Automatically Improving Accuracy for Floating Point Expressions

Pavel Panchekha Alex Sachez-Stern James R. Wilcox Zachary Tatlock
University of Washington, USA
{pavpan,asnchstr,jrw12,ztatlock}@cs.washington.edu

Abstract

Scientific and engineering applications depend on floating point arithmetic to approximate real arithmetic. This approximation introduces rounding error, which can accumulate to produce unacceptable results. While the numerical methods literature provides techniques to mitigate rounding error, applying these techniques requires manually rearranging expressions and understanding the finer details of floating point arithmetic.

We introduce Herbie, a tool which *automatically* discovers the rewrites experts perform to improve accuracy. Herbie's heuristic search estimates and localizes rounding error using sampled points (rather than static error analysis), applies a database of rules to generate improvements, takes series expansions, and combines improvements for different input regions. We evaluated Herbie on examples from a classic numerical methods textbook, and found that Herbie was able to improve accuracy on each example, some by up to 60 bits, while imposing a median performance overhead of 40%. Colleagues in machine learning have used Herbie to significantly improve the results of a clustering algorithm, and a mathematical library has accepted two patches generated using Herbie.

Categories and Subject Descriptors G.1.0 [*Numerical Analysis*]: General

Keywords Floating point, numerical accuracy, program rewriting

1. Introduction

Floating point rounding errors are notoriously difficult to detect and debug [24, 25, 38]. Rounding errors have led to irreproducibility and even retraction of scientific articles [1–3], legal regulations in finance [15], and distorted stock market indices [29, 34]. Many applications which must produce accurate results, including physical simulators and statistical packages, depend on floating point arithmetic to approximate computations over real numbers. Floating point arithmetic makes these computations feasible, but it also introduces rounding error, which may cause the approximate results to differ from the ideal real-number results by an unacceptable margin.

When these floating point issues are discovered, many developers first try perturbing the code until the answers produced for problematic inputs appear correct [25, 38]. This process can be tedious and frustrating, and may only temporarily mask the error if the test inputs are not representative.

Developers may also respond to rounding error by increasing *precision*, the size of the floating point representation. A developer might replace a 32-bit single precision float with a 64-bit double precision float to try to shift error to lower order bits. But even the largest hardware-supported precision may still exhibit unacceptable rounding error, and increasing precision further would require simulating floating point in software, incurring orders of magnitude slowdown.[1]

Lastly, knowledgeable developers may turn to formal numerical analysis to produce *accurate* programs: programs whose results are close to the ideal real-number results. The numerical analysis literature includes forward and backward error analysis to quantify the error of a program [20, 23], and program transformations which can improve program accuracy [17, 19]. Unfortunately, these techniques often require understanding the subtle details of floating point arithmetic, and the process is still slow and complicated.

As a step toward addressing these challenges, we introduce Herbie, a tool that automatically discovers accuracy-improving program transformations. Herbie's heuristic search randomly samples inputs, localizes error, generates candidate rewrites, and merges rewrites with complementary effects. In order to evaluate the error of a floating-point expression, Herbie first samples input points and compares results computed with floating point to results computed with arbitrary precision. Herbie identifies which operations contribute most to this error by comparing intermediate results from the two computations. Herbie then applies a database of rewrite rules and performs series expansion to generate alternatives for the identified operations. Finally, Herbie combines alternatives that improve accuracy in different input regions to produce a single program that improves accuracy across all regions.

Herbie complements recent work in analyzing and verifying floating-point programs [4, 6, 10]. These tools can help guarantee that a program achieves its specified accuracy bounds. However, when a program is not sufficiently accurate, these tools do not directly help the developer improve the program's accuracy. Herbie helps the programmer by automatically discovering accuracy-improving transformations. While the transformations Herbie discovers typically correspond to techniques from the numerical methods literature, Herbie does not statically analyze programs and so cannot provide worst-case error bound guarantees. If an application requires verified error bounds, the analysis and verification techniques mentioned above can be applied to Herbie's output.

PLDI '15, June 13–17, 2015, Portland, Oregon.

http://dx.doi.org/10.1145/10.1145/2737924.2737959

[1] Even arbitrary precision floating point can exhibit rounding error if the user selects insufficient precision, so the developer needs expertise to carefully select a precision that provides sufficient accuracy [38]. Our approach to selecting this precision is outlined in Section 4.1.

We evaluate Herbie on examples drawn from a classic numerical methods textbook [19] and consider its broader applicability to floating point expressions extracted from a mathematical library as well as formulas from recent scientific articles. Our results demonstrate that Herbie can effectively discover transformations that substantially improve accuracy (recovering up to 60 bits lost to rounding error) while imposing a median overhead of 40%. Furthermore, Herbie has already been applied by colleagues in machine learning who were able to significantly improve the results of a clustering algorithm. Authors of a mathematical library, Math.js, have accepted two patches generated using Herbie, which improved the accuracy of several complex number routines.

This paper contributes Herbie, a tool for automatically improving the accuracy of floating point expressions. Herbie provides the following subsystems:

- A method to evaluate the average accuracy of a floating-point expression (Section 4.1).
- A technique for localizing the source of rounding error (Section 4.3).
- An algorithm to flexibly apply sequences of rewrites and simplify the results (Sections 4.4 and 4.5).
- A Laurent series expander which supports transcendental and non-analytic functions (Section 4.6).
- An approach to inferring regimes where the error behavior of a program differs (Section 4.8).

Herbie and its subsystems also provide a foundation for others to build upon when exploring floating point accuracy issues. We have published the full Herbie implementation.[2]

The rest of the paper describes Herbie in detail. Section 2 provides a brief background on floating point arithmetic. Section 3 illustrates Herbie on a representative example. Section 4 details Herbie's subsystems and describes their role in Herbie's heuristic search for accuracy-improving program transformations. Section 5 illustrates Herbie's effectiveness at correcting real-world floating point inaccuracies. Section 6 evaluates Herbie's effectiveness by considering a suite of textbook rounding error problems, as well as a larger corpus of real-world floating point expressions. Section 7 surveys the most closely related work, and Section 8 considers future directions for Herbie.

2. Floating Point Background

Floating point numbers are a bounded-size approximation of the set of real numbers. Each floating point number represents a real number of the form

$$\pm(1+m)2^e,$$

where m, the *significand* (also called the mantissa), is a k-bit number in $[0,1)$, and e, the *exponent*, is an l-bit signed integer.[3] Floating point numbers come in a variety of precisions; for example, IEEE 754 double-precision floats are represented by a sign bit, a 52 bit significand, and an 11 bit exponent, while single-precision floats are represented by a sign bit, a 23 bit significand, and an 8 bit exponent. Since their exponents are distributed uniformly, floating point values are distributed roughly exponentially, allowing very large and very small values to be represented.

[2] At `http://herbie.uwplse.org`

[3] IEEE 754 floating point also represents a few special values: positive and negative *infinity*, positive and negative *zero*, *not-a-number* error values, and *subnormal* numbers of form $\pm m2^{-(2^{(l-1)}-1)}$.

Floating point operations use a *rounding mode*,[4] a function to convert real numbers to floating-point numbers. Let $\mathsf{F}(r)$ denote the rounded floating point value of real number r and $\mathsf{R}(f)$ denote the real number represented by the floating point value f. The rounding mode must guarantee that $\mathsf{F}(\mathsf{R}(x)) = x$ and also that $\mathsf{R}(\mathsf{F}(x))$ is one of the two closest floating point values to x.

2.1 Error of Floating-Point Functions

Since a floating point value can only exactly represent a real number of the form $\pm(1+m)2^e$, the conversion F must introduce error for some inputs. For real numbers neither too large nor too small (that is, whose logarithm in base 2 is between $-2^{l-1}+2$ and $2^{l-1}-1$), this error is only due to insufficient precision in the significand. Thus, the error is approximately 2^{-k} times smaller than the output itself. For example, the rounding error for reals near one quadrillion is approximately 0.125 in double precision. We write $\mathsf{F}(x) = x + x\epsilon$, where ϵ is the floating point conversion error[5] and is of absolute value less than 2^{-k}, and where applications of F to different inputs will result in different errors ϵ.

Primitive arithmetic operators on floating point numbers such as addition and multiplication are guaranteed to produce accurately rounded results. For example, the floating point sum $x+y$ of floating point values x and y is guaranteed to be equal to the real-number sum of x and y, rounded: $\mathsf{F}(\mathsf{R}(x)+\mathsf{R}(y))$. The addition $x+y$ of two floating point values x and y thus has real value $x + y + (x+y)\epsilon$.

Operators such as exponentiation and trigonometric functions are usually not computed in hardware and must be implemented by libraries.[6] Due to the table maker's dilemma [27], these more complicated functions cannot provide similar accuracy. Instead, the implementation of a mathematical function $f(x_1, x_2, \dots)$ typically guarantees that its result is among the $2u$ closest floating point values to the exact result $\mathsf{F}(f(\mathsf{R}(x_1), \mathsf{R}(x_2), \dots))$ (within u ulps). For example, $\exp(x)$, for a floating point value x, will have value $e^x + ue^x\epsilon$. Typically, u is less than 8, guaranteeing that all but the two or three least-significant bits are correct.

Since the ϵ is small, individual operations are accurate. However, combining these individual operations might still produce inaccurate programs, because floating-point error is not compositional.

2.2 Non-compositional Error

Though individual floating-point operations are accurate, formulas that combine these operations can still be inaccurate. For example, consider the expression $(x+1)-x=1$. The addition introduces error ϵ_1 and produces $x + 1 + (x+1)\epsilon_1$. The subtraction then introduces ϵ_2 and produces

$$1 + (x+1)\epsilon_1 + \epsilon_2 + (x+1)\epsilon_1\epsilon_2.$$

The ϵ_2 term is small compared to the true value 1, but $(x+1)\epsilon_1$ (and, for very large x, even $(x+1)\epsilon_1\epsilon_2$) may be large if x is large. Thus, for large values of x, this expression may have large error: the expression may incorrectly evaluate to 0, or to some large quantity (depending on the rounding mode). So even though all intermediate computations are accurate, the whole expression is inaccurate. Situations where large values are subtracted to produce a small value occur in real-world formulas, such as the quadratic formula (see Section 3); such "catastrophic cancellation" can cause large rounding error.

[4] Applications can choose between either rounding up, down, or toward zero; or rounding to the mathematically closest value, with ties breaking either toward the value with a zero least significant bit, or away from zero.

[5] ϵ is often called "machine epsilon", though it is due to the floating-point representation and precision, not to specifics of the hardware.

[6] The x87 implements these functions in hardware; SSE and NEON do not.

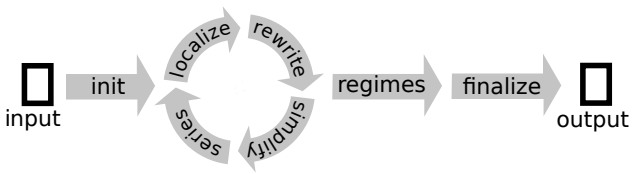

Figure 1. Herbie's process for improving program accuracy.

For $-1 < x < 1$, this expression exhibits little error, but as x grows larger, the error grows as well. In general, complicated expressions often exhibit multiple input regions with distinctly different error behavior. We call this phenomenon *non-uniform error*, and have found that handling it is an essential part of improving the accuracy of floating-point programs.

2.3 Rearrangement

To correct inaccurate formulas, programmers must rearrange their computations to avoid rounding error. These rearrangements are often based on identities of real-number arithmetic. For example, to avoid catastrophic cancellation in $(1+x)-x$, one could reassociate into $1+(x-x)$, which simplifies to 1 and is exactly accurate. Note that many real-number identities, including associativity, are *false* for floating-point arithmetic, which is why they change the floating-point results and thus have the potential to improve accuracy.

The necessary rewrites can be unintuitive. Richard Hamming [19] provides the example of computing $\sqrt{x+1}-\sqrt{x}$. The naive translation to floating point is inaccurate for large positive numbers due to catastrophic cancellation. Hamming's solution involves rearranging this expression into $1/(\sqrt{x+1}+\sqrt{x})$ using the difference-of-squares identity. Herbie's heuristic search is well suited to finding such rearrangements.

3. Overview

Consider the familiar quadratic formula:

$$\frac{-b-\sqrt{b^2-4ac}}{2a} \tag{1}$$

This form, found in high-school algebra textbooks, is inaccurate for negative b and large positive b when translated naively to a floating point computation. This expression is prone to two types of rounding error: for negative b, *cancellation* between $-b$ and $\sqrt{b^2-4ac}$; and for large positive b, *overflow* in computing b^2. (Other types of error may also occur, but we focus on these two errors in this example, as these are the two largest sources of error in this expression.) Herbie can automatically produce a more accurate version using all of Herbie's major subsystems (see Figure 1).

To begin, Herbie must determine that the expression (1) in fact has the floating point inaccuracies described. To do this, Herbie chooses 256 inputs at random and compares the answers produced by (1) in floating-point and arbitrary-precision mode (see Section 4.1). The arbitrary-precision evaluation produces an exact answer, so the difference between the two is caused by rounding error. Having evaluated the error, Herbie can now proceed to avoid it by modifying the program.

For negative b, the error is caused by cancellation at the outer subtraction in the numerator $\left(-b-\sqrt{b^2-4ac}\right)$. For b^2 much larger than a or c, the discriminant $\sqrt{b^2-4ac}$ approximately equals $\sqrt{b^2}$. But for negative b, $\sqrt{b^2}=-b$, so

$$(-b)-\sqrt{b^2-4ac} \approx (-b)-(-b),$$

subtracting two large values to compute a small result and causing catastrophic cancellation.

To avoid this cancellation, Herbie must rewrite equation (1) to avoid subtracting terms with nearby values. Herbie begins by *localizing* the error to the operation responsible for it (see Section 4.3). Herbie does this by computing, for each operation, a *locally approximate* result, the result of applying the operation, as a floating-point operator, to exactly-computed arguments (see Section 4.1). The error of the locally-approximate result measures the extent to which that operation contributes to the error of the program as a whole. In the case of equation (1), localization identifies the subtraction of $-b$ and $\sqrt{b^2-4ac}$ as the main source of error.

Once a source of error has been identified, Herbie attempts to eliminate this error by rewriting the operation which causes it. Herbie does this by applying a database of rewrite rules, each describing basic arithmetic facts. Each rule which matches at the problematic operation is applied to produce a rewritten program; if a rule's top-level pattern matches, but subpatterns do not, Herbie also attempts to rewrite the expression's children until those subpatterns match (Section 4.4). By applying the rewrite rule $x-y \rightsquigarrow (x^2-y^2)/(x+y)$ to the problematic subtraction found by localization, Herbie produces

$$\frac{(-b)^2-\left(\sqrt{b^2-4ac}\right)^2}{(-b)+\sqrt{b^2-4ac}}/2a \tag{2}$$

Other applicable rewrite rules produce ten other rewritten programs. So far, none of these rewrites significantly improve accuracy. However, the program in (2) can be simplified to produce a program that avoids the catastrophic cancellation by algebraically cancelling the two b^2 terms.

Herbie uses a specialized simplification pass to cancel these terms (Section 4.5). Simplification discovers a sequence of five rewrite rules which transform the program in (2) into

$$\frac{4ac}{(-b)+\sqrt{b^2-4ac}}/2a \tag{3}$$

Herbie only simplifies the children of a rewritten node, and so does not further simplify this expression.

For negative b, the program in (3) is much more accurate than the original (1). However, for positive b, it is less accurate than the original, due to cancellation at the addition in the denominator. Herbie notes that programs (1) and (3) are each more accurate on some points than the other, and keeps both as alternatives for further consideration. Eventually, Herbie will have to combine both candidates into a single program that is accurate for both positive and negative b.

Before this combination takes place, Herbie also attempts to fix the inaccuracy of both (1) and (3) for large positive b. When b is positive and greater than approximately 10^{127}, b^2 overflows, resulting in floating-point infinity. This causes the entire expression (1) to evaluate to infinity, even though its actual value is finite. To avoid the problems caused by overflow, a series expansion in b about infinity can be used. Using the approximation $\sqrt{1+x} \approx 1+\frac{1}{2}x$, the numerator of (1) can be rewritten as

$$\frac{-b\sqrt{1-\frac{4ac}{b^2}}-b}{2a} \approx \frac{-b(1-\frac{2ac}{b^2})-b}{2a} = -\frac{b}{a}+\frac{c}{b} \tag{4}$$

Herbie uses series expansion based on symbolic evaluation (Section 4.6) to compute this approximate form, which is more accurate than either (1) or (3) for large positive b.

Herbie has now discovered three separate candidates, each of which is accurate on certain inputs: candidate (3) for negative b, candidate (4) for large positive b, and candidate (1) for all others. To produce an accurate output program, Herbie's regime inference algorithm (Section 4.8) combines the three candidates by inferring

an **if** statement to select between them based on the value of b. The final program produced by Herbie is

$$\begin{cases} \frac{4ac}{-b+\sqrt{b^2-4ac}}/2a & \textbf{if } b < 0 \\ \left(-b-\sqrt{b^2-4ac}\right)\frac{1}{2a} & \textbf{if } 0 \le b \le 10^{127} \\ -\frac{b}{a}+\frac{c}{b} & \textbf{if } 10^{127} < b \end{cases}$$

This program is considerably more accurate than the original (see test case `quadm` in Section 6). The series expansion for large positive b makes it more accurate than the form described in common surveys and textbooks [17, 19], which omit the $10^{127} < b$ case.

In summary, Herbie localizes error to certain operations, applies rewrite rules at those operations, and simplifies the results; series expansion allows Herbie to handle inaccuracies for particularly large or small input values; and regime inference allows these techniques to work together by combining several candidate programs into one.

4. How Herbie Improves Accuracy

Herbie improves program accuracy through a heuristic search, using the accuracy of candidate programs to guide its search. Herbie's goal is to produce a program whose semantics, as a floating point program, matches, as closely as possible, the input program's semantics as a real-number formula.

4.1 Sampling Points

Herbie uses sampled input points to estimate the accuracy of candidate programs. These input points are sampled uniformly from the set of floating point bit patterns. That is, each sampled point is a combination of a random mantissa, exponent, and sign bit. By sampling exponents uniformly, Herbie generates both very small and very large input points, as well as inputs of normal size, allowing Herbie to improve programs which are inaccurate only for particularly large or small values.[7] Herbie uses 256 random sample points to guide its search; we've found that this number of samples estimates program accuracy sufficiently well.

To evaluate the accuracy of a candidate program, Herbie also must know the output generated by the real-number semantics of the original program on the sampled input points. Herbie uses arbitrary precision floating point using GNU MPFR [16] to approximate this output. Arbitrary precision floating point does not immediately banish inaccuracy and rounding error, because a working precision must still be selected. In contrast to arbitrary-precision integers, whose precision can be dynamically expanded as necessary, arbitrary-precision floating-point needs a fixed precision to be chosen. If the chosen precision is too small, the arbitrary-precision evaluation will suffer from rounding error, much like ordinary floating point.

Selecting the right precision is difficult, as accuracy does not improve smoothly with precision. For example, consider the program $((1+x^k)-1)/x^k$ at $x = \frac{1}{2}$. Until k bits of precision are available, the computed answer is 0, even though the correct result is 1. Once k bits are available, the correct answer is computed exactly. A similar pattern occurs with many real-world programs. To avoid this problem, Herbie increases the working precision until the first 64 bits of the computed answer do not change for *any* sampled input point. We have found this method to select a sufficiently large working precision for arbitrary precision evaluation (see Section 6), allowing us to compute the exact floating point result. As many as 2989 bits are required for computing the exact floating-point results for our test cases (see Section 6).

[7] This sampling is approximately exponential. Uniform distributions over the reals fail to capture the structure of floating-point values and prevent Herbie from improving any but the most trivial examples.

Definition herbie-main(program) :
points := sample-inputs(program)
exacts := evaluate-exact(program, points)
table := make-candidate-table(simplify(program))
repeat N **times**
  candidate := pick-candidate(table)
  locations := sort-by-local-error(all-locations(candidate))
  locations.take(M)
  rewritten := recursive-rewrite(candidate, locations)
  new-candidates := simplify-each(rewritten)
  table.add(new-candidates)
  approximated := series-expansion(candidate, locations)
  table.add(approximated)
return infer-regimes(table).as-program

Figure 2. Herbie chooses sample points and computes exact outputs, and enters the main loop. At each step of the loop, a candidate explored by focusing on expressions with local errors, rewriting those expressions, and simplifying the results. Extra candidates are generated by series expansion. After the loop is done, regime inference combines these candidates into a single program. In our evaluation, we used $N = 3$ and $M = 4$ (see Sections 6.1).

Once sample points are chosen, and exact and floating-point answers are computed, some metric for error is necessary to compare the two. Absolute and relative error are natural mathematical measures, but both of these measures are ill-suited to measuring the error between floating-point values [38]. We follow STOKE [36] in defining error as (the base-2 logarithm of) the number of floating-point values between the approximate and exact answers:

$$\mathcal{E}(x, y) = \log_2 |\{z \in \mathsf{FP} \mid \min(x, y) \le z \le \max(x, y)\}|$$

Intuitively, this counts the number of most-significant bits that the approximate and exact result agree on.[8] A program's error at a point is then the difference between the exactly computed floating-point prefix and the answer computed using floating point semantics. Programs are compared by their average bits of error over all valid inputs. This measure of error is invariant over the input space and avoids special handling for infinite and subnormal values. As a happy by-product, Herbie treats overflow and underflow identically to rounding error of any other kind, and can automatically prevent it.

4.2 The Main Loop

Since the space of possible floating point programs is vast, Herbie does not try to synthesize an accurate program from scratch. Instead, Herbie applies a sequence of rewrite rules to its input, each of which may change the floating-point semantics. These rules are specified as input and output patterns; for example, $x - y \rightsquigarrow (x^2 - y^2)/(x + y)$ is a rule, with x and y matching arbitrary subexpressions. Herbie contains 126 rules, including those for the commutativity, associativity, distributivity, and identity of basic arithmetic operators; fraction arithmetic; laws of squares, square roots, exponents, and logarithms; and some basic facts of trigonometry. Each of these rules is a basic fact of algebra, and incorporates no knowledge of numerical methods. We avoid rewrite rules that aren't true for real-number formulas, so that Herbie does not waste time producing programs that compute expressions unrelated to the input program.[9]

[8] Note that this can be as many as 64 bits (for double-precision values), even though the mantissa is only 53 bits long. This happens if the two values differ by orders of magnitude. For example, if a computation should return 0 but instead returns 1, it has approximately 62 bits of error.

[9] As discussed in Section 6, adding "unsound" rewrite rules would slow Herbie down, but would not impact its output.

```
Definition local-error(expr, points) :
for point ∈ points :
    args := evaluate-exact(expr.children)
    exact-ans := F(expr.operation.apply-exact(args))
    approx-ans := expr.operation.apply-approx(F(args))
    accumulate E(exact-ans, approx-ans)
```

Figure 3. To compute the local error of a subexpression, compute the exact value of its arguments, and evaluate its operator to its arguments both in floating point and exactly. The difference between these two values is the local error at that location.

Avoiding such rules was usually easy, but required some care to avoid false "identities" such as $\sqrt{x^2} = x$, which is true only for positive x.

Herbie uses a greedy, hill-climbing search to apply this database of rules: it tracks a set of candidate programs, applies rules at various locations in these candidates, evaluates each resulting program, and repeats the process on the best candidates. However, a naive implementation of this process would spend too much time on useless rewrites, have difficulty finding rewrites that enable future useful rewrites, and would rarely be able to algebraically cancel terms. So Herbie uses specialized localization, rewriting, and simplification algorithms to prune the set of applicable rewrites, consider sequences of dependent rewrites as a unit, and automatically cancel terms. Furthermore, rewrite rules are not suited to deriving polynomial approximations, so Herbie also has a specialized series expansion pass to handle inaccuracies near 0 and $\pm\infty$. After the main loop finishes, Herbie uses regime inference to infer a program that branches between different candidates based on the input values. The remainder of this section explains each of these techniques in detail.

4.3 Localizing Error

Even small programs admit exponentially many possible rewrites; Herbie prunes this search space by identifying those rewrites which are likely to improve accuracy. To do this, Herbie localizes the error of the program to individual operations and then rewrites the operations responsible for the most error. Localization reflects the intuition that operations which are already accurate can be left alone.

Herbie focuses on operations with high *local error*, the error between an operation's floating-point and exact evaluations when its arguments are computed exactly (see Figure 3). By exactly evaluating arguments, Herbie avoids penalizing operations for errors in their inputs (garbage in, garbage out). For each operation, Herbie averages the local error for all sample points, and focuses its rewrites at the operations with the highest average.

4.4 Recursive Rewrite Pattern Matching

After localizing the error to a particular operation, Herbie applies rewrites from its database to that location. Each rewrite replaces the operation with a different, potentially more accurate, way of computing the same value. One approach would be to simply apply each matching rule; however, this would fail to discover many important sequences of rewrites. A common problem is that an expression may require multiple rewrites to enable a rewrite that actually improves accuracy. For example, consider the expression

$$\left(\frac{1}{x-1} - \frac{2}{x}\right) + \frac{1}{x+1}.$$

Herbie correctly identifies the $(+)$ operator as having the highest local error (it adds terms of similar magnitude and opposite sign). To improve the accuracy of this program, all of the fractions must be placed over a common denominator, and then the numerator must be simplified.

```
Definition recursive-rewrite(expr, target) :
▷ select and where are non-deterministic choice and requirement
select input ⇝ output from RULES
where input.head = expr.operator ∧ output.head = target.head
for (subexpr, subpattern) ∈ zip(expr.children, input.children) :
    if ¬matches(subexpr, subpattern) :
        recursive-rewrite(subexpr, subpattern)
where matches(expr, input)
expr.rewrite(input ⇝ output)
▷ Collect valid non-deterministic executions into a list of candidates
```

Figure 4. To recursively rewrite an expression, pick a rewrite rule which matches the current operator and produces the desired target operator. Recursively rewrite each subexpression that does not match its subpattern in the rule's input pattern. Ensure that the results of rewriting each child now match the chosen rewrite rule; if this rewrite rule repeats a pattern variable, it may not match even after rewriting all subexpressions. Each valid set of choices describes one possible recursive rewrite of the expression.

Herbie has rules for fraction addition and subtraction; however, doing a single fraction addition or subtraction does not significantly change the accuracy of the program, since accuracy loss is caused by a cancellation that occurs when *all* of the fractions are added together. In order to improve the accuracy of this program, Herbie must use the fraction addition/subtraction rules twice: once on the parenthesized subtraction,

$$\left(\frac{1}{x-1} - \frac{2}{x}\right) + \frac{1}{x+1} \rightsquigarrow \frac{x - 2\,(x-1)}{(x-1)x} + \frac{1}{x+1}$$

then again for the remaining addition,

$$\frac{x-2(x-1)}{(x-1)x} + \frac{1}{x+1} \rightsquigarrow \frac{(x-2(x-1))(x+1)+(x-1)x}{(x-1)x(x+1)},$$

which can later be simplified to $2/(x^3 - x)$. Finding this sequence of rewrites by brute force search would be difficult because of the large number of rules that can apply at each step, and the large number of locations at which a rewrite might be necessary. However, in this example and in many others, the first rewrite occurs at a *child* of the focused-upon expression, and enables a later rewrite at the focused-upon expression. Herbie's recursive rewrite pattern matching algorithm (see Figure 4) automatically handles this case by rewriting each subexpression of an expression, recursively, to match its associated pattern in the rule. On the benchmarks from section 6, this recursive algorithm produces dozens of rewrite sequences for each focused location; they vary from one to eight rewrites in length.

4.5 Simplification

After applying a rewrite rule at an expression, it may become possible to cancel terms, and this is often necessary to improve accuracy. For example, as detailed above, Herbie produces the numerator $(x-2(x-1))(x+1)+(x-1)x$, which must be simplified to 2 to reduce error. Simplifying expressions would be difficult with localization and recursive rewriting, since simplification often requires making changes far from the source of the error, so Herbie uses a specialized simplification pass. Simplification is applied after each recursive-rewrite step. It automatically cancels terms, which can otherwise contribute to catastrophic cancellation, and avoids redundant computations, which can accumulate error. Generally, the goal of simplification is to produce a smaller, equivalent program.

Simplification often needs to perform commutations, reassociations, and other transformations which do not themselves simplify expressions, in order to enable rewrites that cancel terms or otherwise simplify the expression. Herbie solves this problem by creating

```
Definition simplify(expr) :
iters := iters-needed(expr)
egraph := make-egraph(expr)
repeat iters times :
  for node ∈ egraph :
    for rule ∈ SIMPLIFY-RULES :
      attempt-apply(rule, node)
return extract-smallest-tree(egraph)

Definition iters-needed(expr) :
if is-leaf(expr) :
  return 0
else :
  sub-iters := map(iters-needed, expr.children)
  iters-at-node := if is-commutative(expr.head) then 2 else 1
  return max(sub-iters) + iters-at-node
```

Figure 5. Herbie simplifies expressions by creating an equivalence graph [31], and applying rewrite rules at all nodes in the graph. The number of times each rewrite rule is applied depends on the height of the expression and the number of commutative operators within it. From the final equivalence graph, Herbie chooses the program represented by the smallest tree. Programs are simplified after each rewrite step, and Herbie simplifies only the children of the node which was most recently rewritten.

an equivalence graph [31] of programs reachable from the input via a small number of rewrites (see Figure 5). The equivalence graph allows the simplification algorithm to implicitly handle dependencies between rewrites. Simplification uses a subset of the rewrite database that includes rules to remove function inverses (as in $(\sqrt{x})^2 \rightsquigarrow x$), cancel like terms (as in $x - x \rightsquigarrow 0$), and rearrange terms (as in $x + (y + z) \rightsquigarrow (x + y) + z$).

Herbie makes three modifications to the traditional equivalence graph algorithm. First, Herbie only simplifies the children of the expression that was rewritten, which usually restricts simplification to small expressions while still allowing the most important rewrites. Second, Herbie allows certain transformations to prune the equivalence graph, removing all other items from their equivalence class; for example, when an expression reduces to a constant value, the equivalence class containing that expression is pruned to contain only the literal value, since a literal is always the simplest way to express a constant value. Third, Herbie does not attempt to expand the equivalence graph to contain all possible sequences of rewrites—Herbie does not attempt to saturate the graph. Instead, Herbie bounds the number of iterations (see iters-needed in Figure 5) to the number necessary to cancel two terms anywhere in the expression. From the final equivalence graph, Herbie chooses the program represented by the smallest tree.

4.6 Series Expansions

Some expressions have rounding error for inputs near zero or infinity, but no expression with better accuracy can be found just by applying rewrite rules. It is often possible to avoid this rounding error by using polynomial approximations. For example, the expression $e^x - 1$ is inaccurate near $x = 0$, since e^x is near 1 for those inputs, leading to catastrophic cancellation. No way of rearranging this expression avoids the cancellation error; however, for x near 0, the polynomial approximation $x + \frac{1}{2}x^2 + \frac{1}{6}x^3$ is accurate. Such approximations improve accuracy in many cases, and often help avoid over- and under-flow.

Herbie's series expansion procedure proceeds from the bottom up: each variable or constant is turned into a trivial series, and each function application combines the series expansions of its arguments as dictated by standard mathematical formulas. A series expansion of an expression e in one variable is represented by an offset d and a stream c of coefficients such that

$$e[x] = c_0 x^{-d} + c_1 x^{1-d} + c_2 x^{2-d} + \cdots$$

Note that the series starts not at a constant term, but at x^{-d}. This allows handling expressions with reciprocal terms, and allows accurate series expansion when two reciprocal terms cancel, such as in the expression $\frac{1}{x} - \cot x$. Each coefficient c_n is a symbolic expression; if a term has no series expansion (such as $e^{1/x}$), it is placed into the constant term c_0 of the series expansion; for example, the series expansion of $e^{1/x} + \sin x$ is

$$e^{1/x} x^0 + 1x^1 + 0x^2 + \frac{1}{3}x^3 + \cdots$$

Series expansions at ∞ are also performed, in which case the exponents in the series count down from x^d instead of up from x^{-d}. For expressions with multiple free variables, an analogous multivariate series expansion is used. When truncating series, Herbie uses the three nonzero terms with the smallest degree; we've found this sufficient for the regimes that series expansions are used in.

4.7 Candidate Programs Table

Between iterations of the core loop, Herbie prunes the set of candidate programs, keeping only the ones that achieve the best accuracy on at least one sample point. These are exactly the programs that will be useful for regime inference. In each iteration of the main loop, Herbie picks a program from the table, uses it to generate new candidate programs, and adds them back to the table, pruning it to a minimal set.

Once a program has been picked from the table, it is marked so that it will not be picked again. This means that eventually all programs that are one step away from any program in the table will be found and iterated on, resulting in a "saturated" table. We found that in practice, running until the table reaches saturation does not give better results than running for 3 iterations.

Herbie stores the set of candidate programs as a pair of maps: one from points to a set of alternatives that are tied for best at that point, and the other from alternatives to the points they are tied for best at. A candidate is added to the set only if it is better at some point than the currently best alternatives for that point. After a candidate is added, there may be existing candidates which are no longer best on any point; Herbie prunes these candidates to a minimal set.

Because programs can have equal accuracy on a given point, pruning to the minimal set of programs is algorithmically challenging. For example, consider a set of three candidates on three points: candidate 1 is best at point 1; candidate 3 at point 3; and all are tied at point 2. Herbie must prune candidate 2, since discarding it does not decrease accuracy. When multiple candidates are tied, picking a minimal set is an instance of Set Cover, which is known to be NP-hard. Herbie uses a variant of the $O(\log n)$ Set Cover approximation algorithm [9] to solve this problem. There are often points with a unique best candidate; these candidates cannot be pruned, so Herbie removes both these candidates and any points they are best at from the Set Cover instance before using the approximation algorithm. Pruning keeps the size of the candidate set small—on the benchmarks from Section 6, we have not seen a candidate set of more than 28 programs, even though as many as 285 programs are generated before pruning.

4.8 Regime Inference

Often no candidate program is most accurate on all inputs; instead, each performs well on some inputs, and not on others. For example, to improve the quadratic formula (discussed in Section 3), Herbie must combine three expressions. A similar pattern occurs in many real-world programs, with different expressions accurate on different

```
Definition infer-regimes(candidates, points) :
for x_i ∈ points :
    best-split_0[x_i] = [regime(best-candidate(−∞, x_i), −∞, x_i)]
for n ∈ ℕ until best-split_{n+1} = best-split_n :
    for x_i ∈ points ∪ {∞} :
        for x_j ∈ points, x_j < x_i :
            extra-regime := regime(best-candidate(x_j, x_i), x_i, x_j)
            option[x_j] := best-split[x_j] ++ [extra-regime]
        best-split_{n+1}[x_i] := lowest-error(option)
        if best-split_n[x_i].error − 1 ≤ best-split_{n+1}[x_i].error :
            best-split_{n+1}[x_i] := best-split_n[x_i]
split := best-split_*[∞]
split.refine-by-binary-search()
return split
```

Figure 6. Regime inference via a dynamic programming algorithm. Instead of computing the best way to split $(-\infty, \infty)$, compute the best way to split $(-\infty, x_i)$, for all x_i. This problem admits a simple dynamic program. The best split of $(-\infty, x_i)$ into $n+1$ regimes is just the best way to split $(-\infty, x_j)$ into n regimes plus one regime between (x_j, x_i); or, it is the best split of $(-\infty, x_i)$ into just n regimes. So, add regimes until the best split does not change; then take the best split of $(-\infty, \infty)$. After the best split is found, the boundary between each pair of regimes is refined by binary search.

input regions, which we call *regimes*. Herbie's *regime inference* algorithm automatically detects which programs to use on which inputs. Regime inference also ensures that accuracy on one input region does not come at the cost of error in another region. This is particularly valuable for series expansions, which by their nature are often accurate only for some input regions.

Herbie finds the optimal set of branches using a variant of the Segmented Least Squares dynamic programming algorithm (as described in Kleinberg and Tardős [26]). The dynamic program computes the optimal set of at most k regimes in $(-\infty, x_i)$, where x_i is a sampled point, which has the optimal substructure property required for dynamic programming; see Figure 6 for details. Once Herbie has determined that a branch should be placed between two sampled points, it uses a binary search on the chosen variable to find the exact location of the regime boundary.

Of course, too many branches are likely to over-fit the sampled points; imagine, for example, a program which uses a different expression for each input point. Regime inference must also balance the potential benefit of adding a branch against the cost of doing so: branches can improve accuracy, but are computationally expensive. This balance is implemented by penalizing programs with branches (by one bit of error per branch) in the regime inference algorithm.

5. Case Studies

This section describes three examples where Herbie improved the accuracy of real-world programs: in two cases, Herbie found an accuracy problem in a numerical library, and produced a fix; in the other, Herbie improved the results of a clustering algorithm.

Complex Square Roots Herbie demonstrated utility on real-world library code by finding an inaccuracy in an open-source JavaScript math library, Math.js [13]. Among other functions, Math.js supports operations on complex numbers. To compute the real part of the square root of a complex number $x + iy$, Math.js used the expression $\frac{1}{2}\sqrt{2\left(\sqrt{x \cdot x + y \cdot y} + x\right)}$, which is a standard mathematical definition. However, for negative x (especially when y is small), this expression is inaccurate. Herbie synthesized a more-accurate form of this expression, which for negative x computes

$$\frac{1}{2}\sqrt{2\frac{y^2}{\sqrt{x \cdot x + y \cdot y} - x}}$$

This improvement was implemented as a patch to Math.js, accepted by the Math.js developers, and released with version 0.27.0 of Math.js [32].

Complex Sine and Cosine Herbie found a second inaccuracy in Math.js when we introduced series expansion as a technique Herbie could apply. Math.js used to compute the imaginary part of the cosine of $x + iy$ with the expression $\frac{1}{2}(\sin x)(e^{-y} - e^{y})$. For small values of y, the two exponentials would cancel catastrophically, causing the result to have a zero for its imaginary part, instead of a small non-zero value. Herbie synthesized a more-accurate form of this function for small values of y, using a series expansion for $e^{-y} - e^{y}$:

$$-(\sin x)\left(y + \frac{1}{6}y^3 + \frac{1}{120}y^5\right)$$

Herbie also found similar improvements to the complex sine routine and to the hyperbolic sine routine. All three improvements were implemented as a patch to Math.js, accepted by the Math.js developers, and released with version 1.2.0 of Math.js [33].

Probabilities in a Clustering Algorithm Herbie has also been useful to practitioners who are not directly interested in numerical accuracy. A colleague researching machine learning recently had difficulties with a Markov chain Monte Carlo update rule in a clustering algorithm: the update rule would produce spurious negative or very large results, leading to poor clustering. Our colleague needed to compute

$$\frac{(\operatorname{sig} s)^{c_p}(1 - \operatorname{sig} s)^{c_n}}{(\operatorname{sig} t)^{c_p}(1 - \operatorname{sig} t)^{c_n}}, \text{where } \operatorname{sig} x = \frac{1}{1 + e^{-x}}$$

Our estimates suggest that this simple encoding produces seventeen bits of error. In an attempt to improve the clustering, our colleague manually manipulated the expression until the performance of clustering algorithm improved; our estimates suggest that this improved variant had ten bits of average error:

$$\left(\frac{1 + e^{-t}}{1 + e^{-s}}\right)^{c_p}\left(\frac{1 + e^{t}}{1 + e^{s}}\right)^{c_n}$$

When we fed the original, naive implementation to Herbie it produced an improved version of the program with only four bits average error:

$$\exp\left(c_p \ln \frac{1 + e^{-t}}{1 + e^{-s}} + c_n \ln \frac{1 - \frac{1}{1+e^{-s}}}{1 - \frac{1}{1+e^{-t}}}\right)$$

Further manipulations do not improve accuracy, so Herbie does not perform them. In this case, Herbie produced superior results with no need for manual algebraic manipulation.

6. Evaluation

In addition to the case studies described above, we evaluated Herbie on benchmarks drawn from Hamming's *Numerical Methods for Scientists and Engineers* (NMSE) [19], a standard textbook for applying numerical analysis to scientific and engineering computations. We also separately evaluate our error estimation technique and our regime inference algorithm, and describe our tests of Herbie's wider applicability.

Our evaluation includes twenty-eight worked examples and problems from Chapter 3, which discusses manually rearranging formulas to improve accuracy, the same task that Herbie automates.

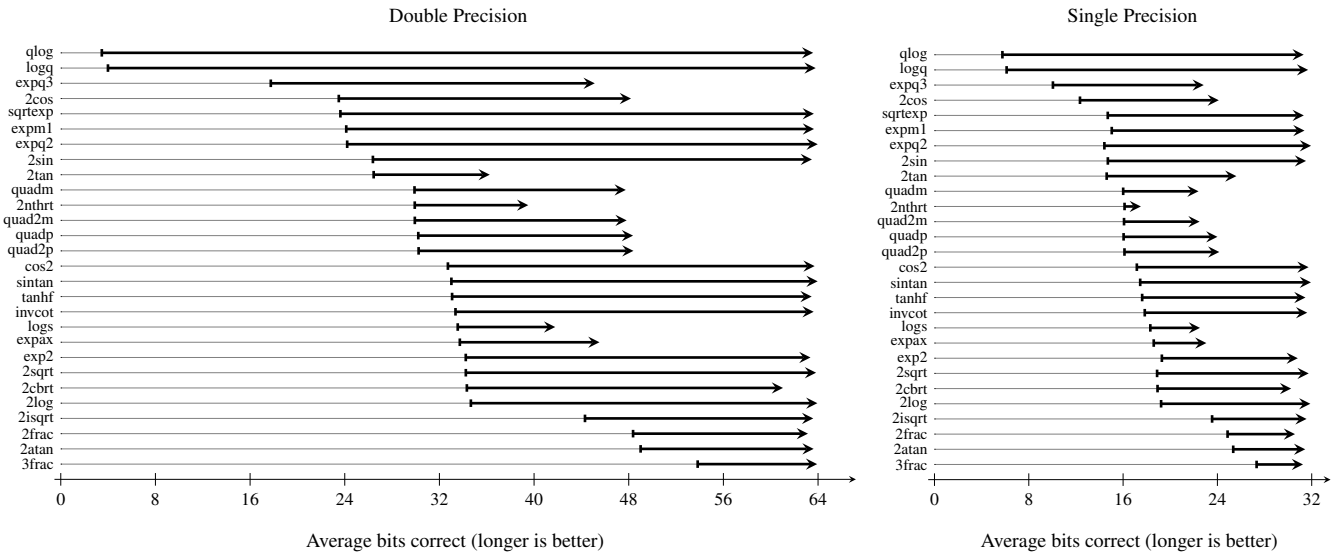

Figure 7. Each row represents the improvement in accuracy achieved by Herbie on a single benchmark. The thick arrow starts at the accuracy of the input program, and ends at the accuracy of Herbie's output. Accuracy is measured by the number of correct output bits, averaged across 100 000 random input points.

Four of the problems and examples are from the introductory section of this chapter, which focuses on the quadratic formula (`quadp`, `quadm`, `quad2p`, `quad2m`); twelve from the section on algebraic rearrangement (`2sqrt`, `2tan`, `3frac`, `2frac`, `2cbrt`, `2cos`, `2log`, `2sin`, `2atan`, `2isqrt`, `tanhf`, `exp2`); eleven from the section on series expansion (`cos2`, `expq3`, `logq`, `qlog`, `sqrtexp`, `sintan`, `2nthrt`, `expm1`, `logs`, `invcot`, `qlog`); and two from the section on branches and regimes (`expq2`, `expax`). Each of Hamming's problems and examples is a single floating-point expression.

6.1 Improving Accuracy

We translated each example into a Herbie input and ran Herbie in a standard configuration. Herbie was run twice: once optimizing for single-precision performance, and once optimizing for double-precision performance. The main text of this section describes only the double-precision results; the single-precision results were similar, as shown in Figure 7.

Herbie is currently implemented in 6.5 KLOC of Racket. The main loop was capped at 3 iterations, and localization was limited to choosing 4 expressions. All experiments were carried out on an Intel Core i5-2400 with 6GB RAM running Arch Linux, Racket 6.1, and GCC 4.9.1. For all of our benchmarks, Herbie ran in under 45 seconds.

Accuracy Our results are shown in Figure 7. For all of our test programs, Herbie improves accuracy by at least one bit. Hamming provides solutions for 11 of the test cases. Herbie's output is less accurate than his solution in 2 cases (`2tan` and `expax`) and more accurate in 3 cases (`2sin`, `quadm`, and `quadp`); in the remaining cases, Herbie's output is as accurate as Hamming's solution.

Overhead We timed the original program and the program produced by Herbie by compiling both to C, using GCC 4.9.1 with flags `-march=native`, `-mtune=native`, `-mfpmath=both`, `-O3`, and `-flto`. Figure 8 is the cumulative distribution of the slow-down for Herbie's output. The median program produced by Herbie was 40% slower. Though Herbie's search does not explicitly balance program speed against accuracy, simplification keeps programs small, explaining the low overhead. Note that in a few cases, the program produced by Herbie is faster than the input program. For these programs, Herbie found a series expansion which was accurate for a large part of the valid input range, replacing an expensive transcendental function with a simple polynomial expression. For other programs, series expansion did not improve program speed because the cost of a branch outweighed the simpler expression.

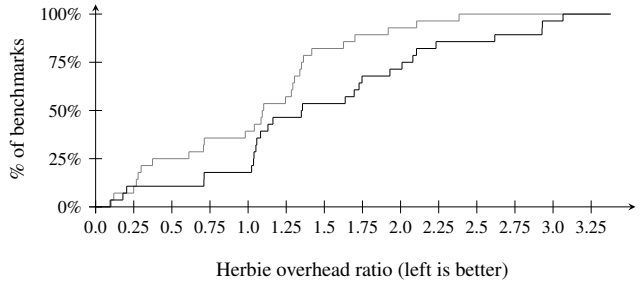

Figure 8. Cumulative distribution of the slow-down from using Herbie. The horizontal axis shows the ratio between the run-time of the input and output programs. The black line is the overhead in Herbie's standard configuration; the gray line is the overhead when regime inference is disabled.

Error Estimation Each program was run on 100 000 points drawn randomly from the set of double-precision floating point values. Results were compared with a ground truth computed via the MPFR arbitrary-precision library [16], which required between 738 and 2989 bits to compute an exact output for all double-precision inputs. Accuracy was measured by the number of bits in error in the approximate output, compared to the exact answer (as in Section 4.1), and was averaged over all points for which the exact answer was a finite floating point value.

6.2 Error Evaluation

For the figures above, we computed each program's average error over 100 000 sampled points. By the Central Limit Theorem, this estimate of average error has a standard error of at most

$64/\sqrt{100\,000} \approx 0.2$ bits.[10] Thus, measured improvements correspond to actual improvements in program accuracy.

To check that sufficiently many bits were used in Herbie's arbitrary-precision evaluations, we compared each against an evaluation with 65 536 bits of precision. In every instance, the answers rounded to double precision were identical, demonstrating that Herbie used sufficiently many bits to compute its ground truth.

For each test case, almost all sampled points either had error less than 8 bits or more than 48 bits; in other words, the distribution of error for different inputs was highly bimodal. Thus, average error roughly estimates how many valid inputs a program can evaluate accurately. Herbie's improvement to average error corresponds to moving points from the high-error to the low-error category.

We also evaluated Herbie's effect on maximum error, and found that Herbie can significantly improve that metric as well. We exhaustively tested Herbie's single-precision output for four test cases by enumerating *all* single-precision floating-point numbers. In some cases, the improvement is modest, such as for `2isqrt`, where Herbie improves maximum error from 29.5 to 29.0 bits. More dramatically, for `2sqrt`, Herbie produces an output program with at most 2 bits of error, even though the input program exhibits as many as 29.8 bits of error. To evaluate max error for expressions with more than one argument, and to evaluate error for programs in double-precision, we also wrote a tool to sample millions of input points[11] and find the maximum error for all sampled points. Of the twenty-eight programs, maximum error improved by more than one bit for seven of them, and by more than a tenth of a bit for two more.

6.3 Regime Inference

We measured the overall effect of regime inference on the accuracy and speed of Herbie's output across our benchmarks. We reran Herbie over our benchmark suite with regime inference disabled, and compared this handicapped Herbie to Herbie in its default configuration. Regime inference helps improve the accuracy of 17 of the 28 programs; Figure 9 graphs this improvement. Many of the large improvements from regime inference are due to the way regime inference enables powerful but specialized transformations. For example, series expansions improve accuracy on many benchmarks, but the candidates produced by series expansion are only accurate on a limited range of input values. Without regime inference, series expansion does not function; many of the improvements in Figure 9 are due not only to regime inference but also series expansion. Since series expansion is used in many of the benchmarks to avoid cancellation and overflow, regime inference affects many test cases. The branches from regime inference added a median overhead of 7% (see Figure 8).

6.4 Extensibility

Real-world computations are likely to involve functions which Herbie does not understand, or for which Herbie's rule database does not contain the necessary rules. So, it is important that Herbie support user extensibility. We tested Herbie's extensibility in two ways: first, we tested that the user can add rules to solve examples that Herbie doesn't solve by default; and second, we tested that adding invalid rules doesn't make Herbie's output less accurate.

The test case `2cbrt` is the expression $\sqrt[3]{x+1} - \sqrt[3]{x}$; Herbie originally did not improve its accuracy, because its database of rewrite rules did not include the difference of cubes formula $x^3 - y^3 = (x-y)(x^2+xy+y^2)$. As a test of Herbie's extensibility, we added rules for the difference of cubes formula to Herbie (which required five lines of code). Herbie, with this extended ruleset, is able to improve the `2cbrt` test case, and has exactly the same results on all others. This suggests that users would be able to add custom, domain-specific rules to handle cases where Herbie's built-in rules are not sufficient.

Adding rules to Herbie would be difficult if incorrect rules could worsen its result. This does not happen: invalid rules do not increase accuracy, so Herbie never keeps the results of applying them. To test this, we added invalid rules to Herbie: for each pair of rules $p_1 \rightsquigarrow q_1$ and $p_2 \rightsquigarrow q_2$, we added the dummy rule $p_1 \rightsquigarrow q_2$, which is usually invalid. Herbie was run, with these dummy rules, on the main suite of 28 benchmarks; it achieved identical results as without these rules, but ran twice as slowly. This suggests that there is no burden on the user to carefully check the validity of rules they add to Herbie, aiding Herbie's extensibility.

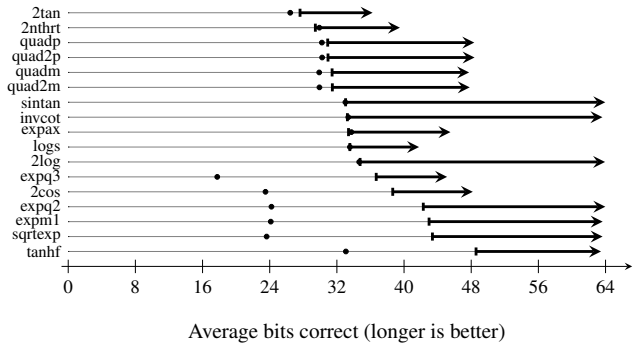

Figure 9. Each arrow represents one of the 17 programs where regime inference improves accuracy; the arrow points from the accuracy without regime inference to the accuracy with regime inference enabled. A dot is drawn at the accuracy of the original program; note that in many of the cases, Herbie is unable to improve accuracy without regime inference.

6.5 Wider Applicability

Many numerical programs are not library functions or textbooks examples, but are instead simulations or data analyses used by scientists, engineers, and mathematicians in the course of their work. The expressions encountered in these programs are more complicated and less structured than the problems in NMSE. Herbie has proven success in improving the accuracy of such programs (see Section 5), but we also made a broader test of Herbie's usefulness on such expressions. We gathered mathematical formulas from a variety of sources and tested both their numerical accuracy and Herbie's ability to fix any inaccuracies. These sources included papers from Volume 89 of Physical Review; standard definitions of mathematical functions, such as hyperbolic functions, or arithmetic on complex numbers; and approximations to special functions like `erf` and ζ.

Of the 118 formulas gathered, we found that 75 exhibited significant floating point inaccuracies. Of these 75 examples, Herbie was able to improve 54 with no modifications and without enlarging its database of rules. This is yet another confirmation that rounding error can arise in the daily practice of scientists and engineers, and that Herbie can often ameliorate these errors. However, it is important to note that for these examples we have not determined if inaccuracies arise for realistic inputs; and, for formulas Herbie was unable to improve, whether a more accurate program exists.

[10] In our experiments, the standard error was an order of magnitude smaller than the conservative upper bound given above.

[11] Each program was sampled for 10 hours, with the most computationally intensive test sampling 3.7 million inputs, and the least computationally intensive sampling 96 million.

7. Related Work

Program Transformations M. Martel proposed a bounded exhaustive search for algebraically-equivalent programs for which a better accuracy bound could be statically proven [28]. Martel's line of work builds an abstract interpretation to bound rounding errors using a sound over-approximation. His technique then generates a set of programs equivalent over the real numbers, and chooses the one with the smallest rounding error. Martel's approach, since it is based on abstract interpretation, generalizes well to programs with loops [22]. However, the bounded exhaustive search limits the program transformations that can be found, since a brute-force search cannot scale with a large database of rewrites. It is also dependent on accurate static analyses for error, which makes supporting transcendental functions difficult. Herbie is fundamentally different from Martel's work in its use of sampling rather than static analysis, its use of a guided search over brute-force enumeration, and its ability to change programs without preserving their real semantics, such as with series expansion.

Genetic programming and SMT synthesis have also been explored for synthesizing fixed-point programs for evaluating polynomial expressions [11, 14]. Herbie does not support fixed-point programs, and uses a variety of analyses, instead of genetic programming or SMT, to prune and direct its search

Numerical Analysis Numerical analysis studies mathematical computations as performed by computers. It includes a vast literature on how to evaluate mathematical functions. The technique of rearranging formulas appears in surveys [17, 24], and in common textbooks [19, 20]. Herbie uses the techniques invented in this literature, but rearranges formulas automatically, avoiding the need for an expert. It is difficult to determine the working precision necessary to accurately evaluate a function [27]. Recent work on this problem allowed the creation of MPFR, an arbitrary-precision floating point library with correct rounding [16]. Herbie uses MPFR internally to exactly evaluate expression.

Verification of Numerical Code Floating point arithmetic is defined in the IEEE 754 standard [21]. However, verification is difficult as programming languages often do not require adherence to this standard [30]. Programs for computing discriminants [6] and solving simple partial differential equations [7] have recently been verified, and the Gappa tool [12] allows certifying numerical error estimates in a proof assistant. Automatic proofs have been also explored: Rosa [10] uses an SMT solver to automatically prove error bounds, FPTaylor [37] uses Taylor expansions and global optimization to find tight over-approximations to floating point error, and Ariadne [4] uses an SMT solver to find inputs which cause floating point overflow. Tools like Rosa could be used to prove that Herbie's output meets an application-specific accuracy specification. Several analysis tools have also been developed: Fluctuat uses abstract interpretation to statically track the error of a floating-point program [18], FPDebug uses a dynamic execution with shadow variables in higher precision [5], and CGRS [8] uses evoluationary search to find inputs that cause high floating-point error.

Optimization of Floating Point Programs Several tools have looked at program transformations to speed up floating-point programs. GCC's `-ffast-math` flag allows rewrites which change floating point results; GCC gives no guarantees about the resulting accuracy.[12] The Stoke super-optimizer supports optimizing floating point programs while guaranteeing that the resulting accuracy is acceptable [36]. Precimonious [35] attempts to decrease the precision of intermediate results to improve run-time and memory use. None of `-ffast-math`, Stoke, and Precimonious improve floating point accuracy.

[12] For our evaluation (Section 6), we did not use this flag because it often undid Herbie's accuracy improvements.

8. Conclusion and Future Work

Herbie automatically improves the accuracy of floating point expressions by randomly sampling inputs, localizing error, generating candidate rewrites, and merging rewrites with complementary effects. Our results demonstrate that Herbie can effectively discover transformations that substantially improve accuracy (recovering up to 60 bits lost to rounding error) while imposing a median overhead of 40%. In the future, we will extend Herbie to reduce error accumulation within loops. We would also like to explore combining Herbie with FPDebug (to extract high-error expressions from programs), FPTaylor and Rosa (to give guarantees of improved error), and STOKE (to do accuracy-aware optimization).

Acknowledgments

We thank Emina Torlak, Eva Darulova, and Marc Andrysco for reading early drafts and offering insightful feedback. We also thank our shepherd Guy Steele and the anonymous reviewers for guidance and valuable suggestions while preparing the final version of this paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1256082.

References

[1] M. Altman and M. McDonald. The robustness of statistical abstractions. *Political Methodology*, 1999.

[2] M. Altman and M. P. McDonald. Replication with attention to numerical accuracy. *Political Analysis*, 11(3):302–307, 2003. URL `http://pan.oxfordjournals.org/content/11/3/302.abstract`.

[3] M. Altman, J. Gill, and M. P. McDonald. *Numerical Issues in Statistical Computing for the Social Scientist*. Springer-Verlag, 2003.

[4] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. POPL '13, 2013.

[5] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. URL `http://doi.acm.org/10.1145/2254064.2254118`.

[6] S. Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58(2):220–225, Feb. 2009. URL `http://hal.inria.fr/inria-00171497/`.

[7] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4): 423–456, Apr. 2013. URL `http://hal.inria.fr/hal-00649240/en/`.

[8] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, and A. Solovyev. Efficient search for inputs causing high floating-point errors. pages 43–52. ACM, 2014.

[9] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979. URL `http://www.jstor.org/stable/3689577`.

[10] E. Darulova and V. Kuncak. Sound compilation of reals. POPL '14, pages 235–248, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. URL `http://doi.acm.org/10.1145/2535838.2535874`.

[11] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. Synthesis of fixed-point programs. EMSOFT '13, pages 22:1–22:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1443-2. URL `http://dl.acm.org/citation.cfm?id=2555754.2555776`.

[12] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1):2:1–2:20, Jan. 2010. `http://gappa.gforge.inria.fr/`.

[13] J. de Jong. math.js: An extensive math library for JavaScript and Node.js, 2013. URL http://mathjs.org/.

[14] H. Eldib and C. Wang. An SMT based method for optimizing arithmetic computations in embedded software code. FMCAD '13, 2013.

[15] European Commission. *The introduction of the euro and the rounding of currency amounts*. Euro papers. European Commission, Directorate General II Economic and Financial Affairs, 1998.

[16] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. URL http://doi.acm.org/10.1145/1236463.1236468.

[17] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991. URL http://doi.acm.org/10.1145/103162.103163.

[18] E. Goubault and S. Putot. Static analysis of finite precision computations. VMCAI'11, pages 232–247, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18274-7. URL http://dl.acm.org/citation.cfm?id=1946284.1946301.

[19] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 2nd edition, 1987.

[20] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002. ISBN 0898715210.

[21] IEEE. IEEE standard for binary floating-point arithmetic. *IEEE Std. 754-2008*, 2008.

[22] A. Ioualalen and M. Martel. Synthesizing accurate floating-point formulas. ASAP '13, pages 113–116, June 2013.

[23] W. Kahan. *A Survey of Error Analysis*. Defense Technical Information Center, 1971. URL http://books.google.com/books?id=dkW7tgAACAAJ.

[24] W. Kahan. Miscalculating area and angles of a needle-like triangle. Technical report, University of California, Berkeley, Mar. 2000. URL http://www.cs.berkeley.edu/~wkahan/Triangle.pdf.

[25] W. Kahan and J. D. Darcy. How Java's floating-point hurts everyone everywhere. Technical report, University of California, Berkeley, June 1998. URL http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[26] J. Kleinberg and E. Tardős. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321295358.

[27] V. Lefévre and J.-M. Muller. The table maker's dilemma: our search for worst cases. Technical report, Ecole Normale Supérieure de Lyon, Oct. 2003. URL http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm.

[28] M. Martel. Program transformation for numerical precision. PEPM '09, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-327-3. URL http://doi.acm.org/10.1145/1480945.1480960.

[29] B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999.

[30] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008. URL http://doi.acm.org/10.1145/1353445.1353446.

[31] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979.

[32] P. Panchekha. Numerical imprecision in complex square root, 2014. URL https://github.com/josdejong/mathjs/pull/208.

[33] P. Panchekha. Accuracy of sinh and complex cos/sin, 2014. URL https://github.com/josdejong/mathjs/pull/247.

[34] K. Quinn. Ever had problems rounding off figures? This stock exchange has. *The Wall Street Journal*, page 37, November 8, 1983.

[35] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. SC '13, 2013.

[36] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating point programs using tunable precision. PLDI '14, 2014.

[37] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. FM'15. Springer, June 2015.

[38] N. Toronto and J. McCarthy. Practically accurate floating-point math. *Computing in Science Engineering*, 16(4):80–95, July 2014.