of the constants themselves would remain the same as when storing them as rationals,\(^{12}\) but the intervals of the operations using them could see an improvement.

This should be a fairly minor change in Omelette, but may require some work in Daisy since it is a much bigger project.

Applying Omelette To Other Analyses

The focus of this report has been specifically Taylor analysis. However, Daisy provides many other methods for computing rounding errors — and notably, many that also require some form of expression simplification. To achieve this, these methods use simplification algorithms not too dissimilar from the one for Taylor analysis. Plausibly, these could also be replaced with Omelette, and the methods as a whole may show a similar improvement to that of Taylor analysis.

The short-circuiting optimization would likely not be universally applicable, and so cost functions would need to be derived for each use-case.

\(^{12}\)This is to say, approximated.
8 Related Work

Herbie

Herbie is a tool to automatically rewrite numerical programs to minimize the rounding error produced by them [1]. To do this, they utilize e-graphs — though notably, as discovered in our work in Section 3.4, a cost function cannot be used to extract the optimal numerical programs directly when using bottom up extraction. This is due to the difficulties in encoding the expected magnitude of rounding error as a monotonic cost function. Instead, they utilize more advanced heuristics.

Combining E-Graphs with Abstract Interpretation

A paper published by Coward et al. explores the combining of e-graphs with abstract interpretation [9]. In so doing, they present the idea of using the e-class analysis provided by Egg to implement interval arithmetic for real-valued expressions. They, like us, find that using equivalences proved during equality saturation yields a tighter interval than when applying interval arithmetic on its own. Though much of their abstractions are of little relevance to us, this idea led us to the optimization presented in Section 4.3.
References


A Benchmark Names

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Name</th>
<th>Identifier</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>bspline3</td>
<td>b49</td>
<td>mse_problem_3_2_1_negative</td>
</tr>
<tr>
<td>b2</td>
<td>carbonGas</td>
<td>b50</td>
<td>mse_problem_3_2_1_positive</td>
</tr>
<tr>
<td>b3</td>
<td>cartesianToPolar_radius</td>
<td>b51</td>
<td>mse_problem_3_2_2</td>
</tr>
<tr>
<td>b4</td>
<td>complex_sine_cosine</td>
<td>b52</td>
<td>mse_problem_3_2_3</td>
</tr>
<tr>
<td>b5</td>
<td>complex_square_root</td>
<td>b53</td>
<td>mse_problem_3_2_4</td>
</tr>
<tr>
<td>b6</td>
<td>delta</td>
<td>b54</td>
<td>mse_problem_3_2_5</td>
</tr>
<tr>
<td>b7</td>
<td>delta4</td>
<td>b55</td>
<td>mse_problem_3_2_6</td>
</tr>
<tr>
<td>b8</td>
<td>doppler1</td>
<td>b56</td>
<td>mse_problem_3_2_7</td>
</tr>
<tr>
<td>b9</td>
<td>doppler2</td>
<td>b57</td>
<td>mse_problem_3_4_1</td>
</tr>
<tr>
<td>b10</td>
<td>doppler3</td>
<td>b58</td>
<td>mse_problem_3_4_2</td>
</tr>
<tr>
<td>b11</td>
<td>expix</td>
<td>b59</td>
<td>mse_problem_3_4_3</td>
</tr>
<tr>
<td>b12</td>
<td>expix_32</td>
<td>b60</td>
<td>mse_problem_3_4_4</td>
</tr>
<tr>
<td>b13</td>
<td>expix_log</td>
<td>b61</td>
<td>mse_section_3_1</td>
</tr>
<tr>
<td>b14</td>
<td>floudas</td>
<td>b62</td>
<td>mse_section_3_6</td>
</tr>
<tr>
<td>b15</td>
<td>floudas1</td>
<td>b63</td>
<td>nonlin1</td>
</tr>
<tr>
<td>b16</td>
<td>floudas2</td>
<td>b64</td>
<td>nonlin2</td>
</tr>
<tr>
<td>b17</td>
<td>floudas3</td>
<td>b65</td>
<td>pendulum</td>
</tr>
<tr>
<td>b18</td>
<td>hartman3</td>
<td>b66</td>
<td>pid</td>
</tr>
<tr>
<td>b19</td>
<td>hartman6</td>
<td>b67</td>
<td>polarToCartesian_x</td>
</tr>
<tr>
<td>b20</td>
<td>himmilbeau</td>
<td>b68</td>
<td>polarToCartesian_y</td>
</tr>
<tr>
<td>b21</td>
<td>hypot</td>
<td>b69</td>
<td>predatorPrey</td>
</tr>
<tr>
<td>b22</td>
<td>hypot32</td>
<td>b70</td>
<td>rigidBody1</td>
</tr>
<tr>
<td>b23</td>
<td>iAmmodified</td>
<td>b71</td>
<td>rigidBody2</td>
</tr>
<tr>
<td>b24</td>
<td>i6</td>
<td>b72</td>
<td>rump_from_c</td>
</tr>
<tr>
<td>b25</td>
<td>intro_example</td>
<td>b73</td>
<td>rump_revisited</td>
</tr>
<tr>
<td>b26</td>
<td>jacobisMethodX1</td>
<td>b74</td>
<td>rump_with_pow</td>
</tr>
<tr>
<td>b27</td>
<td>jacobisMethodX2</td>
<td>b75</td>
<td>runge_kutta_4</td>
</tr>
<tr>
<td>b28</td>
<td>jacobisMethodX3</td>
<td>b76</td>
<td>sec4_example</td>
</tr>
<tr>
<td>b29</td>
<td>jacobisMethodX4</td>
<td>b77</td>
<td>sine</td>
</tr>
<tr>
<td>b30</td>
<td>jetEngineModified</td>
<td>b78</td>
<td>sineOrder3</td>
</tr>
<tr>
<td>b31</td>
<td>kepler0</td>
<td>b79</td>
<td>sine_newton</td>
</tr>
<tr>
<td>b32</td>
<td>kepler1</td>
<td>b80</td>
<td>sphere</td>
</tr>
<tr>
<td>b33</td>
<td>kepler2</td>
<td>b81</td>
<td>sqrt</td>
</tr>
<tr>
<td>b34</td>
<td>logexp</td>
<td>b82</td>
<td>sqrt_add</td>
</tr>
<tr>
<td>b35</td>
<td>matrixDeterminant</td>
<td>b83</td>
<td>sum</td>
</tr>
<tr>
<td>b36</td>
<td>matrixDeterminant2</td>
<td>b84</td>
<td>test01_sum3</td>
</tr>
<tr>
<td>b37</td>
<td>matrixDeterminant2modified</td>
<td>b85</td>
<td>test02_sum8</td>
</tr>
<tr>
<td>b38</td>
<td>n_bodyModified</td>
<td>b86</td>
<td>test03_nonlin2</td>
</tr>
<tr>
<td>b39</td>
<td>n_bodyModified</td>
<td>b87</td>
<td>test04_dgmon9</td>
</tr>
<tr>
<td>b40</td>
<td>nse_example_3_1</td>
<td>b88</td>
<td>test05_nonlin1_r4</td>
</tr>
<tr>
<td>b41</td>
<td>nse_example_3_10</td>
<td>b89</td>
<td>test05_nonlin1_test2</td>
</tr>
<tr>
<td>b42</td>
<td>nse_example_3_3</td>
<td>b90</td>
<td>test06_sum4_sum1</td>
</tr>
<tr>
<td>b43</td>
<td>nse_example_3_4</td>
<td>b91</td>
<td>test06_sum4_sum2</td>
</tr>
<tr>
<td>b44</td>
<td>nse_example_3_6</td>
<td>b92</td>
<td>turbine1</td>
</tr>
<tr>
<td>b45</td>
<td>nse_example_3_7</td>
<td>b93</td>
<td>turbine2</td>
</tr>
<tr>
<td>b46</td>
<td>nse_example_3_8</td>
<td>b94</td>
<td>turbine3</td>
</tr>
<tr>
<td>b47</td>
<td>nse_example_3_9</td>
<td>b95</td>
<td>verhulst</td>
</tr>
<tr>
<td>b48</td>
<td>nse_g42_positive</td>
<td>b96</td>
<td>x_by_xy</td>
</tr>
</tbody>
</table>

Table 4: The benchmarks corresponding to the identifiers used throughout the paper.
B  Benchmarks for T-interval-magnitude and T-interval-width

Figure 3: The differences in absolute error when using t-interval-magnitude cost for different values of T.

Figure 4: The differences in absolute error when using t-interval-width cost for different values of T.

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>T = 0.0</th>
<th>T = 0.05</th>
<th>T = 0.2</th>
<th>T = 0.4</th>
<th>T = 0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-interval-width</td>
<td>1626%</td>
<td>1636%</td>
<td>1611%</td>
<td>1630%</td>
<td>1626%</td>
</tr>
<tr>
<td>T-interval-magnitude</td>
<td>769%</td>
<td>783%</td>
<td>784%</td>
<td>778%</td>
<td>769%</td>
</tr>
</tbody>
</table>

Table 5: Average analysis times when using t-interval-width and t-interval-magnitude costs for different values of T.